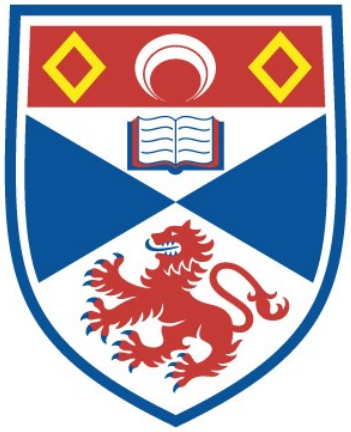
A SEAMLESS FRAMEWORK FOR FORMAL REASONING ON SPECIFICATIONS: MODEL DERIVATION, VERIFICATION AND COMPARISON

Juan Jose Mendoza Santana

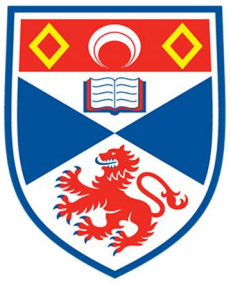
A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2019

A seamless framework for formal reasoning on specifications: model derivation, verification and comparison

Juan Jose Mendoza Santana



University of
St Andrews

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy (PhD)

at the University of St Andrews

January 2019

Candidate's declaration

I, Juan Jose Mendoza Santana, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 37,000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in January 2015.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Date Signature of candidate

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date Signature of supervisor

Permission for publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Juan Jose Mendoza Santana, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

Printed copy

No embargo on print copy.

Electronic copy

No embargo on electronic copy.

Date Signature of candidate

Date Signature of supervisor

Underpinning Research Data or Digital Outputs

Candidate's declaration

I, Juan Jose Mendoza Santana, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Date Signature of candidate

Abstract

While formal methods have been demonstrated to be favourable to the construction of reliable systems, they also present us with several limitations. Most of the efforts regarding formal reasoning are concerned with model correctness for critical systems, while other properties, including model validity, have seen little development, especially in the context of non-critical systems.

We set to advance *model validation* by relating a software model with the corresponding requirements it is intended to capture. This requires us to express both requirements and models in a common formal language, which in turn will enable not only model validation, but also model generation and comparison.

We present a novel framework (TOMM) that integrates the formalization of class diagrams and requirements, along with a set of formal theories to validate, infer, and compare class models. We introduce SpeCNL, a controlled domain independent subset of English sentences, and a document structure named ConSpec. The combination of both allows us to express and formalize functional requirements related to class models.

Our formal framework is accompanied by a proof-of-concept tool that integrates language and image processing libraries, as well as formal methods, to aid the usage and evaluation of our theories. In addition, we provide an implementation that performs partial extraction of relevant information from the graphical representations of class diagrams.

Though different approaches to model validation exist, they assume the existence of formal specifications for the model to be checked. In contrast, our approach has been shown to deal with informal specifications and seamlessly validate, generate and compare class models.

Acknowledgements

This work would not exist if not for the encouragement of Dr Juliana Bowles, who not only motivated me to pursue a PhD but is also the most supportive supervisor anyone can hope for.

Without any doubt, my parents deserve a fair share of gratitude, for thirty years have they been guiding me to become the best person I can be. I thank them for their wisdom and their tireless effort to give me and my sister the best education possible, and for teaching us to always fight for our dreams no matter what they are.

I must also thank my extended family, including all my aunts and uncles, that took care of me during my childhood, and inspired me to go further. Also, to my cousins that are also my brothers and sisters and with whom I have enjoyed so many amazing moments in life. I am thankful because being as close as we are, I know it would have been difficult to deal with my absence during the PhD years.

And last but not least, I want to thank my soulmate Vinodh Rajan S., who has walked this road with me almost from day one, with whom I have overcome the most unimaginable circumstances. Thanks for being my partner in crime, my other half and my beloved boyfriend.

Funding

This work was supported by the Mexican Council of Science and Technology (CONACyT) and the University of St Andrews respectively through the scholarship for postgraduate studies abroad; and the 7th Century scholarship.

Contents

Chapter 1

Introduction

This chapter introduces the motivation for our work, listing the problems we address. They are narrowed down by discussing the research questions the concepts that shape our objectives. Our results are then introduced in the form of individual contributions and their corresponding evaluations. Finally, we provide a description of each chapter as a blueprint for this thesis.

1.1 Motivation

Eight years of personal experience in building industrial software systems has led us to the identification of several problems associated with software models in the development process, particularly, those concerning the maintenance of a consistent relation between requirements and various software models associated with a system. Though software models are to be derived from requirements, in many cases, CASE tools are used to reverse-engineer the models from the existing codebase[48, 70, 164, 169, 181, 224]. This activity does not allow us to establish rigorously whether a given model is valid with respect to the actual requirements of a system, which is a problem we tackle in this research in the form of model validation (which should not be confused with requirements or system validation)[182, 281].

Model validation makes use of formal methods to verify that a given model satisfies a set of self-contained restrictions[24, 72, 74] and is closely related to model checking (which concerns itself only with the internal properties of a model[190]). Current approaches to model validation require the existence of formal specifications, which imposes an additional challenge to software engineers. For this reason, we aim to develop a solution that allows to deal with non-formal specifications, in particular, the specification of functional requirements using elements of natural language.

Model validation is not the only formal activity that has drawn our attention. Model generation is also relevant to us in this context, as it produces the artefacts that will be eventually related to the requirements. Current approaches for model generation can be divided into manual generation, rule-based generation and machine learning generation[2]. Manual approaches[9, 113] are very prone to human error; this is overcome in rule-based approaches[51, 123, 147, 186, 231, 277] by its capabilities to integrate formal methods to generate valid models. This particular aspect of rule-based model generation is the one we develop further in our work.

Machine learning approaches[81, 188, 220, 245, 273, 309] are relatively new, and they depend heavily on the availability of big datasets, which are currently insufficient for model generation from requirements specifications. For this reason, we are concerned about advancing techniques to aid the generation of such datasets.

Even though formal methods have potential applications in industrial software, they have been mostly studied only through their application on life-critical systems[8, 53, 85, 211, 304]; this is partly attributed to the lack of proper competence in software developers to generate mathematical specifications required to use these methods[52, 73, 139, 268]. Having notations that will allow developers to interface with formal methods intuitively contributes to mitigating this problem.

Apart from that, there is a distinct lack of proper integration of formal methods with the existing developmental tool-chain. Most of the existing tools deal with individual aspects of formal verification, whether it be formalizing requirements[28, 84, 208, 246], checking correctness properties[3, 152, 189], or finding proofs[16, 96, 274]. The lack of seamless integration of these tools represents an additional challenge, which also needs to be addressed.

The current limitations of formal methods in industrial software development, coupled with their promising applications in model validation and generation (and by extension, model comparison), have been the prime motivation that has driven us to perform the research presented in this thesis.

1.2 Research Questions

In Section 1.1, we discussed some of the existing problems related to software models, and how formal methods can aid to reduce these problems. These problems are tackled by addressing the following research question.

What are the formal structures and systems required to seamlessly validate, generate and compare UML class diagrams in relation with functional requirements?

To better answer the above underlying research question, we have decomposed it into the following specific questions.

- What fragments of natural language and formal languages can be combined in order to express and formalize functional requirements?
- What axioms have to be satisfied in order to establish the validity of class diagrams with respect to functional requirements within a formal system?
- What inference rules must be applied to generate valid class diagrams from functional requirements?
- What are the equivalence axioms that enable to compare two class diagrams within a formal system?
- How can functional requirements expressed in natural language, and images representing class diagrams, be automatically processed so that they can be used within systems for formal reasoning?

1.3 Objectives

To summarize our motivation, the overall goal of this research is to elaborate on the foundations of a formal framework that will support different reasoning tasks over requirements and software models. This goal is broken down into the following objectives.

1. Develop a requirements specification format to capture functional requirements related to class diagrams.
2. Define a set of formal notations to represent functional requirements and class diagrams seamlessly.
3. Define the formal systems required to achieve class model validation, generation and comparison.
4. Develop a proof-of-concept tool that supports our reasoning framework combining image and natural language processing libraries, together with proof solvers.

5. Evaluate our specification format against existing sets of requirements.
6. Use our proof-of-concept to evaluate our theories for model validation, generation and comparison within a set of clear cases that cover the different expected outcomes.

1.4 Contributions

Our main contribution is a formal framework enabling seamless validation, generation and comparison of class models. Through the development of our framework, we also make two additional contributions to the specification of functional requirements, and the automation of our framework. These contributions are enumerated below.

- A Controlled Natural Language, named SpeCNL, and a document structure, named ConSpec to capture functional requirements.
- A formal framework named TOMM composed of:
  - A set of first-order logic predicates to formalize class diagrams and ConSpec requirements.
  - A theory for validation of class diagram with respect to ConSpec requirements using semantic equivalences.
  - A theory for generating valid class diagrams from ConSpec requirements.
  - A theory to compare class diagrams based on semantic equivalences.
- A python-based proof-of-concept tool named T4TOMM that supports:
  - Partial class model extraction from images of class diagrams using image processing libraries.
  - Formalization of class models into SMT-LIB formal models.
  - Formalization of ConSpec documents into SMT-LIB formal models using libraries for natural language processing.
  - Validation of class models against ConSpec requirements using SMT solvers.
  - Generation of class models from ConSpec requirements using SMT solvers.
  - Comparison of class models using SMT solvers.

Through this thesis, we continuously use the words integrated and integration to refer to the fact that the formalization for ConSpec specifications and class models can be similarly (seamlessly) used for either model validation, model generation or model comparison. This concept also denotes the fact that these activities coexist within TOMM and T4TOMM.

As of April 19th, our paper was accepted for the 15th European Conference on Modelling Foundations and Applications (ECMFA). The conference is dedicated to advancing the applications of Model-Based Engineering and will take place from July 15th to July 19th at the city of Eindhoven in the Netherlands. The paper focusses on the formalization of functional requirements, and class model validation. It covers parts 1 and 2 from the list of contributions.

1.5 Evaluation and Results

Our three distinguishable contributions were evaluated in the following manner. To evaluate of requirements specification format, we made use of a subset of requirements taken from a public repository of requirements documents[289], which were translated into our format. From this process, most of the functional requirements related to class models were successfully translated. Some limitations for our specification format were identified when trying to express complex comparison sentences, such as *"the temperature must be between 10°C and 15°C"* . Besides, we observed some limitation when specifying a sequence of actions, which though nor required for class diagrams, might be useful for sequence diagrams and other models. Further discussion is provided in Section 7.1.

Our formal framework was evaluated through various scenarios containing valid and invalid class models, which were checked against a set of functional requirements. We then checked class models generated from requirements against existing valid models for the same requirements. Model comparison, in turn, was evaluated in terms of existing and generated models that cover the scenarios of equivalence and difference. Our proof-of-concept, described in Chapter 6, showed that the theories we developed satisfy the expected results for validating, generating and comparing classes, attributes, operations, and inheritances. The evaluation of our formal framework and our proof-of-concept is discussed in Section 7.2

Also, we evaluated the capabilities of our proof-of-concept regarding class model extraction from graphical representations. The results showed that despite several limitations, our implementation is capable of extracting some classes, attributes, and operations successfully. This partial extractions,

however, require further manual corrections to be usable. This evaluation is expanded in Section 7.2.5.

Possible extensions for our work were identified through Chapter 7 and summarised in Section 8.2.

1.6 Content of the thesis

The following chapters will elaborate on how we addressed our formulated problem, objectives and research questions.

Chapter 2 starts by presenting an overall picture of this work, and how our contributions are established through a formal reasoning framework and a proof-of-concept. This chapter describes a roadmap for the thesis, which helps the reader to target specific topics if desired.

In **Chapter 3** the context for our research is established. Section 3.1 introduces the basic concepts required to understand our contributions. In Section 3.2 we discuss the state-of-the-art related work. Additionally, in Section 3.3 we present previous work done by us, which though different from the current research, provided us with experience and motivation to develop this work.

Chapter 4 defines all the elements that integrate our controlled language to specify functional requirements, as well as our structure for contract-based specifications. In this chapter, we demonstrate how typical requirements are translated into our specification format.

Chapter 5 describes our formal reasoning framework, which supports the formalisation of class models and specifications, model inference, validation and comparison. This chapter presents the axioms, inference rules and semantics required for each task.

In **Chapter 6**, we discuss the implementation of a proof-of-concept (tool) to support our framework. We describe how class diagrams and requirements specifications are automatically formalised and checked by a formal solver in order to enable automated reasoning. The external dependencies for natural language processing, model verification and image processing are also described here.

Chapter 7 describes the evaluation process for our specification format, our reasoning framework, and our proof-of-concept. The different methodologies to evaluate each of our contributions are described in this chapter, together with the different resources and cases to conduct the evaluation.

Finally, **Chapter 8** summarises our findings, our contributions and the corresponding evaluation. It also reflects on the possible extension of our current work in the future.

Chapter 2

Overview

The overview provided in this chapter will help the reader to understand the structure of this dissertation, and relate each chapter to its corresponding component in our proposed framework. Several concepts and terms are mentioned here; however, they are properly explained until Chapter 3.

The framework described here addresses existing problems regarding software models, namely model validation, model verification and model comparison. The approach we propose to tackle these problems makes use of formal methods to establish a relationship between the elements of functional requirements and class models. Though formal methods are more commonly used in critical systems[53, 85, 130, 149], our framework allows us to demonstrate their usage in the development of non-critical systems defined in Section 3.1.3.

Throughout this thesis, we use as a reference the library system initially described by Callan[62], and then expanded by other authors[42, 128, 142]. The original requirements for this system are shown in Text 2.1, while the original class diagram is shown in Figure 2.1. These requirements and diagram are used to exemplify the application of our specification formats (Chapter 4), the theories that compose our theoretical framework (Chapter 5), the proof-of-concept developed (Chapter 6) and the evaluation of our contributions (Chapter 7).

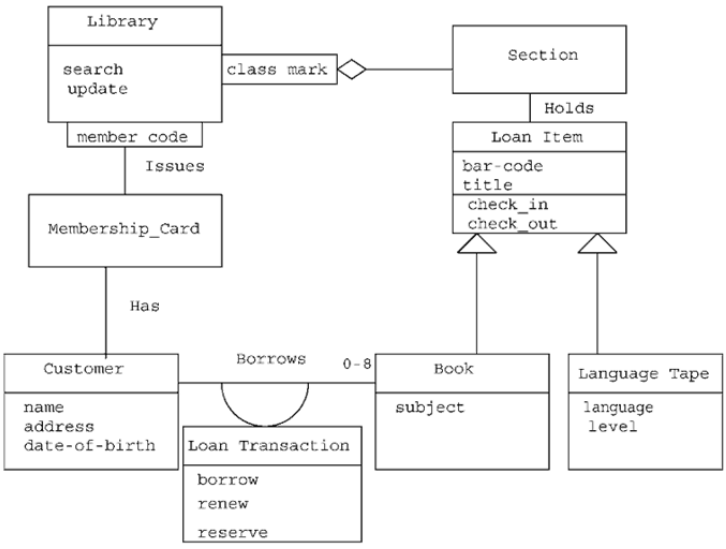


Figure 2.1: Class Diagrams for Library Example

A library issues loan items to customers. Each customer is known as a member and is issued a membership card that shows a unique member number. Along with the membership number, other details on a customer must be kept such as a name, address, and date of birth. The library is made up of a number of subject sections. Each section is denoted by a classification mark. A loan item is uniquely identified by a bar code. There are two types of loan items, language tapes, and books. A language tape has a title language (e.g. French), and level (e.g. beginner). A book has a title, and author(s). A customer may borrow up to a maximum of 8 items. An item can be borrowed, reserved or renewed to extend a current loan. When an item is issued, the customer's membership number is scanned via a bar code reader or entered manually. If the membership is still valid and the number of items on loan less than 8, the book bar code is read, either via the bar code reader or entered manually. If the item can be issued (e.g. not reserved) the item is stamped and then issued. The library must support the facility for an item to be searched and for a daily update of records.

Text 2.1: Original requirements for the Library system described by Callan[62]

2.1 Requirements Specifications

We have defined class models at the centre of our formal framework. However, to successfully perform model validation and generation, it is needed to interact with requirements specifications. Dealing with all possible specification formats, such as diagrams, business documents, records, etc., is infeasible because of the infinite number of variations existing[182, 200, 245, 281, 288]. Hence, we set ourselves to work with formats that can be integrated in a formal framework (see Sections 3.1.7.1 and 3.2.3). These existing alternatives

do no satisfy the constraints that we have set for our framework, hence the need for a new document structure, and a new language to express functional requirements in a way that can be formally related to class diagrams. Both language and document structure are briefly introduced in here, while their details are explained in Chapter 4.

2.1.1 SpeCNL

The language used on a daily basis to communicate with our peers is powerful enough to express emotions, timed events, and an endless number of complex constructs. However, some of these aspects of the language are not relevant when specifying software requirements. SpeCNL is a language based on the grammatical elements of a simplified form of English, to express sentences related to class diagrams. The description of these elements is the purpose of Section 4.1.

2.1.2 ConSpec

SpeCNL allows writing simplified sentences with well-defined structures. However, these sentences alone do not constitute a requirements document. We then propose a document structure, named ConSpec, to delimit the semantics of these sentences in the context of functional requirements. ConSpec is properly described in Section 4.2.

2.2 TOMM

Thinking Of Models and More, or *TOMM* for shorter, is the name given to our framework. The word *thinking* denotes our research activity but also refers to the formal reasoning capabilities that distinguish TOMM, and which are explained in Chapter 5.

TOMM was initially designed to support model validation as a complement for model checking. However, through its development, we observed the opportunity to extend it over other ways of formal reasoning, such as model generation through inference, and model comparison thorough axiom checking. These extensions are part of the current version of TOMM and also demonstrate its flexibility.

In Figure 2.2 we show the interaction of the different theories we have developed for TOMM. In this schematics, theories are represented in boxes, arrows represent interactions, and icons represent the inputs and outputs for each theory.

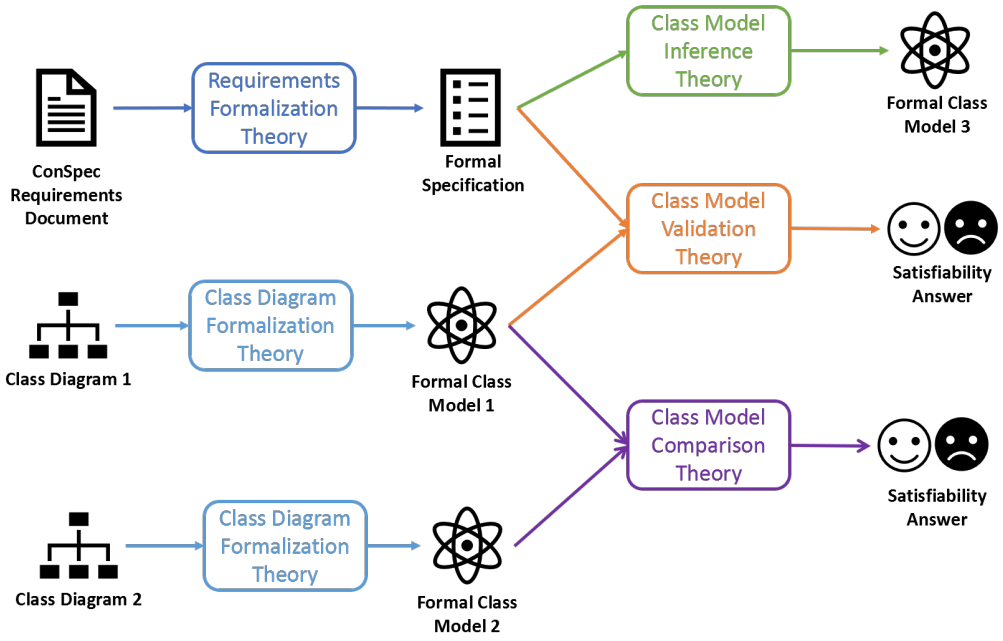


Figure 2.2: Schematic diagram for TOMM

Sections 2.2.1 to 2.2.4 contain an introduction of the theories underlying TOMM, and their detailed description is found in Chapter 5.

2.2.1 Formalization

It is seen in Figure 2.2 that formalization theories precede the usage of the other theories. This step is required to capture both functional requirements and class diagrams in one common language that will enable formal reasoning. In Section 5.1, we discuss how to formalize these artefacts using predicate-logic, which will be introduced in Section 5.1.

2.2.2 Model Validation

In principle, anyone can draw a class diagram, but not every drawn class diagram is guaranteed to be valid with respect to the requirements it is supposed to capture. In the context of TOMM, a diagram is valid if all of its elements are related to some elements in the specification, and if all the elements of the specification are mapped to some elements in the diagram. We define the notions of soundness and completeness in Section 5.3 in order to capture formally capture these conditions.

The validation theory we propose can be used, for instance, to detect

Figure 2.3: The kiss

outdated class diagrams, which is a common problem in software development. It also helps to automate the evaluation process of collections of diagrams, which is a common task when teaching software engineering.

2.2.3 Model Generation

Drawing class diagrams is less about aesthetics than it is about analysis and abstraction; these are intended to encode information about a problem and its corresponding solution. Identifying the elements of a problem and capturing them in a class model is what we do through inference, which is discussed in Section 5.2. This approach to model generation guarantees that inferred models are valid concerning our validation theory.

2.2.4 Model Comparison

By observing the work that Hayez and Klimt have gifted us (Figure 2.3), it becomes evident that it is possible to have more than one representation of one single concept, in this case, a *kiss*.

Similarly, having more than one class diagram representing the same requirements is not unusual, as it is the case with the diagrams of Callan[62] and Kim[171] shown in Figure 2.4

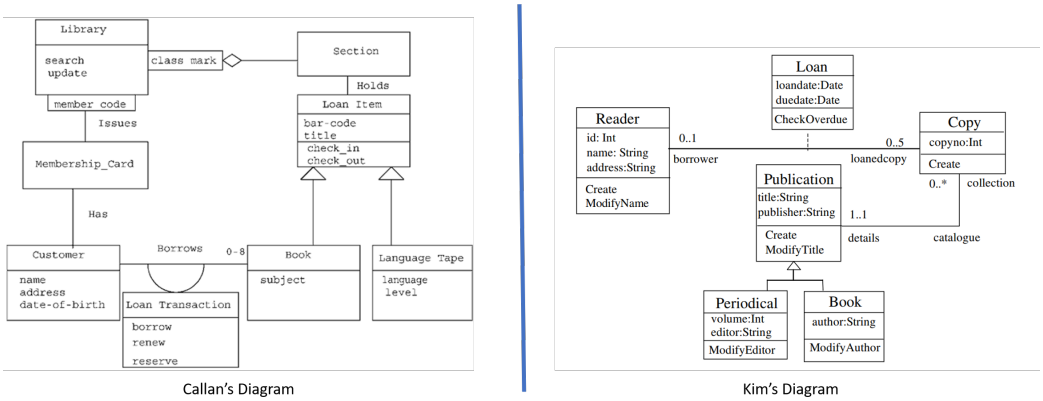


Figure 2.4: Two diagrams representing the same requirements for a library system

In the case of the paintings, it is evident for the observer that though they are not exactly the same, they capture the same concept. However, a more in-depth analysis of their elements must be conducted in order to determine the similarities between two class diagrams. A formal theory to determine model equivalence is the work we present in Section 6.5.

If two or more diagrams are proved to be equivalent, then they can be alternatively used to represent the same requirements, which is desirable when generating diagrams using machine learning techniques (see Section 3.1.8 for further detail).

2.3 T4TOMM

TOMM is a theoretical framework to reason about class diagrams formally. However, the impact of any theory is better measured through practice. Hence the need to develop T4TOMM, which is a proof-of-concept that supports the theories described in Chapter 5, and also deals with the specifications defined in Chapter 4. All of the technical aspects of T4TOMM, including libraries, algorithms and meta-models are detailed in Chapter 6.

Figure 2.5 shows how the types of different input files, whether images, or structured documents (YAML and JSON) are transformed into python meta-models using natural language processing and machine learning. These are then transformed into STM-LIB[33, 34] formal models; this is a formal language that can be interpreted by SMT solvers (see Section 3.1.7.4) in order to generate logic proofs. In this way, class model validation, generation and comparison are achieved.

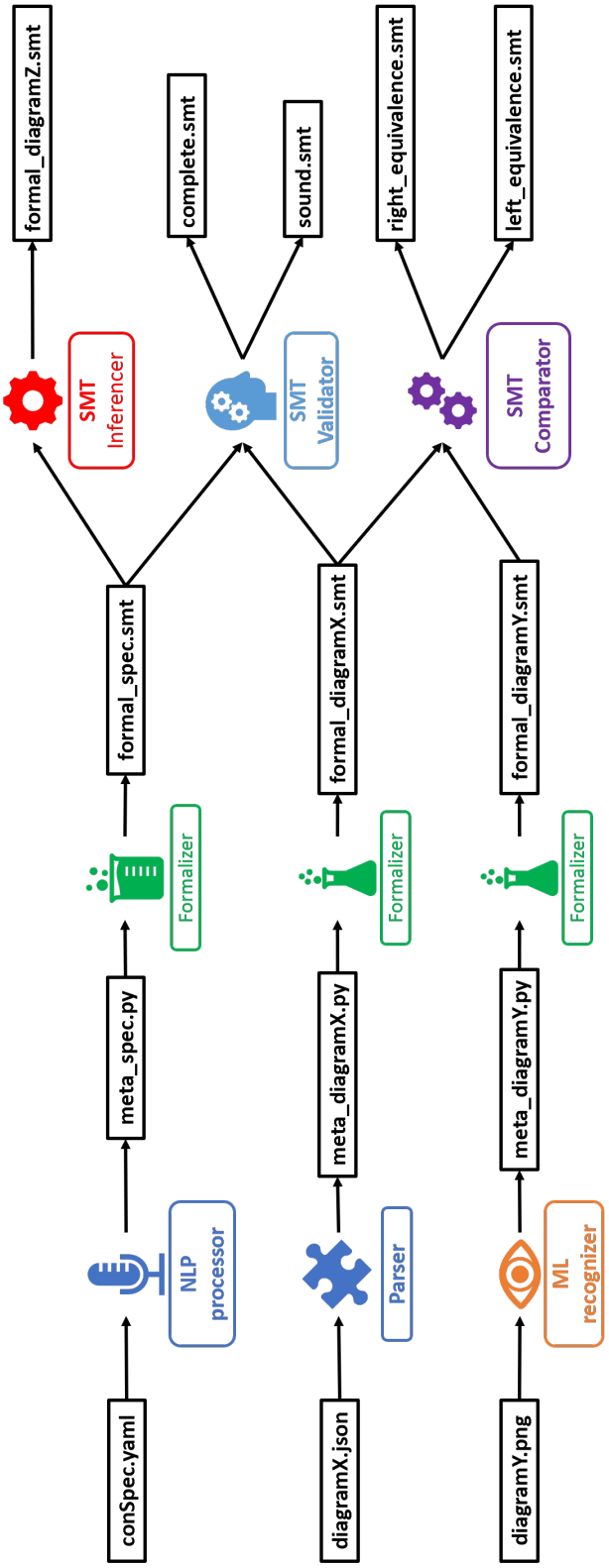


Figure 2.5: Schematics for T4TOMM

2.4 Summary

In this chapter, we presented an overview of our framework. We introduced how specifications and models interact with each other to achieve inference, validation and comparison. In addition to the core elements of our framework, we introduced the structure proposed to specify requirements, and a proof-of-concept that integrates this structure with our theories to create a complete workflow.

Every one of the elements introduced is referenced with the corresponding section or chapter that contains its details, so the reader can quickly identify the relation between all concepts in this dissertation.

Chapter 3

Context

The groundwork for *software specifications and models* is discussed throughout this chapter. Initially, we lay out the relevance of software as a subject of study by summarizing its applications. Subsequently, software engineering and requirements engineering are introduced as the disciplines to which contributions are made through this thesis. We also elaborate upon the fundamental concepts used in requirements specifications, software modelling and program verification.

In this chapter, we also present the state-of-the-art of the most relevant related work, which includes a discussion about existing specification languages and documents, and formal systems for model verification and comparison. We finally present the work we have done previously, which has also motivated the current research.

3.1 Background

In this section, we list the different concepts from software engineering and computer science that build the context for our work. This purpose of this section is not to exhaustively cover each concept, but to introduce the most general ideas and to provide the appropriate references should the reader require more information.

3.1.1 Software Applications

Our research is better appreciated by reflecting upon the evolution of software applications and their impact on modern society.

During the early 50s, software was mainly used by the scientific community to perform complex mathematical calculations[110]. Later, in 1966, the launch

of the DOS/360 operating system by IBM marked the beginning of an era for multi-purpose programs that could be executed from a command line in a straight-forward manner. The next big leap occurred during the 70s, with the popularization of Graphical User Interfaces (GUIs)[230, 259]. From interactive components to resource management, *Operating Systems* have played a fundamental role in the steady growth of the application of software.

However, Operating Systems did not uniquely contribute to the expansion of software usage. The Internet has, in recent years, also contributed substantially. In 1970, the ARPANET project was launched as the first global network of interconnected computers. The development of communication protocols was needed for it to work. Consequently, first applications, such as the case of the electronic mail and bulletin boards, were developed in the 1970s. The World Wide Web in the 80s and the first web browser in the 90s ushered a new direction with the emergence of web applications. Towards the beginning of the 2000s, the Web 2.0 introduced a social aspect, and more recently, the *Internet of Things* has become fundamental for its ubiquitousness.

The development of programming languages has eased the creation of software systems that address well-defined problems within specific domains. For instance, while COBOL[269] was mainly useful to encode communication procedures within banks, Ada was a multi-purpose language developed by the U.S. Department of Defence for military and commercial use[57]. Later examples include languages like C[261], which was preferred to build hardware controllers, and Java[18] which was praised for its cross-platform compatibility. More recently, the survey of programming languages conducted by TIOBE[58] has shown the increasing popularity of Python[263], which has to do with its usage in data science and machine learning[92]. Table 3.1 shows a long term history of the current ten most used programming languages.

Language	**2019**	**2014**	**2009**	**2004**	**1999**	**1994**	**1989**
Java	1	2	1	1	12	-	-
C	2	1	2	2	1	1	1
C++	3	4	3	3	2	2	3
Python	4	7	5	9	27	21	-
Visual Basic .NET	5	10	-	-	-	-	-
C#	6	5	6	7	23	-	-
JavaScript	7	8	8	8	17	-	-
PHP	8	6	4	5	-	-	-
SQL	9	-	-	6	-	-	-

Table 3.1: Long term history of the current 10 most used programming languages from the TIOBE index[58]

To this point, we have illustrated how these technologies have historically opened new horizons for the applications of software. However, our day to day activities themselves provide opportunities for software applications. These new opportunities very often demand existing technologies to be improved and new ones to be created in order to cope with our needs. This is evident when noticing the need to improve fraud detection in banks[12, 116, 239], or tumour recognition in digital imaging[87, 154, 201], or self-driving cars in the automotive industry[122, 250, 275]. These specific needs, in turn, push the development of new processors that deal with data and machine learning algorithms efficiently. This is the symbiotic relationship between technology and software applications that we introduced earlier.

The fact that software is present in almost any aspect of our life, including health, finance, business management, marketing, transport, home and fashion amongst others, makes it not only interesting but also an important subject of study. Thanks to high-level programming languages, software development is becoming more accessible, resulting in a large number of people developing applications. However, the increase in accessibility comes with its own set of risks and concerns. Because not every developer takes appropriate care[105, 175] of a software's safety, security, quality, correctness, testing, usability, and maintainability, amongst other aspects, and in fact, this is not a trivial task[117, 133, 192, 227]. For this reason, it is necessary to construct frameworks like TOMM, that support developers in the achievement of such tasks.

3.1.2 Critical Systems

Critical systems[119] are those whose possible failures would cause a considerable impact, especially concerning human lives. In consequence, they must be carefully design and developed, in such a way that the risk of failure can be virtually nullified. These systems are typically divided into safety critical[53, 177], mission critical[91], business critical[4, 88] or security critical[179, 300].

These systems are developed with heavy use of formal and mathematical models which allow to examine their properties rigorously. Examples of critical systems include aircraft controllers, dosage systems for medical treatments, infrastructure monitoring systems for nuclear reactors, etc.. The need for these systems has motivated a considerable amount of work regarding formal methods to verify systems and specifications. These concepts will be explored in this work in the context of non-critical systems.

3.1.3 Non-Critical Systems

We consider non-critical systems all those that do not imply a risk on human lives. In particular, we look at systems used on daily activities, such as mobile applications and web applications. These systems are commonly integrated by several components, namely user interfaces, communication protocols, web services, etc., and their development process prioritizes functionality and time to market over safety checks and advance verification. As part of this work, we investigated the application of techniques used in the development of critical systems into non-critical ones.

3.1.4 Software Engineering

Over time, processes followed to construct software have evolved together with its applications. In 1965, Dijkstra stated that "programming is the art of organising complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible"[102, p. 6]. His work, amongst others [101, 151, 301], was essential for the development of Software Engineering as a discipline; which has its origins in the 1968 NATO conference on the same subject [204, 232]. Since then, SE has been driving the constant and rapid development of new tools and methodologies that support the construction of software systems.

3.1.4.1 Software Development Processes

Software Engineering pays particular attention to the study of models that describe the process to build software applications. These models are better known as software development life cycles (SDLC), and define the activities to be performed, the artefacts to be generated at every stage, and the different roles to be played by all the people involved in the process, such as users, managers, sponsors, developers, testers, and all the stakeholders in general.

Sequential life cycles, such as the Waterfall[39, 40] model depicted, whose sequence of stages are shown in Figure 3.1, and the V-Model[120], define steps to be executed consecutively until the software system is completed. The stages of the waterfall model are usually used as a reference for other life cycles. Differences are present in the names of the stages, their scope, the way they are approached and the interaction amongst them. However, from a conceptual point of view, these key stages are present in any software development process.

Models like Waterfall fail to adapt to the complexity and changeability of software products in general, as it requires a more efficient development

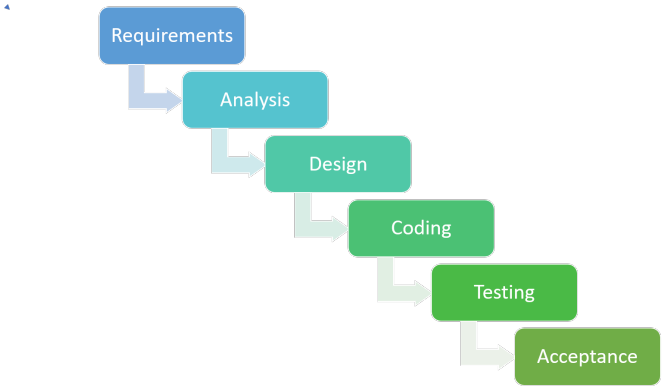


Figure 3.1: Waterfall Model

process[191]. Their inflexibility has motivated the development of incremental life cycles, such as Spiral and Prototyping[266]. These processes incorporate a faster build-fix-release iteration, in which small features are incrementally added to a software product, instead of expecting one "complete" system.

The evolution of applications described in Section 3.1.1 has triggered an exponential demand for software products; together with the need for more realistic and suitable development processes. Furthermore, the *Manifesto for Agile Software Development*[14] has set the grounds for a new generation of life cycle models: the *agile* methods[5, 6]. These methods are characterised for putting interaction between stakeholders in the centre of the process as a critical factor to efficiently develop functional and flexible software applications.

Understanding precisely how a software system is going to participate in the solution of a problem is always the first stage of any life cycle, whether it is agile or waterfall based. Poor understanding of the expected use of a system implies that errors are carried into every stage of the development process, resulting in a defective product. How to adequately define the features required from a software application is the subject of study of Requirements Engineering; which will be discussed in Section 3.1.5.

3.1.4.2 Object-Oriented Programming

Out of the seven most popular programming languages shown in Table 3.1, only C does not support Object-Oriented Programming (OOP), showing how popular this programming paradigm is. UML Diagrams are now a standard part of any Object-Oriented design process. The fact that OOP allows us to create software by drawing parallel to real-world entities by constructing models that mimic the entities' behaviour makes it a very useful to design

a system that is both comprehensive and comprehensible. It also improves the maintainability of the system. Therefore, it is no surprise that OOP has a dominant presence in all facets of programming, particularly concerning Software Engineering.

OOP had its origins with the Simula programming language in the 1960s. However, the concept was popularized by Smalltalk in the '70s and '80s. While there is not a general consensus what exactly constitutes OOP, a language can be considered to support it if it supports the following concepts: *Abstraction*, *Class*, *Encapsulation*, *Inheritance* and *Object* in terms of structuring the language and *Message Passing*, *Methods* and *Polymorphism* in terms of behaviour. Expanding on these individual concepts is beyond the scope of this research. A good overview of the history and on what constitutes OOP is presented in the work of Armstrong[17] and Kim[228, 234].

3.1.4.3 Software Testing

Software testing[15, 56, 228, 229] is performed across the development process to check for the presence or absence of errors in the programs being developed. Software testing is done by comparing the expected outputs from an ideal system, with the actual output produced by the real system. In this way, several problems are detected, including syntax errors, unexpected behaviours, and missing functionality.

Software testing techniques approaches are broadly divided into black-box testing and white-box testing. While black-box testing aims to check the program's functionality in relation to its specification, white-box testing checks the structure of the program in relation to its underlying code.

Software testing is closely related to software validation and verification, which are discussed in Section 3.1.4.4 and Section 3.1.4.5 respectively.

3.1.4.4 Software Validation

Software validation aims to check if the system being developed is the one expected by the user[281, 286]. Software testing (Section 3.1.4.3) and requirements validation(Section 3.1.5.4)[182, 200, 244] are some of the most common methods to conduct software validation[255, 281, 286]. Other methods are partial inspections and reviews through every stage of the development process[281].

Software validation is performed at the end of every iteration of the development cycle (see Section 3.1.4.1). However, intermediate validations help to track the progress of the development process and to identify problems regarding customers' expectations opportunely[255, 264].

3.1.4.5 Software Verification

Software verification aims to check if the system being developed complies with its specifications[10, 281, 286]. Techniques for software verification are usually classified into static and dynamic.

3.1.4.5.1 Static Verification

Static verification, also referred to as static analysis, checks the code of a program without executing the program itself[89]. Compilers typically perform static analysis for programming languages, which detect syntactic mistakes in the code.

In addition programs can be checked against formal specifications through programs for formal verification (discussed in Section 3.1.7.3). These specifications are captured in formal languages, such as Spec#[32] for C# code, JML[193] for Java code, and Eiffel contracts for the Eiffel programming language[218]. Formal specifications are further discussed in Section 3.1.7.1. This combination of formal specifications and program code are key to perform static verification.

3.1.4.5.2 Dynamic Verification

Unlike static verification, dynamic verification does require the execution of the program being verified. This is typically done through testing and experimentation. Because testing frameworks do not commonly rely on formal specifications, it is necessary to design suitable test cases that capture the requirements for the system being verified.

Dynamic verification is limited by the samples used in the sequence of experiments and tests, and it is rarely possible to conduct exhaustive testing for absolutely all possible scenarios. For example, it is not possible to exhaustively test a program to add two natural numbers, for it requires an infinite number of test cases. This problem is better addressed through static verification, in which formal theories for natural numbers are used. Though dynamic verification is more practical, it lacks the rigour achieved through static program verification[10].

3.1.5 Requirements Engineering

With the constant growth in the complexity of software systems[30], there is undoubtedly a need for a discipline such as software engineering. Even though the relationship between software development life cycles and software

engineering has been made clear in Section 3.1.4, we need to focus now on the role of Requirements Engineering. We examine the purpose of requirements engineering and its role in the successful construction of software applications.

The term requirements engineering has been around since 1979 [13]; but it was not until the 90s that it became a specific research area, thanks to the IEEE Computer Society [288]. The CHAOS Report [157] published by the Standish Group International in 1994 had a significant impact on the way system requirements are perceived. Ever since, requirements engineering has received increasing attention from the practitioners of software engineering.

The CHAOS report states that 13% of the IT executives surveyed attributed the success of a software project to the *clear statement of requirements*, which makes it the third most important factor. The report mentions *realistic expectations*, and *clear vision & objectives* as success factors with 8.2% and 2.9% respectively.

In contrast, the report shows factors that challenge projects, such as *incomplete requirements & specifications*, *changing requirements and specifications"*, and *unrealistic expectations* with 12.3%, 11.8%, and 5.9% each one of them.

Even though the validity of this report is arguably outdated [131], it has motivated a considerable amount of research in the area of requirements engineering, and it makes it clear that requirements-related factors have a high impact in the success, challenge or failure of a software project.

While software engineering studies how a system should be constructed to solve a specific problem, requirements engineering studies how to state that problem. The interaction of these two disciplines makes it possible to propose methodologies that support a more precise definition of *what* a software should do and *how* it should do it.

In Sections 3.1.5.1 to 3.1.5.4, we introduce the most significant work in requirements engineering that sets the grounds for our contribution to this discipline, before we detail them in Chapter 4.

3.1.5.1 Types of Requirements

Expectations for a software application are divided into two: those that are related to the problem addressed, and those that have to do with the qualitative attributes of the product itself. In the literature, these expectations are referred to as functional and non-functional[281] requirements respectively.

Functional requirements describe explicitly *what* the system should do, i.e. what business rules must be followed to solve the problem. Examples of these rules are: *the system has to record the name and age of the users*, and *users that are not administrators must not be able to remove entries.*

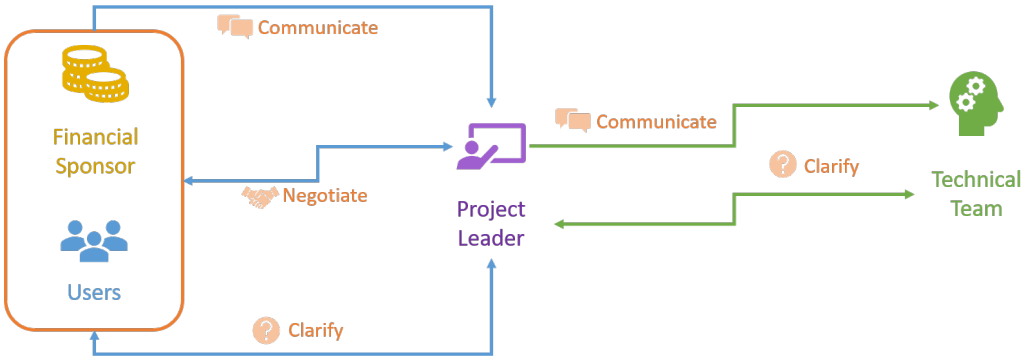


Figure 3.2: An example of a Requirements Communication Cycle

Non-functional requirements express characteristics of the system as a whole; that is, the capabilities and emergent properties it should observe. Examples of these properties are reliability, response time, memory usage, etc.

While non-functional requirements are well characterised[182] and are standard for any software system, functional requirements are rather specific to the domain of the system and to the problem it aims to solve. These qualities justify the need for further research targeting functional requirements, which is the case of our work.

3.1.5.2 Requirements Communication Cycle

Together with the elicitation techniques, it is necessary to establish an appropriate communication process amongst the stakeholders. We refer to this process as the *requirements communication cycle*, and its implementation may have subtle variations depending on the development methodology[248] being used and the roles defined in it. Nonetheless, it is possible to identify the most general elements of any communication cycle, as it is depicted in Figure 3.2.

In this example, the financial sponsor and the system users communicate the expected functionality to the project leader. This role is typically performed by the more senior members of the development team, such as software architects or product engineers, or by members of the administration team, including the product owner, team leaders, project managers, or SCRUM masters[1] where applicable. The project leader will review the requirements and will negotiate them with the sponsor and the users, just in case some requirements are not within the resource limits of the project (time, budget,

[1]The SCRUM master plays the role of the project manager in the SCRUM methodology[64]

knowledge, etc.). The project leader will also communicate the requirements to the technical team, typically formed of developers, testers, and quality assurance certifiers. The technical team can request clarifications from the users and sponsors via the project leader. Through the negotiations and clarifications, the requirements will be updated, until the final version is agreed upon. This refined version will be used to generate the software specification that will guide the current iteration of the development process.

3.1.5.3 Requirements Elicitation

Requirements elicitation is a common activity in all software development processes (Section 3.1.4.1). The goal of this activity is to precisely define the expected functionality of a software application in the form of requirements. This is achieved through the understanding of the problem to be addressed, and the discovery of the relevant business rules that will provide the solution.

When eliciting requirements, there is a series of tasks that should be performed, such as discovering, understanding, classifying, organising, prioritizing, negotiating and documenting requirements[281]. These activities are performed differently according to the experience of the developing team, the structure of the organisation, and the domain of the business. However, there is a series of commonly used techniques that allow us to succeed in the activities mentioned above.

The most utilised technique to collect requirements is *interviewing*. One way is to listen to stakeholders explaining the business processes in an open way, similar to brainstorming. Another way of doing it is to ask a set of well-defined questions to specific stakeholders. The second alternative is complementary to the first one in order to clarify doubts arising from the open session. Regardless of the format, or combination of formats used for the interviews, it is important to preserve a record of the output of this process, so that the information is available for future references.

Less common techniques[137, 288, 312] are *ethnography*, *stories and scenarios*, *introspection*, *focus groups*, *repertory grids*, *card sorting*, *protocol analysis*, etc. We do not discuss these techniques in detail, as we consider them as variations or extensions of interviewing; and their specific details do not have an impact on our work.

From the analysis of the existing techniques[182, 249, 281], and our experience in the software industry, we consider that semi-open interviews combined with observation, constitute the most suitable approach to elicit functional requirements.

In Semi-open interviews, a project leader meets with the sponsors and users (not necessarily at the same time) and discusses system functionality.

While their needs are being expressed, the leader must take notes and try to identify possible use-cases[77]. A use-case is, in essence, an atomic, well-defined, activity that occurs under specific circumstances. An example of a use case for a library system would be "The user of the library can borrow books and tapes".

Once all potential use-cases have been identified, their attributes must be clarified by including all constraints, and actors for each activity. Examples of these components are the constraint on the maximum number of books that can be borrowed, the dependency on a valid membership card, and the actors who do the action, either public in general or members of the library. Further interviewing and direct observation of the task at hand should lead the clarification of its attributes.

In some circumstances, business rules are already expressed through written documents, diagrams, or manuals. Hence, further review of existing resources is required to complement interviewing and observation in order to identify and detail use-cases. In addition to our preference of techniques, other alternatives from the ones mentioned earlier, or from any other personal experience, are applicable in the construction of complete specifications.

3.1.5.4 Requirements Validation

According to Sommerville[281], requirements validation is "the process of checking that requirements define the system that the customer really wants". Its goal is to identify problems with the requirements document that could have an impact on the development of the system. For instance, incomplete requirements derive into incomplete models, which at the same time produce incomplete software applications. This problem implies that the different artefacts (models, code, test cases, documentation, etc.) produced in every stage of the development process have to be reworked[182, 235, 281].

Different literature about requirements validation agrees on the aspects of requirements that have to be checked, such as consistency, completeness, feasibility, and testability [47, 182, 281]. Techniques proposed to validate requirements include reviews with stakeholders, cross-referencing, readings, prototyping, and use cases[47, 182, 244, 281]. More recent approaches attempt to integrate formal representations, such as ontologies, in the requirements validation process[98, 111, 148].

Requirements validation contributes to the process of software validation described in Section 3.1.4.4.

3.1.6 Software Modelling

Design and modelling is a relevant activity in most software development life cycles, even agile methodologies. The agile manifesto[14] establishes that *"Continuous attention to technical excellence and good design enhances agility"* and *"The best architectures, requirements, and designs emerge from self-organizing teams"* . From the Waterfall process depicted in Figure 3.1, it is observed that this activity comes after requirements have been defined and analysed, and before the program is built (or coded).

Modelling systems is the equivalent to drawing the blueprints for a house. That is, to define its components, as well as their shape, their scope and their interaction, based on the requirements and the best practices for software development.

Through this activity, high-quality modular programs are developed, which feature reusability, extensibility, readability amongst its qualities. Modelling also increases awareness on reliability, security and safety properties of a software system.

Models are abstractions that allow us to visualize different perspectives of the software to be built, including structural and behavioural elements. Different modelling techniques exist and are used for different purposes. For instance, mock-ups are preferred when designing front-end software components. One particular modelling approach we are interested in is the one proposed through the Unified Modelling Language (UML), which will be discussed next.

3.1.6.1 UML

The Unified Modelling Language (UML)[258] is a collection of diagrams proposed by the Object Management Group (OMG) as part of their Model Driven Architecture (MDA). The language has been widely adopted in industry and academia as the standard way to document, model and visualize the structure and behaviour of software systems together with the interaction between its components. The Figure 3.3 shows all the diagrams that are part of UML. In Section 3.1.6.2, we will discuss further Class Diagrams, which are the current object of our research.

UML has inspired the development of a number of Object-Oriented methodologies, including OMT[99], Objectory[162], the Booch method[49], and more recently the RUP[183] process. Together with these methodologies, a considerable number of diagramming tools have been developed, such as ArgoUML, PlantUML and RationalRapsody. Even popular Integrated Development Environments (IDEs) such as NetBeans, Eclipse and Visual Studio have

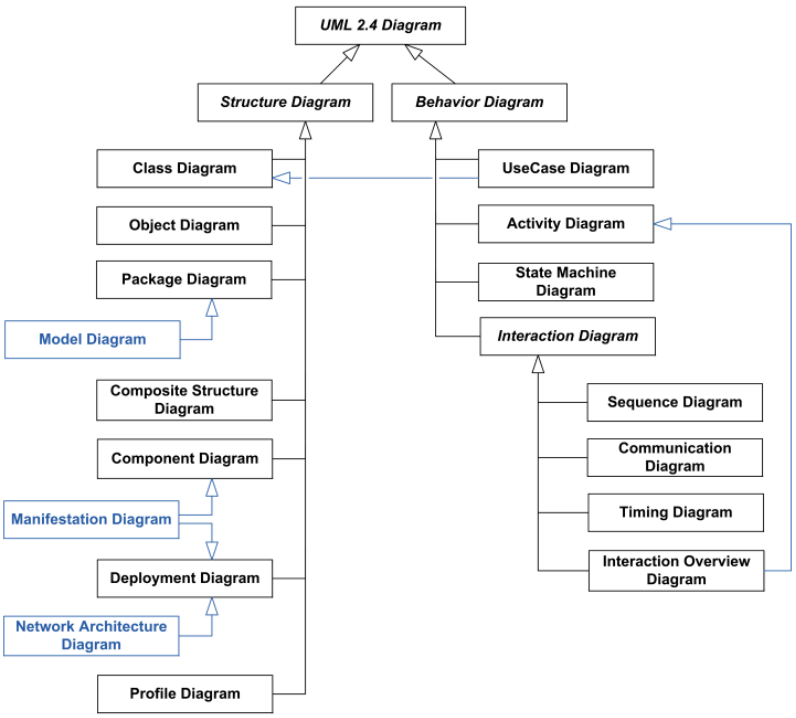


Figure 3.3: UML Diagrams

embraced the modelling language either natively or via extensions.

Despite its virtues, UML also carries some risks[38]. The *misuse* of the diagrams and the wrong interpretation of their intended purpose is one common problem amongst its users. Besides, there is the risk of the overuse, as the attempt to model every aspect of every system with UML.

While it is clear that UML works seamlessly with Object-Oriented methodologies and programming languages, it is not the case for applications based on web languages such as HTML and CSS, which are modelled better trough visualizations such as mockups[203, 262].

3.1.6.2 Class Diagrams

Class diagrams are the part of UML that describes the static structure of a software component in terms of its classes and the relations amongst them. A class describes an entity in terms of its attributes and operations. Attributes describe a set of named properties for the instances of a given class, such as the name and age of a person. Operations are used to indicate behavioural features of a class. For example, a calculator class performs addition and subtraction operations. Every operation is identified by its name, type,

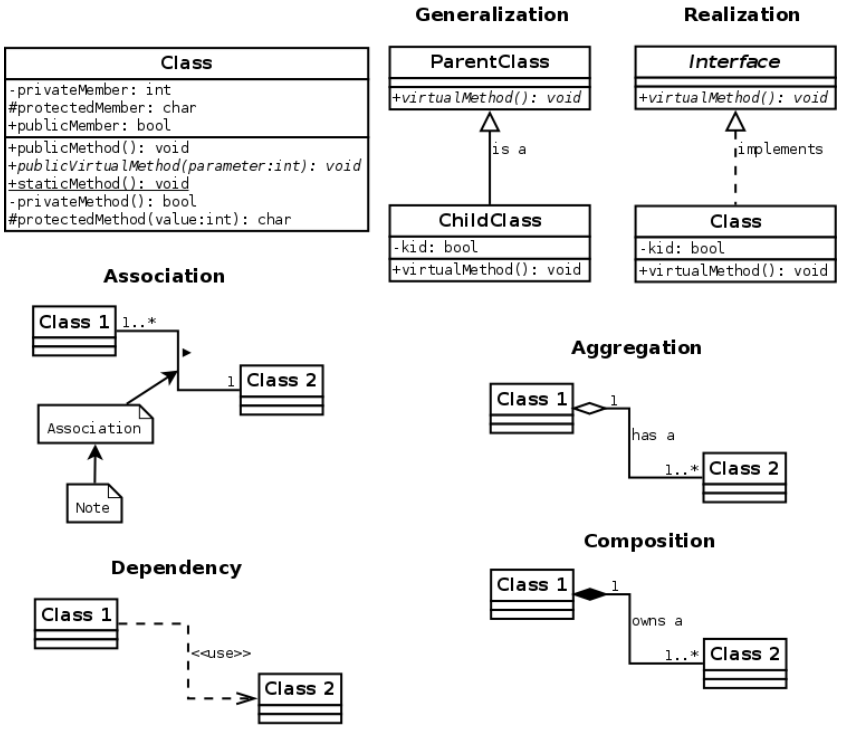


Figure 3.4: UML Class Diagrams Cheat Sheet

parameters, and the constraints for invocation.

The relations between classes are described through a set of well-defined association types. The most common examples of binary associations are aggregation and composition. The former is used to indicate that a property is part of a composite and that the property exists independently. The later, in contrast, indicates that the parts cease to exist when the composite does.

Figure 3.4 summarizes the main elements of a class diagram. A comprehensive reference for UML class diagrams, and UML in general, is available in the work of Fakhroutdinov[115] as well as the books of Rumbaugh[265], Pender[241] and Fowler[121].

Within the problem domain, classes model components of the business process, such as accounts, cards and customers in a baking system. Within the solution domain, classes model functional units with specific purposes within the system, such as data connectors, handlers, helpers, factories, interfaces, etc. Class diagrams are expected to contain elements of both domains. However, within this research, we focus on diagrams that capture the elements of the problem domain, for they come directly from the requirements specification. They also contain specific semantic interpretations as it will be described in Section 6.3.

3.1.6.3 OCL

The Object Constraint Language (OCL)[258, 278, 296, 297] is another standard of the OMG, which extends UML to specify constraints over classes, their attributes and operations formally. It is a declarative language with precise semantics to build well-formed formulae that express restrictions over the possible values of the elements of an object-oriented model. It is based on first-order predicate logic with a syntax similar to that of object-oriented programming languages.h

Constraints are expressed in the context of UML classes, but they are evaluated for the instances of the class. There are three main types of constraints. Invariants is a condition that must hold during the entire lifetime of an object. Preconditions are constraints that must hold before the execution of a given operation, while postconditions must hold at the end of its execution. These conditions allow us to capture parts of the behaviour of the operation itself within static structural diagrams. In Section 3.1.7.2.3, we discuss these constraints further within the context of Hoare logic and Design by Contract.

3.1.6.4 Model-Driven Development

This approach to software development puts models at the centre of the development process[19, 271, 281]. It has been pushed forwards thanks to the development of Computer-Aided Software Engineering (CASE) tools[215, 270], and the efforts of the OMG to develop modelling standards[258, 279].

3.1.7 Formal Methods

Formal methods are a collection of mathematical languages, theories and techniques to specify, analyse and verify software systems[73]. In this section, we introduce the elements of formal methods that are part of our work.

3.1.7.1 Formal Specifications

From the CHAOS report[157] discussed in Section 3.1.5, and the work of Jorgensen[165] and Fuchs[123], it is noticed that the success of a software project is closely related to the definition of precise and clearly understood requirements. The requirements document discussed in Section 3.2.1 discuss how requirements documents can be structured. However, problems inherited from human communication itself remain. That is the case with ambiguity, pragmatics, context and even common sense, which we have to deal with through proper specifications.

The main difference between requirements and specifications is that the former are usually express in an informal, natural and open way through natural language. The later, instead, are meant to be precise, unambiguous, and structured. This is achieved employing special notations, such as diagrams, mathematical equations, formal languages, or domain-specific constructs.

Another noticeable difference is that, while requirements tend to reflect the expectations of a system as a whole, specifications deal with particular elements of the system, such as architecture, functionality, component interaction, roles, etc.o In this way, requirements are understood as the superset of various specifications.

Throughout this work, we indistinctly use the terms requirements specification, software specification, functional specification, system specification or simply specification, to refer to the structured representation of the functional requirements.

We already mentioned some of the problems that proceed from communication in natural language. In order to deal with such problems, we surveyed specifications in the literature. The work of Kontonya[182] proposes *data-flow modelling*, *semantic data models*, *object-oriented approaches* and *formal methods*. In addition, Pressman[249] mentions *written documents*, *graphical models*, *formal mathematical models*, *usage scenarios*, and *prototypes*. Moreover, Sommerville[281] discusses *variations of natural language*, *graphical notations*, and *mathematical specifications*. The interesting overlap amongst the work of these authors and their relation with our research is further discussed in Chapter 4.

The choice of the format to be used depends on the type of software application to be developed. For instance, if we were to describe a software that affects human lives, *formal methods* would be a sensible choice, because it allows to *rigorously* guarantee that specific properties hold. In contrast, if we were to define a system mainly composed of visual elements, *prototypes* would be more suitable.

Rarely, systems are monolithic; instead, they are complex compositions of different subsystems. Hence, a complete software application may require as many specification formats as the number of components it contains. In this research though, we constrain ourselves to the specification of non-critical software applications as defined in Section 3.1.1, for they are responsible for the logical functionality of a system. Our specification format is presented in Section 4.1, and it constitutes an alternative for the formats discussed in Section 3.2.2.

3.1.7.2 Logics

Logics are formal languages with precise semantics and syntax. The later is defined thorough grammatical rules used to generate all well-formed formulae, within the language. A logic may additionally define an inference engine, which is a set of laws that allow the generation of new well-formed formulae from existing ones. A logic system integrated by a formal language and an inference engine allows to reason about statements within the system[86, 118, 282].

In a logic system, there is an initial set of well-defined formulae, called axioms, that define facts we know or assume to be true. Then, one can try to ask if another given formula is derived from the facts we know. If the application of the laws in the inference engine eventually leads to the formula in question, we have proven that such formula follows from the facts. This is just an overview of the way logic systems[303].

3.1.7.2.1 Popositional Logic

Propositional logic or propositional calculus is the most basic type of logic. In it, simple propositions of the type " Plato is a man" are expressed. Propositions are usually represented by letters from the alphabet, and they always must have either a *true* or *false* value[303]. In other words, a proposition is an assertion about some fact.

Propositions can be combined to form more complex structures through the following logical connectives: *not* ($\neg$), *and*($\wedge$), *or*($\vee$), *if...then*($\implies$), and *if and only if*($\iff$)[2]. If we assume that the literal P represents the proposition " Plato is a man", and the literal Q represents the proposition " Plato is mortal", we can write " Plato is human and mortal" as $P \wedge Q$, which is a well-formed formula within propositional calculus.

In order to reason about these propositions, a deductive apparatus or inference engine is required. One the apparatuses used with this logic is the *(Gentzen) natural deduction system*[129], which defines rules to introduce and eliminate connectives. For example, assuming we have two propositions A and B, it is possible to combine them in $(A \wedge B)$; and from A and $(A \implies B)$ we can infer B. This rule is usually written as shown below, and it is well-known as *modus ponens* or *implication elimination*[106, 112, 303]. With these rules, propositional logic becomes a powerful method to perform formal reasoning.

$$\frac{A \qquad (A \implies B)}{B}$$

[2]Refer to [303, p. 19] for further details about these connectives.

The type of reasoning shown above does not apply to all aspects of human reasoning. For instance, in the expression "I'll meet you after school", we need to understand the implications of the word "after". Sentences like "John's cat is big" may require an interpretation of the ownership of the cat, and sentences like "this place is nice" depends upon the understanding of which place we are referring to. Despite these limitations, propositional calculus has set the grounds for the development of more extensive logics.

3.1.7.2.2 Predicate Logic

First Order Logic (FOL) or predicate calculus[3] is one of the extension of propositional logic. In it, we model the world through the establishment of properties amongst elements. These properties are called predicates, and the elements can be either fixed or variable. For instance, to express the proposition "Plato is a man", we define the property of being a man as $\mathrm{MAN}(x)$, where x is a variable; then we apply that property to the specific subject Plato, resulting in the predicate $\mathrm{MAN}(Plato)$.

The ability to deal with properties also requires the ability to deal with subjects to whom the properties apply. In the previous example, we applied the property $\mathrm{MAN}(x)$ to one specific subject, but FOL also allows us to reason about groups of subjects. This is achieved through the *universal quantifier* ($\forall$), which states that a given property holds for all the elements in a given set, and the *existential quantifier* ($exists$), that indicates that a property holds for some (at least one) of the elements in a set. An example of quantification is $\forall x\, \mathrm{MAN}(x) \implies \mathrm{MORTAL}(x)$, which means that for any subject x, if x is a man, then x is also mortal.

In predicate logic, the connectives of propositional calculus are combined with predicates, variables and quantifiers in order to build more complex expressions. For instance, note that there is no "empty" quantifier, so if we wish to say that a property does not hold for any element, we can use the formulas $\neg\exists x\, \mathrm{P}(x)$ or $\forall x \neg\, \mathrm{P}(x)$. Predicates also allow us to express relations between subjects. We can express that Socrates is Plato' teacher thorough the formula $\mathrm{TEACHER}(Socrates, Plato)$.

We can also build more complex sentences such as "Mary has a little lamb", translated into the expression $\exists x \in People, \exists y \in Animals \mid \mathrm{IS\_NAMED}(x, Mary) \wedge \mathrm{IS\_LAMB}(y) \wedge \mathrm{IS\_LITTLE}(y) \wedge \mathrm{HAS\_A}(x, y)$. This last example illustrates the complexity required to express short sentences in FOL, and thus the reason why this format is not recommended to write requirements specifications.

[3] Refer to [303, p. 43] for more detailed information about FOL.

Reasoning in first-order logic is based on the natural deduction system of predicate calculus, with the inclusion of rules to introduce and eliminate quantifiers. The general idea of these rules is that if there is a property that holds for an arbitrary element, then it holds for all the elements. Whereas if that property holds for a specific subject (like Plato), then there exists at least one variable that satisfies that property, but we cannot say that it holds for all subjects. Inversely, if there is a property that holds for all variables, then it holds for any arbitrary subject, and if that property holds for any existing subject, then it must hold for a particular subject.

In spite of the complexity to express requirements specifications, we acknowledge the reasoning capabilities of predicate calculus, and we make use of them in modelling (Section 5.1), inference (Section 5.2), validation (Section 5.3), and comparison (Section 5.4) of class diagrams.

3.1.7.2.3 Hoare logic and Design by Contract

Hoare logic[150] was explicitly developed to prove the correctness of a computer program. The general idea is that given a valid set of conditions P, also called preconditions, the execution of a set of instruction S should always generate the expected output Q, also called postconditions. Hoare calculus states specific rules to indicate what should happen after the execution of single statements including skip, assignment, composition, conditions, consequences, and while loops. The syntax and semantics of this logic are very different from any of the previous ones since it reasons specifically about computer programs.

An example of reasoning within this logic is the following application of the assignment rule $\{x + 1 = 73\}\ y := x + 1\ \{y = 73\}$. In this example, we observe the execution of the assignment $y := x + 1$. Before this execution, we know that $x + 1$ is 73, and we assign that same value to the variable y. Therefore, after the execution, y must be 73. This calculus allows us to check every statement within a program until the end, and check whether the output obtained from the application of the rules is the same as the expected output.

Hoare calculus has an enormous influence on formal methods for software verification. Meyer's work on Design by Contract[216, 217, 219] is grounded in this logic. He introduced preconditions, postconditions and invariants into the Eifel programming language. Since then, variations of contract-based software verification have been developed for different programming languages, including C#[32], Java[37, 65, 79, 80, 173], and Python[238] amongst others. Given this influence and impact, both Hoare logic and design by contract set

the grounds for our work in Section 4.2.

3.1.7.2.4 Other logics

Though our work is firmly grounded on predicate calculus and Hoare logic, we briefly discuss other existing logics for the sake of completeness. **Modal logics**[303, p. 279], in contrast with propositional and predicate logics, do not assign an absolute true or false value to their statements; instead, the truth value can be qualified as a possibility, a necessity, or an impossibility. **Temporal logics**[205] is a type of modal logics that provides temporal operators, to reason about the past, present and future truth value of changing statements. For instance, we could say that a property will hold sometimes, or has always held. These logics have been used mainly to specify properties of concurrent systems[71, 243], and they are useful to capture reachability problems in paths[294].

Description logics[21, 22, 63] are commonly used in formal knowledge representation. They ease the description of concepts through hierarchies, and operators such as inclusion, equivalence, definition, negation, union, intersection, and universal and existential qualification (there are no conditional operators). These logics allows to check the properties of a concept based on the relations it has. For example, we could check if a penguin can fly based on its relations with other birds. Hence, they turn out to be useful information retrieval mechanisms similar to ontologies.

There is a considerable number of other logics out there, but we only discussed those that are tightly related to our work. Now we talk about some of the problems in logics such as decidability, and understandability.

3.1.7.2.5 Common Problems

To define the **decidability** problem, we have to remember the simplified operation of a logic system, which is to apply inference rules to existing formulae until we reach one expected formula. A logic system is *decidable* if there is any method at all to determine whether a given formula can be generated from the system. A logic can also be *semi-decidable* if there exists a method that accepts a given formula if and only if it belongs to the logic; otherwise the method either rejects the formula or does not halt.

Predicate calculus, in general, is undecidable[50, 132], it means that some formulae cannot be proven either true or false, and the same applies for Hoare logic [43, 184, 197]. Modal logics are decidable fragments of FOL[295], as it is the case for some temporal logics[55, 97, 184, 260] and description logics in

general[22, 153, 156].

Decidability can be seen as a formal or technical problem, whereas **understandability** is considered a more practical problem. It has to do with the ease of use of logics as a communication mechanism. We showed a few examples of statements in predicate and Hoare logics, and though trivial, they still require proper understanding of the syntax and semantics of these logics. The need for proper knowledge on these logics and the level of abstraction they demand are also the main limitations in their use for requirements specifications. It would be ridiculous to expect users, sponsors and other stakeholders to get an in-depth understanding of logics. Hence the need for a format that is more familiar to them.

3.1.7.3 Model Verification

Formal verification refers to the rigorous analysis of the behaviour of a software artefact expressed in a formal language. *Program verification* has received considerable attention from the research community since the establishment of its principles in Hoare logic[150] (discussed in Section 3.1.7.2.3).

Since a program is usually built based on a model, the former can only be as correct as of the later. For this reason, *model verification* is equally important to construct correct programs. According to Balaban[29], correctness is "the capability of a model to denote a finite but not empty reality". In his work, he accounts for two components of correctness; *consistency* which checks for non-emptiness and *finite satisfiability*, which checks for termination.

In addition, Cabot's work[60] deals with strong and weak satisfiability, as well as redundant constraints. They use Constraint Logic Programming to formalize class diagrams and the solver ECLPS to prove these properties. The relationship between the work of Cabot and Balaban is extensive, and it serves to exemplify the efforts done in model verification.

3.1.7.4 SAT/SMT Solvers

A SAT solver is a program that evaluates interpretations of boolean formulae in order to find a solution that can satisfy another given formula. Though the satisfiability problem belongs to the class of NP-Complete problems, these tools are still powerful enough to perform some program verifications[16].

The power of SAT solvers can be improved with the use of Satisfiability Modulo Theories (SMT), which extend the application of solvers beyond just boolean variables. With SMTs, we can evaluate problems including first-order logic, arithmetic, and other domain-specific theories[95]. Amongst the most popular SMT solver we find Z3[96], Yices[107], Simplify[100], and CVC4[35].

For our current implementation, we have chosen CVC4 as discussed in Section 6.1.2.2.

3.1.8 Data Augmentation for Machine Learning

Our current research is not intended to be exhaustive regarding Machine Learning (ML); however, it is important to highlight the potential use of our framework within it. In this section, we briefly introduce the concepts of ML that define the potential application context for our equivalence theory described in Section 5.4.

3.1.8.1 Machine Learning

Machine Learning[45, 226, 302] is a collection of algorithms used to generate predictive models. There are roughly three types of machine learning techniques, namely supervised, unsupervised, and reinforcement.

Supervised learning depends on labelled training data, i.e. input combination of values mapped to expected outputs, in order to synthesize a function that can reproduce the same mapping for new values. Supervised learning includes Decision Trees[45, 198], Random Forest[45, 196], KNN[45, 310] and Logistic Regression[45, 68].

In contrast, unsupervised learning[31, 76] examines a large number of values in order to find patterns within them; this task is usually referred to as clustering. Clustering algorithms include hierarchical clustering[93], k-Means[163], gaussian mixture models[210], self-organizing maps[180] and hidden Markov models[109, 254].

Reinforcement learning[168] is used to train models for decision-making based on rewards and punishments. This type of learning is characterised by the interaction of agents with an external environment through actions and states. Some of the most common reinforcement learning algorithms include Q-learning[240] and SARSA[285].

3.1.8.2 Neural Networks

Artificial Neural Networks (ANN)[144, 145] are computational models based on the connection of small computational units (neurons) in order to build complex mathematical functions. ANNs are composed of neurons grouped in layers and connected through weighted connections that propagate information through the network. A learning rule enables to adjust the weights of the connections through iterations of the learning process until the expected output of the network is achieved.

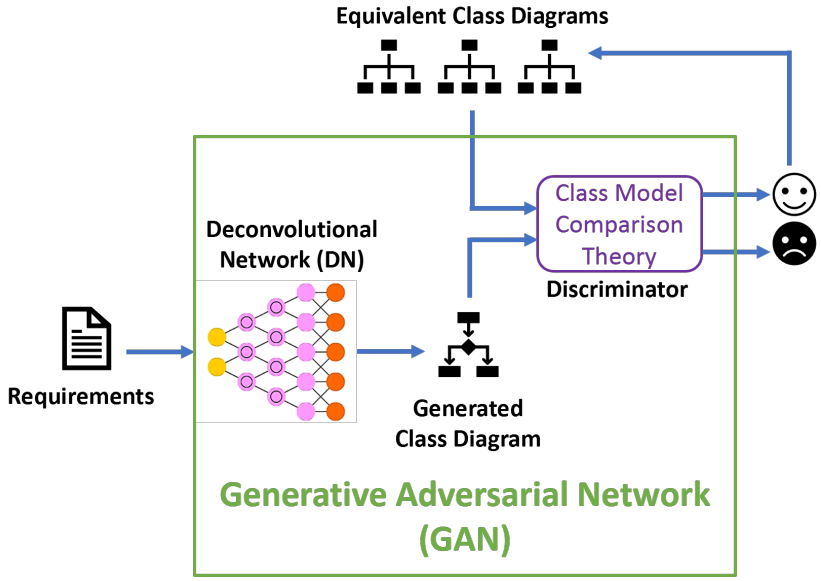


Figure 3.5: Usage of our Comparison Theory to within a CNN that generates class models from requirements documents

Neural Network are widely used in Machine Leaning[46, 145, 155, 298], and have motived new learning strategies[194, 233] including generative models[160, 172]. Generative models have demonstrated their capabilities for image generation and recognition[256, 257, 308].

We are particularly interested in Generative Adversarial Networks[138] (GANs), in which a generative model competes against a discriminator in order to improve the outputs of the model. We believe that our equivalence theory proposed in Section 5.4 can be used as a discriminator in a GAN as shown in Figure 3.5. In this way we could generate *additional* class models from a requirements document, assuming we already have at least one valid model for the same requirements.

3.1.8.3 Data Augmentation

If we have a requirements document, and we want to classify its contents (sections, words, sentences) based on the elements of class diagrams (classes, attributes, operations), we can resort to machine learning. However, it requires a considerable collection of requirements labelled with their corresponding class diagram tags. Up until now, we have not been able to find any such dataset, the closest thing we have found is a collection of requirements documents with some class diagrams attached[289], but even for this set, the number of diagrams available is limited.

In Section 3.1.8.2 we explained how GANs and our comparison theory could be combined to generate additional class diagrams for a given set of requirements, this particular activity is known as data augmentation[103, 293].

3.2 Related Work

3.2.1 Requirements Specification Document

Regardless of the technique used, any resources acquired and/or generated through the elicitation process must be collected for future reference. Recordings of the interviews, written questionnaires and responses, digital images, and notes are some of the formats that can be used for this purpose. These resources are used to derive the detailed requirements document.

Kontonya[182] and Sommerville[281] discuss the importance of the requirements document to guide the development process and to communicate expectations within the stakeholders. Prototypes of this document are found in the literature [1, 36, 146, 159]. Based on our experience in the software industry and our previous research[212–214], we propose our own requirements specification document in Section 4.2.

Our work expands some of the elements of the software requirements specification proposed by the IEEE in the ISO/IEC/IEEE 29148 standard[159]. This standard integrates all relevant components of functional and non-functional requirements into 19 well-defined sections. This research has an impact in 3 of those sections, namely *Specific requirements*, *Functions*, *Verification*. Our contributions to the first and second sections are described in Chapter 4, whereas our work regarding the third section is detailed in Section 5.3.

3.2.2 Formal Specification Languages

3.2.2.0.1 Specification Pattern Systems

They were introduced by Dwyer[108] to specify common behaviours in temporal logic through formal patterns. However, these patterns turn out to be too abstract to be understood by some stakeholders, such as users and customers, or even by software developers, without previous training in such logic. Even though they are well-defined patterns, they are not designed to deal with functional requirements explicitly.

3.2.2.0.2 Ontologies

They were developed to represent knowledge through well defined relations between concepts. Though they were conceived within the area of artificial intelligence, they have proven to be equally crucial in RE, because they can formally describe real-world concepts. Examples of its use in RE are present in the literature[54, 167, 195, 242, 277, 307].

The manipulation of ontologies imposes a challenge for their users; this is because the complexity of its graphical representation, as a network of connected concepts, may be intractable for problems with numerous connections. Additionally, their semantics is not implicitly known by all stakeholders. For instance, it is challenging for users and customers to understand the notations and structures used, whereas software engineers and architects are typically more familiarised given its similarity with other technical notations.

3.2.3 Controlled Natural Languages

A Controlled Natural Language (CNL) is defined by a finite set of grammatical rules that resemble the constructions of natural language with certain restrictions over the vocabulary, the syntax or the semantics used.

In 3.1.5.2, we talked about the communication process followed to elicit requirements. Throughout this process, the type of information exchanged may differ with respect to the interlocutors, thereby requiring different communications formats[272, p. 379]. For instance, the language spoken between computers is different from the one spoken between humans, and the language spoken in Engineering is not the same as the one spoken in Medicine. Despite the considerable number of existing CNLs[185], we investigate only those that satisfy the following relevant features: they must be based on the English language, they must be computable, and they must allow to communicate software requirements. We present a review of such CNLs next.

3.2.3.1 SBVR

SBVR Structured English[236] is a language developed within the context of model driven development[271] that defines *terms*, *names*, *verbs* and *keywords* as its building blocks. It is characterised by its use of visual elements, such as colouring and underlining, to identify components. From the official overview of the approach, we know that "SBVR has a sound theoretical foundation..., the base is first-order predicate logic with some limited extensions into modal logic"[237].

This CNL has been used as an intermediate language between English and formal languages such as Alloy[161] and OCL[258]. The main difference

between our approach presented in Section 4.1 and SBVR is that we only make use of grammatical constructs without graphical elements. Also, the number of expressions supported by our CNL, though sufficient for capturing functional requirements, is smaller than the ones allowed by SBVR.

3.2.3.2 ACE and FE

Attempto Controlled English (ACE)[123] is accompanied by an automatic and unambiguous translation into first order logic. It supports features, such as complex noun phrases, anaphoric references, subordinate clauses, modality and questions. *Formalized-English (FE)*[209] is based on Conceptual Graphs[223] and prioritizes expressiveness. Besides quantification and negation, it supports contexts, possibility, collections, intervals, and sentences that can be reduced to FOL. The problem with these two CNLs is that while they are more expressive and more comfortable to formalise, their syntactic rules are more numerous and complex.

Given this problem, we argue that the usefulness of a CNL to express requirements specifications is inversely proportional to its complexity. The work of Williams[299], Funk[124], and Garcia[127] has been considered to set the grounds for our argument. They demonstrated how the use of complex CNLs[247] is not trivial for experts and non-experts.

From here, the current challenge we address is to develop a CNL that minimises complexity but maximises expressiveness in the definition of software specifications. That is, a language that can be understood by all stakeholders without any additional training, and quickly learnt by the stakeholders responsible for documenting the requirements. With these considerations, the use of CNLs to write specifications is justified.

In this section, we have discussed different alternatives to consider when building requirements specifications. From the options listed, we have highlighted the challenge we address with the development of our CNL, which is presented in the following chapter. Within the alternatives, we have also discussed logics, and though they are not our first choice for specifications, they remain relevant for our contributions in sections 5.2 and 5.3.

3.2.4 Model Checking

Model checking is the application of formal verification to prove the absence or presence of well-defined properties of a system. The most common case is to check safety and liveness of concurrent systems[190]. The former assures that nothing bad happens, and the second one implies that something good eventually happens. Good and bad events are checked in terms of interaction

and access to resources, being deadlocks and starvation two typical aspects to be checked for[61]. Model checking makes use of temporal logics, discussed in Section 3.1.7.2, to reason about the reachability of specific states of the system that represent the properties just mentioned.

In this way, model checking aims to verify whether the model satisfies the safety specifications of a system, which is to some extent similar to the problem we address in Section 5.3. However, it differs in that model checking requires a formal specification of safety properties, while our work on model validation needs requirements to be specified in a reduced and properly defined set of sentences in English, which we present in Chapter 4. Another difference is that model checking evaluates the states of the model, and our approach evaluates the relation between a class diagram and the requirements it captures. Another difference is that the area of application for model checking is mostly hardware design, while our approach targets the modelling of non-critical software, described in Section 3.1.3.

From the literature[29, 59, 134, 278], we identified the basic properties to be internally verified in a class model. They are:

- **Strong satisfiability:** all the elements of the diagram are instantiated.
- **Weak satisfiability:** at least one class is instantiated.
- **Liveliness of a class:** one specific class must be instantiated.
- **Lack of constraints subsumption:** there are no redundant constraints

In particular, there has been a considerable effort in developing formal verification of UML Class Diagrams. Cabot et al.[59] use constraint programming to verify weak and strong satisfiability or absence of constraint redundancies. Gorgolla et al. [134] have developed a tool for the validation of UML models and OCL constraints based on animation and certification. The work of Soeken[278] illustrates how to encode a class diagram into a SAT problem, where OCL constraints represent states to be checked for. Miao[221] proposes a formalisation of UML class, sequence and statechart diagrams in Z3; although no particular properties are checked. Clavel and Egea [75] present "a rewriting-based tool that supports automatic validation of UML class diagrams with respect to OCL constraints". This short survey samples the kind of work done regarding UML, and it is interesting that most of them have to do with OCL and the states and inner properties of the diagram.

We ought to notice that it is difficult to find work related to static and dynamic verification of models; these activities seem to be more closely related

to computer programs and not to their models. Though formal verification has a more reasonable impact in model verification, it is limited to the internal properties of the model, leaving aside the relationship between models and their respective specifications; this brings up the question about *how can we know that a given model is correct with respect to its specification?*. Model validation aims to answer this question, and we discuss it on Section 5.3 in order to answer this question.

3.2.5 Model Validation

Software validation answers whether the right application is being built, or whether it satisfies its intended use.

Both requirements and software validation have been studied before; however, model validation is usually given up in favour of model checking.

In this work, it is argued that model validation is relevant for the construction of realisable software systems, and as such it requires a proper definition. Analogically to requirements validation, model validation is defined as *the process of checking that the model captures precisely what the specification states.*

Requirements validation is usually done with the assistance of the users of the system, and software validation requires test cases. Model validation, in contrast, is achieved utilising formal methods, that allow to abstract requirements and models, and to formally establish the relation between them. In order to tell whether the model satisfies the requirements or not, our proof-of-concept makes use of formal verification tools, such as SMT solvers, and formal systems such as first-order logics, which have been introduced in Section 3.1.7.4 and Section 3.1.7.2 respectively.

In Section 5.3 the components of model validation are discussed, including the inference rules that enable to prove this property of models.

3.2.6 Model Generation

Abdouly et al.[2] have provided us with a survey of tools that generate UML diagrams from requirements. We have used this survey to provide our classification of these tools based on the specific problems we addressed, as defined in Section 1.1. The categories we proposed are manual generation, rule-bases generation and machine learning generation tools; each of them is now discussed.

Manual generation refers to the approach in which a person, typically a software engineer, extracts information from the requirements of a system in order to generate models. The most common manual approach is inspection[9,

114], in which inspectors intervene in the stages of the development process to analyse the artefacts of the system. The most significant limitation of this approach is the possibility of human error, which motivated the development of automated techniques.

Rule-based generation approaches includes systems based on natural language processing[51, 123, 147, 186, 231, 277]. They make use of mapping between language and model structures based on well-defined rules. Some of these approaches, however, still required some manual work in order to deal with the complexity of natural language.

From the work of Abdouly, we noticed a lack of tools that use machine learning; this has been related to the nature of the data involved. Representing both documents and images within learning algorithms is a challenge being addressed in areas where more data is available, such as tumour detection systems, and multi-language translation systems. The lack of data available for requirements-to-model translation has set a significant obstacle in the development of these approaches.

3.2.7 Model Comparison

From the literature review, we found that most of the existing work aims to compare and relate different kinds of diagrams. For example, Siau[276] compares the application of use case with class diagrams, while De Lucia [94] compares Entity-Relationship diagrams with class diagrams. Another approach based on OCL was proposed by Gorgolla[135, 136], which compares constraints over the diagrams, rather than their contents. Berardi[41] and Queralt[252] propose formalization methods for class model reasoning based on detection of inconsistencies and redundancies. Our approach, however, uses a different approach, based on semantics and contents of the model, and not in their constraints.

3.2.8 Model Extraction

Model extraction is the task of processing images representing class diagrams, in order to generate equivalent models using some descriptive notation. Related work includes that of Ho[251], who used image processing and machine learning to classify class diagrams, based on graphical features such as lines, shapes, etc. In a similar way, Futrelle[126] presents an effort to extract and classify diagrams from PDF documents, and Dragan[104] aims to identify class stereotypes. To the extent of our knowledge, and based on our research, there are no systems that aim to explicitly extract the components of a

class diagram using image processing. However, since this is a secondary contribution we do not provide an exhaustive review of this topic.

3.3 Previous Work by the authors

3.3.1 Construct by Contract

Our interest in the development of reliable systems was initially expressed through Construct by Contract (CbC)[212]. It is an ideal development process in which contracts from Meyer's Design by Contract[217] are extended across the entire software development life cycle. CbC expected requirements to be expressed in terms of specification contracts that could be transformed into model contracts, and program contracts to aid model and program verification.

This concept inspired the development of the requirements specifications presented in Chapter 4. Though CbC motivated the current work, our new contributions aim to tackle real scenarios, far from the idealized ones covered by our previous work.

3.3.2 TOTOTL

Tototl[213] is also part of a previous effort to contribute to the development of reliable systems by providing a toolkit for Construct by Contract. It is a tool that, like the one proposed in Chapter 6, provides an interface to interact with formal methods easily. In concrete, Tototl analyses constraints expressed in a business-rules language named SBVR[27] and generates corresponding OCL translations to annotate class models.

This translation is done by means of reduced sentences in English with a basic structure composed by an actor, and action and a receiver. This type of sentences inspired the work presented in Chapter 4. Figure 3.6 shows the implementation of Tototl to support these sentences and their representation throughout the different stages of the development process.

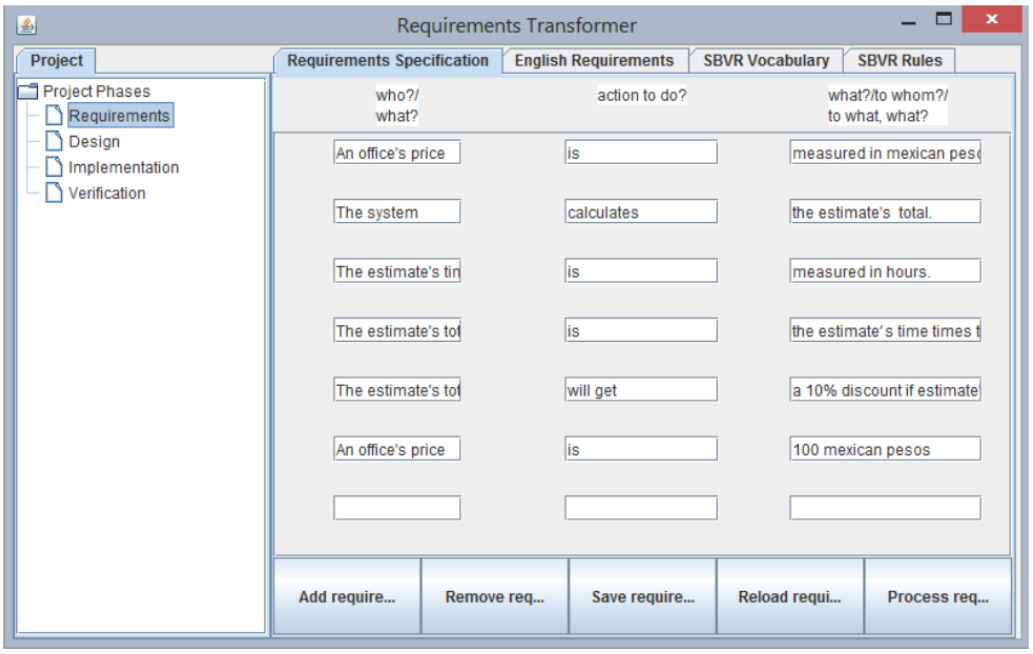


Figure 3.6: Tototl Implementation

This work was presented in the Conference on Computing Natural Reasoning (CoCoNat) 2015, at Indiana University, Bloomington. It motivated us to research further about formal specifications derived from natural language.

3.4 Summary

This chapter introduced our subjects of study, which are non-critical software applications, and requirements specifications. For the former, we talked about the diversification and classification of software applications, which allowed us to define the scope of our research. For the later, we discussed the impact of Software Engineering and Requirements Engineering, and the relation of our research with the Requirements Engineering Process. We surveyed the state-of-the-art for requirements elicitation and specification, and we detailed different specification formats, including SPS, ontologies, logics, and CNLs. We concluded this chapter with some foundations for software verification.

Chapter 4

Requirements Specification

Understanding the problem to be solved constitutes the first step towards the construction of a successful software system. History, literature, and experience have demonstrated that *elicitation*, *specification*, and *validation* of requirements are crucial tasks to ensure the success of the solution built[249, 280]. In this chapter, we concentrate our efforts in describing our contributions to the *specification* task, which serves as the foundation to the rest of our work.

In the following pages, we describe a set of rules to specify requirements systematically, along with an original document structure inspired by legal contracts. These rules and format are further elaborated through an example that describes the process to document functional requirements into a more standard specification that should have the following properties[249, 280].

- **Structured**: The document that captures the specification must be separated in sections that capture similar or related specification elements.
- **Clear**: All the specification elements must be understandable for all stakeholders.
- **Concise**: Redundancy between specification elements must be avoided.
- **Correct**: Specification elements must not contradict each other.
- **Unambiguous**: Every specification element is uniquely defined and referenced.
- **Verifiable**: Specification elements can be checked for these properties.

By specification elements, we refer to all the constructs that integrate the specification, such as sentences, formulae, equations, diagrams, definitions, etc.

In the Section 4.1 of this chapter, we discuss how Controlled Natural Languages (CNLs), introduced in Section 3.2.3, can be used to specify software requirements. We also propose a CNL that supports the properties defined above. In Section 4.2, we present the structure of a requirements specification document inspired by legal contracts to specify non-critical systems. Also, in Section 4.3, we present the transformation process required to derive this specification document from the needs of the user using our proposed CNL.

Controlled Natural Languages were introduced in Section 3.2.3 as languages that resemble the structure of natural language, though with limited grammar and/or lexicon. The difficulties that these languages present[124, 127, 299], such as complexity and domain restriction, were discussed together with the alternatives to overcome them[140].

As part of the solution to address these difficulties, it is proposed that only a specific set of people within a project must learn the exact rules of the CNL used. The responsibility of the *project leader* to communicate the requirements to the development team, as depicted in Figure 3.2, makes it the most suitable role to perform this task. Project leaders, unlike users and sponsors, can make use of the knowledge about the CNL rules for the different projects they are involved. In this way, the other stakeholders do not need to memorise these rules, yet they are capable of understating the specification document written.

In summary, we optimise the collective effort of the stakeholders when specifying software systems using CNLs by minimising the number of people that need to learn their rules, and maximising the number of stakeholders that can understand the requirements captured. We have discussed some general purpose CNLs[123, 185, 209, 236] CNLs in Section 3.2.3, now we propose a Controlled Natural Language that targets specifications for non-critical software systems.

4.1 SpeCNL

In this section, we present SpeCNL (read as spec-n-l), our CNL designed to address the specification of functional requirements for non-critical software applications. We define all the elements that represent the building blocks of our language, including essential parts of speech, some basic concepts and more complex blocks that are specific-purpose sentences. Note that we repeatedly use Backus–Naur form (BNF)[178] throughout this section to

define the grammars for SpeCNL, we do this using the syntax for NLTK[199] grammars.

4.1.1 Parts of Speech

Parts of Speech (POS) are categories used in languages and grammars to associate a specific behaviour to a word within a sentence, based on its grammatical function. For example, in the sentence "*give me a* can *of tuna*", the word *can* behaves as a *noun*, whereas in the sentence "*the kid* can *read*", the same word behaves as a *modal verb*.

Within computer science, POS are used in several Natural Language Processing (NLP) tasks[166], including automatic translation, text comprehension, speech-to-text and text-to-speech conversions, etc. The set of categories used as Parts of Speech may vary depending on the purpose of the application that uses them. The most common set is the Penn Treebank[207] that contains 48 categories, and is used in popular libraries such as the Stanford POS tagger[291, 292].

Even though the Penn Treebank tagset captures a significant number of categories, some of them are not needed to write software specifications, such as adverbs and interjections. Likewise, perfect tenses can be avoided by providing special semantics for other tenses that simplify expressing the needs of users. We propose the adaptation shown in Grammar 4.1 to be used within our CNL. Notice that these rules do differentiate between plural and singular nouns, however, in our theories described in chapter 5 we assume that the relations of singular and plural nouns are known. In Chapter 6 we discuss the usage of natural language processing to generate these relations automatically.

```
NON -> any noun either singular or plural
NOS -> a singular noun
NOP -> a plural noun
VBB -> any verb in infinitive form
VB -> any verb in simple present conjugation
VBP -> any verb in simple past form
VBI -> any verb in participle form
ADJ -> an adjective
NUM -> any real number
INT -> any integer number
DEC -> any decimal number
DIG -> any digit
LAMBDA -> the empty string
```

Grammar 4.1: POS in SpeCNL

4.1.2 Concepts

Like Parts of Speech representing the category of a *word*, we need to categorise more complex structures. We refer to the set of these categories as *concepts*. These behavioural elements capture functions that are reused throughout all the different sentences that form our Controlled Natural Language.

Modals

They indicate the level of obligation that must be observed by other concepts. For our approach, only two levels of obligations are needed. *Must* is the highest level, meaning the constraint has to hold at all times. *Can* on the other hand, represents the minimal level of obligation, implying the constraint does not have to hold at all times. Modals are formally described in Grammar 4.2.

```
MODAL -> 'can' | 'must'
```

Grammar 4.2: Modals in SpeCNL

Comparators

These are elements that allow the comparison of two values of an attribute. For instance, "*A is* more *flexible than B*". We identify two general types of comparators: *equality* and *inequality*. Within our CNL they are expressed in Grammar 4.3:

```
COMPARATOR -> INEQUALITY 'or' EQUALITY | INEQUALITY | EQUALITY |
↪ COMPARATOR_SYMBOL
INEQUALITY -> 'greater' 'than' | 'less' 'than'
EQUALITY -> 'equal to'
COMPARATOR_SYMBOL -> '>' | '<' | '=' | '<=' | '>='
```

Grammar 4.3: Comparators in SpeCNL

Entities

An *Entity* has two main functions: it can execute an action, or be affected by an action. Entities are simple nouns or quantified nouns as described in Grammar 4.4.

```
ENTITY -> PLURAL_ENTITY | SINGULAR_ENTITY | QUALIFIED_ENTITY | NON |
↪  ATTRIBUTE | NON
SINGULAR_ENTITY -> SINGULAR_INDICATOR QUALIFIER NOS
PLURAL_ENTITY -> PLURAL_INDICATOR QUALIFIER NOP
QUALIFIED_ENTITY -> QUALIFIER NON
SINGULAR_INDICATOR -> 'a' | 'an' | 'one' | '1' | 'the'
PLURAL_INDICATOR -> INT | 'the'
QUALIFIER -> ADJ | NN | VBI | QUALIFIER'-'QUALIFIER | LAMBDA
```

Grammar 4.4: Entities in SpeCNL

Attributes

An *Attribute* is an element or property of an entity. They always have to refer to the entity they belong to in order to be valid. They cannot exist on their own, and they are described in Grammar 4.5.

```
ATTRIBUTE -> ENTITY APOS NON | NEUTRAL_INDICATOR NON 'of' 'the' NON
```

Grammar 4.5: Attributes in SpeCNL

Actions

An *Action* is the activity performed by an entity or over an entity. It contains a verb in the simple present tense and its level of obligation is assigned through a modal, as seen in Grammar 4.6.

```
ACTION -> VB | MODAL VB
```

Grammar 4.6: Actions in SpeCNL

4.1.3 Sentences

We have already presented the common "atomic" concepts of our Controlled Natural Language. In this section, we present a list of structures, named *sentences*, that are to be used to build the different elements of the specification. These sentences have well-defined purposes, and a collection of them is used to generate the requirements specification document.

Structural Sentences

These are used to capture the structure of the concepts that define the problem domain. Because these sentences express properties that apply to all elements of the domain, they must be written using the pluralisation of a noun. Some

examples of these sentences are: *books must have an author*, or *books can have an ISBN*. Its formal description corresponds to Grammar 4.7:

```
    STRUCTURAL_SENTENCE -> NON MODAL 'have' STRUCTURAL_ITEM
    STRUCTURAL_ITEM -> ENTITY | ENTITY',' STRUCTURAL_ITEM | ENTITY', and
↪   'STRUCTURAL_ITEM
```

Grammar 4.7: Structural Sentences in SpeCNL

Comparison Sentences

They are used to compare the values of attributes. For example *the age must be greater than 5*, and *the user's age must be greater than 5*. The BNF describing these sentences is Grammar 4.8:

```
    COMPARISON_SENTENCE -> ATTRIBUTE CONSTRAINT COMPARATOR NUM |
    ↪  ATTRIBUTE CONSTRAINT ADJ
    CONSTRAINT -> OBLIGATION | POSSIBILITY
    OBLIGATION -> 'must' 'be'
    POSSIBILITY -> 'can' 'be'
```

Grammar 4.8: Comparison Sentences in SpeCNL

Cardinality Sentences

They are used to represent limits on the size of collections of elements. For example: *users can borrow up to 5 books* and *users must return at least 1 book*. They are structured as described in Grammar 4.9

```
    CARDINALITY_SENTENCE -> ENTITY ACTION LIMIT NUM NON
    LIMIT -> 'up' 'to' | 'at' 'least' | 'maximum' | 'minimum' |
↪   'exactly'
```

Grammar 4.9: Cardinality Sentences in SpeCNL

Conditional Sentences

These sentences are used to express the actions to be taken in specific cases. A *case* expresses the conditions that trigger an action, and the *consequence* is the action to be performed whenever the case occurs. The consequence is composed of a verb in its base form and an entity. For example *if the user's age is less than 15, then omit adult books*, *if the session is expired, then request login*, and *if the book is not registered, then throw error*. This sentences are described in Grammar 4.10.

```
    CONDITIONAL_SENTENCE -> 'if' CASE 'then' CONSEQUENCE
    CASE -> ENTITY CONDITION_MODE CONDITIONAL | CASE 'and' CASE | CASE
↪   'or' CASE
    CONDITION_MODE -> 'is' | 'is' 'not'
    CONDITIONAL -> COMPARATOR NUM | VBP | ADJ | NN
    CONSEQUENCE -> VBB ENTITY
```

Grammar 4.10: Conditional Sentences in SpeCNL

Type Sentences

These sentences are used to express the hierarchical classification of entities. These constructs capture the semantics of classification, i.e. B is a type of A. Concrete examples are: *felines are mammals*, *account holders are users*, *books are loan-items*. Grammar 4.11 is used to construct these sentences.

```
    TYPE_SENTENCE -> SUBTYPE 'are' TYPE
    SUBTYPE -> PLURAL_ENTITY
    TYPE -> PLURAL_ENTITY
```

Grammar 4.11: Type Sentences in SpeCNL

In section 4.2 the process used to generate a ConSpec specification is exemplified.

4.2 ConSpec

In Section 3.2.1, we introduced the concept of *requirements document*, and we discussed the extensive proposal of the IEEE[159] for requirements specification. In here, we present our approach, that is tailored to specify functional requirements for non-critical systems.

Our work is inspired by *legal contracts*[158], which aim to describe as clearly as possible the obligations and concessions between two parties. Legal contracts should contain a well-defined offer, acceptance conditions, and special considerations[78, 90] to be appropriately formed. In general, contracts are well-written documents[170, 253]. We follow the definition of elements and formats of legal contracts to generate our proposed requirements specification.

Other authors have also made use of contracts to design software systems. Back in 1986, Bertrand Meyer proposed *Design by Contract (DBC)*[217, 219], a design principle that imposes formal constraints on software interfaces for the Eiffel programming language. This work is grounded on Hoare Logic[150] that introduces *preconditions* to express the constraints to be satisfied before the execution of a method, *postconditions* to specify the conditions that

should be guaranteed by the program, and *invariants* for the conditions that must hold at all times. A more comprehensive description was presented in Section 3.1.7.2.3.

DBC is suitable for specifying functions/methods in a growing number of programming languages[32, 193, 238]. However, less effort has been paid to the Requirements Engineering Process, which is crucial to make sure the system being built satisfies the expectation of its users and sponsors. Hence, we propose the following contract-inspired specification document, named ConSpec, to address functional requirements for non-critical applications using our SpeCNL described in Section 4.1.

ConSpec is composed of three base elements: a *Title* for the software component to be specified, the current *Version* of the component being developed, and a collection of *Clauses* that are detailed in the following section.

4.2.1 Clause Elements

Clauses are the core component of ConSpec. They describe formally all the activities supported by the software component, including their *roles*, *dependencies*, and *constraints*. They are described next.

Clause Number

The clause number is used to identify every single activity within a contract uniquely. This number is also used to specify dependencies within clauses and for further traceability purposes. Whenever a conflict is detected, one should be able to identify the clause or clauses causing it. The format suggested is that of the letter C as a shorthand for clause, followed by a *unique* combination of digits. The meaning of these digits may vary according to the intention of a user. For instance, in nested clauses, the first digit may represent the main clause, and the consecutive digits may represent the sub-clause number. For example: *C1* could identify the first clause, and *C1.1* could identify the first sub-clause of clause 1. Grammar 4.12 defines the structure of the clause number.

```
CLAUSE_NUMBER -> LETTER_C NUMBER
NUMBER -> NUMBER DOT NUMBER | NUMBER DIG | DIG
```

Grammar 4.12: Structure for cluse number in ConSpec

Activity

An activity describes precisely the function to be specified in the current clause. Its structure is composed of an action, and in some cases, the receiver of an action. The former is expressed through a verb in its base form, whereas the latter, if needed, will be a qualified entity. Note that every clause has *one and only one* corresponding and *unique* activity. Therefore, throughout a specification written in ConSpec, there cannot be duplicated activities. Examples of activities are: *borrow books*, *register*, and *get general-report*. The rules in Grammar 4.13 are used to structure activities.

```
ACTIVITY -> VBB | VBB ENTITY | TO VBB | TO VBB ENTITY
```

Grammar 4.13: Structure for activities in ConSpec

Actor

Actors describe which entities within the system perform the current activity. More than one actor can perform the same activity. Therefore, this section is specified as a collection of individual actors. An actor is either a logical or a physical entity expressed through qualified nouns, such as: *system-administrators*, *customers*, *local-computer*, etc. Actors are constructed using the rules in Grammar 4.14

```
ACTOR -> ENTITY
```

Grammar 4.14: Structure for actors in ConSpec

Condition

Every clause can contain a list of preconditions, a list of postconditions and a list of activity conditions. Based on the work of Hoare[150] and Meyer [217], we define *Preconditions* as constraints that must be satisfied before the execution of the activity; *Activity Conditions* as constraints that must persist through the course of the activity, and *Postconditions* as constraints that must be satisfied after the execution of the activity.

These constraints capture both: structural and behavioural elements of an activity. Note that there are simple requirements that *may not* need some of these conditions to be specified. Conditions are expressed using the grammatical constructions described as *Sentences* in our CNL (see Section 4.1). Grammar 4.15 captures the alternatives to express conditions.

```
    PRECONDITION -> CONDITION
    ACTIVITY_CONDITION -> CONDITION
    POSTCONDITION -> CONDITION
    CONDITION -> STRUCTURAL_SENTENCE | COMPARISON_SENTENCE |
↪   CARDINALITY_SENTENCE | CONDITIONAL_SENTENCE | TYPE_SENTENCE |
↪   CONSTRAINT_SENTENCE
```

Grammar 4.15: Structure for conditions in ConSpec

Consequence

They specify the actions that need to be taken in case any constraints are broken, such as displaying an error message. A clause is a list containing as many consequences as desired. Example of consequences are *show error-dialogue*, *return exit-code*, or *do nothing*. This clause element must follow Grammar 4.16.

```
    CONSEQUENCE -> VBB ENTITY | TO VBB ENTITY
```

Grammar 4.16: Structure for consequences in ConSpec

Dependency

The dependency section of a contract expresses that an activity cannot be performed unless other activities have already been performed successfully. This part of a clause is optional given that there are activities with no dependencies. Dependencies are expressed in an ordered list that contains the *clause number* of the current activity being referenced. The order of the elements in the list represents the order in which other activities must be checked and satisfied before executing the current activity.

A scenario where dependencies could be needed is when we want to say that *before being able to borrow a book, the user must be logged in the system.* In this example, we assume that *login* is an existing activity with clause number *C1* and *borrow books* is the activity of the current clause identified as *C2*; then *C1* should be in the list of dependencies of *C2*.

The grammar required to specify every element of the dependency list is the same as the grammar to specify the clause number, and this *must* be a clause number existing within the current ConSpec specification. This is represented in Grammar 4.17.

```
    DEPENDENCY -> LETTER_C NUMBER
    NUMBER -> NUMBER DOT NUMBER | NUMBER DIG | DIG
```

Grammar 4.17: Structure for dependencies in ConSpec

In this section, we presented the elements that integrate the clauses of our specification format. In section 4.3, we show the steps followed to derive a ConSpec specification from the description of a sample software application.

4.3 Requirements Refinement

Within our approach, requirements refinement is defined as the process followed to translate the needs of the users and sponsors into the clauses of a ConSpec specification. In order to demonstrate this process, we make use of an example commonly found in the literature[26, 62, 142]. This example contains the following requirements for a library system.

> A library issues loan items to customers. Each customer is known as a member and is issued a membership card that shows a unique member number. Along with the membership number, other details on a customer must be kept such as a name, address, and date of birth. The library is made up of a number of subject sections. Each section is denoted by a classification mark. A loan item is uniquely identified by a bar code. There are two types of loan items, language tapes, and books. A language tape has a title language (e.g. French), and level (e.g. beginner). A book has a title, and author(s). A customer may borrow up to a maximum of 8 items. An item can be borrowed, reserved or renewed to extend a current loan. When an item is issued the customer's membership number is scanned via a bar code reader or entered manually. If the membership is still valid and the number of items on loan less than 8, the book bar code is read, either via the bar code reader or entered manually. If the item can be issued (e.g. not reserved) the item is stamped and then issued. The library must support the facility for an item to be searched and for a daily update of records.

Using these existing requirements, we demonstrate the steps required to generate the corresponding specification.

4.3.1 Activity identification

The first step to generate our specification is to identify sentences that express activities to be supported by the software application. In the case of the library example, we separated and enumerated the following sentences describing such activities:

```
-S1: A library issues loan items to customers
-S2: Every customer is issued a membership card
-S3: Items can be borrowed
-S4: Items can be reserved
-S5: Items can be renewed
-S6: Membership cards can be scanned or entered manually
-S7: Items are stamped
-S8: items are issued
-S9: Items can be searched
-S10: Records can be updated
```

4.3.2 Activity specification

Once these sentences are extracted, they have to be translated into activities expressed in the CNL proposed in Section 4.1. To achieve this, we have to convert these sentences into a verb in base form followed by the entity that receives the verb. This translation requires some semantic and grammatical operations. Investigating all the possible operations is an active and continuous research task that is out of the scope of our research. However, we discuss the operations that apply to this example, and show the resulting activities next.

- *Merge equivalent sentences*: S1 and S3 are merged to generate A2 because if the library loans an item, then the item is borrowed.
- *Convert passive voice into active voice*: S4 generates A3 and S5 generates A4 through this operation.
- *Generalise activities*: We generalise S6 as the activity of reading the membership card, either manually or via scanner. Thus A5.
- *Remove non-functional activities*: "Stamping items" in S7 is a physical activity. Therefore, it is omitted from the specification.
- *Redefine action receivers and performers*: Instead of saying that a customer is issued a membership card (S2), we say that the library issues membership cards to its customers (A1).
- *Expand ambiguous sentences*: In S10, the meaning of "update" is not clear. Hence, we clarify by further specifying that items can be registered (A8) and deleted (A9).

The following list contains the activities translated using these operations.

```
-A1: Issue membership-card
-A2: Borrow items
-A3: Reserve items
-A4: Renew items
-A5: Read membership-card
-A6: Issue item
-A7: Search items
-A8: Register item
-A9: Delete item
```

4.3.3 Clause Construction

Once the activities have been refined, the next step is to build up clauses that capture the elements of these activities, as well as adding any other necessary activities. For brevity, we expand only one of the most complex clauses in this section. However, the full specification is provided in the appendix A.2. We have identified and separated the following sentences from the original requirements text that are related to action *A2* in order to illustrate this step.

```
-S11: A library issues loan items to customers
-S12: A customer may borrow up to a maximum of 8 items.
-S13: When an item is issued the customer membership number
      is scanned via a bar code reader or entered manually.
-S14: If the membership is still valid and the number of items
      on loan less than 8, the book bar code is read,
      either via the bar code reader or entered manually.
-S15: If the item can be issued (e.g. not reserved) the item
      is stamped and then issued.
```

The following operations are performed in order to generate the clause elements for A2.

- From S11, we identify *customers* as the actors for this clause.
- From S12 and S14 combined, we extract the precondition that a customer can borrow up to 8 items, which is as *loaned-items must be less than 8*, using the grammar for *Comparator Sentences* from our CNL.
- S6 is used to generate A5, which is an activity on its own.

- From S13, we know that A5 is a dependency for A2. We assume A5 is specified in the clause C5, so the clause dependency is expressed properly.

- S14 contains two additional preconditions: *"if the membership is still valid"*, which is refined into *Customer membership must be valid*, and *"the book bar code is read, either via the bar code reader or entered manually"*, which is refined into *"Book's bar-code must be read"*.

- S15 expresses a postcondition for A2.

- We previously stated that stamping refers to a physical activity, which is ignored by the functional requirements. However, *"... and then issued"* specifies a change in the internal state of the system. This change is refined into the following postcondition: *The item is added to the customer's loaned-items*.

With this reasoning, we generate the following elements of the clause C2 corresponding to the activity *"borrow item"*.

ConSpec 4.1: Library specification, clause C2

```
- C2:
       Activity: Borrow items
       Actors:
           - Customers
       Preconditions:
           - Customer's membership must be valid
           - Loaned-items must be less than 8
           - Book's bar-code must be read
       Postconditions:
           - The item is added to the customer's loaned-items
       Consequences:
           - Return the description of the unsatisfied-precondition
       Dependencies:
           - C5
```

Through this example, we discussed some of the activities needed to refine existing requirements into a ConSpec specification. One could be tempted to automate this refinement process, and we do not discourage the reader from attempting it. However, this activity is left out of the scope of our work, because we prioritised other activities. Nonetheless, this example shows how intuitive it is to specify functional requirements for a software application using SpeCNL and ConSpec.

4.4 Summary

In this chapter, the desired properties in requirements specifications were defined. Our contributions to requirements engineering are presented in the form of a controlled natural language designed to communicate requirements (SpeCNL), and a specification document based on contracts and clauses (ConSpec). The dentition of their elements and an example of their application were also presented here.

Chapter 5

TOMM: a framework for formal reasoning

In this chapter, we describe the theoretical foundations for our framework for formal reasoning over class diagrams. The focus is on class diagrams since this is the (structural) model we generate automatically from natural language requirements. Its elements are a formalization based on predicate logic, a calculus to infer class diagrams, and the axioms to define model validity and equivalence. Throughout this chapter, we make use of the word reliability to refer to the formal strengths of our formal systems. We defined the reliability of our theories based on their soundness and completeness, as is it typically done in formal systems[83, 125, 176].

5.1 Formalization

Expressing requirements specifications and class diagrams in a common language is the first step towards formal reasoning. In Section 3.1.7.1, several logics have been introduced, and here we present a formalization based on predicate logic, which makes it possible to represent and use diagrams and requirements within formal systems.

5.1.1 Requirements Formalization

In Chapter 4, a controlled natural language and a document structure for the specification of functional requirements were presented. Through these elements, requirements are processed thanks to the grammar rules followed, which convey a specific meaning to every element in the specification. The task at hand now is to capture the semantics in a logic language that enables

formal reasoning and seamless interaction with class diagrams. The elements of this language are subsequently described.

5.1.1.1 Elements

In Sections 4.1 and 4.2, the parts of a contract in controlled natural language using Backus-Naur notation were presented. However, these expressions are not valid formulas in predicate logic; therefore they cannot be used in formal reasoning. To solve this problem, formalization is required.

There are two ways to describe the formal notation for specifications, one of them is by enumeration, which requires us to define a one-to-one mapping from the parts of the specification to the elements of the formal notation (cf. used in Section 5.1.2 to formalize class diagrams). The second one is to provide a set of instructions to construct such mappings without enumerating all of them, which is the method used next to describe the predicates of the specification.

1. Every symbol that is a terminal or a non-terminal is interpreted as a predicate in first-order logic.

For instance, the non-terminal *ENTITY* is assumed to be represented as the predicate ENTITY(x). The same assumption holds for terminal symbols such as maximum which is formalized through the predicate MAXIMUM(x).

2. If there is a non-terminal whose substitution rule requires more than one non-terminal symbols, then these symbols are expressed as arguments for the predicate representing the non-terminal. Terminals are not listed in the arguments.

For example in the derivation rule

$$\text{ACTION} \implies \text{MODAL VB}$$

the non-terminal ACTION is replaced with the predicates $\text{ACTION}(x, y) \wedge \text{MODAL}(x) \wedge \text{VB}(y)$. Terminals represent constant values within the structures, hence no need to add them as part of the predicate.

3. The predicate PART_OF(a, b) maintains the relationships between components and subcomponents of a clauses within a contract.

In addition to the clauses derived from the CNL and the contract structure, the predicate above, read as *"b is part of a"* is the one that maintains the structure of the contract document. That is, any element represented by the

variable b is part of the clause represented by the variable a. For example, having predicates CLAUSE(a) and ACTION(x) the action x is assigned to the clause a as PART_OF(x, a). This predicate is of particular importance when inferring class diagrams.

With these rules, all the elements of a ConSpec specification are formalized into predicates, in such a way that only valid predicates are generated from the requirements document. In the following section, an example is presented to discuss the application of these rules.

5.1.1.2 Example

To demonstrate the application of the rules described in the previous section, we use a simple library example. Complete details of the example care provided in the Appendix A.2. The following snippet shows clause `C2` of the specification.

ConSpec 5.1: Library specification, clause C2

```
- C2:
       Activity: Borrow items
       Actors:
           - Customers
       Preconditions:
           - Customer's membership must be valid
           - Loaned-items must be less than 8
           - Book's bar-code must be read
       Postconditions:
           - The item is added to the customer's loaned-items
       Consequences:
           - Return the description of the unsatisfied-precondition
       Dependencies:
           - C5
```

Substitutions showed in Grammar 5.1 exemplify the application of rules in order to generate the activity of the clause.

```
ACTIVITY -> VBB ENTITY
VBB -> 'Borrow'
ENTITY -> QUALIFIED|_ENTITY
QUALIFIED_ENTITY -> QUALIFIER NOP
QUALIFIER -> LAMBDA
NOP -> 'items'
```

Grammar 5.1: Activity generation for the library example

From these transformations we generate the following predicates:

$$\text{ACTIVITY}(borrow, items), \text{VBB}(borrow), \text{NOP}(items)$$

Grammar 5.2 shows the substitutions applied to generate the actor.

```
ACTOR -> ENTITY
ENTITY -> NON
NON -> 'Customers'
```

Grammar 5.2: Actor generation for the library example

Generating the predicates

$$\text{ACTOR}(Customers), \text{NON}(Customers)$$

The first precondition *"Customer's membership must be valid"* is generated using the substitutions shown in Grammar 5.3.

```
PRECONDITION -> CONDITION
CONDITION -> COMPARISON_SENTENCE
COMPARISON_SENTENCE-> ATTRIBUTE CONSTRAINT ADJ
ATTRIBUTE -> ENTITY APOS NON
ENTITY -> NON
NON -> 'Customers' | 'membership'
CONSTRAINT -> OBLIGATION
OBLIGATION -> 'must' 'be'
ADJ -> 'valid'
```

Grammar 5.3: Precondition generation for the library example

From which the following predicates are generated

$$\text{PRECONDITION}(Customers, membership, must, be, valid)$$
$$\text{NON}(Customers), \text{NON}(membership),$$
$$\text{OBLIGATION}(mustbe), \text{ADJ}(valid)$$

,

A similar chain is applied to generate the other elements of the contract. Note that our implementation, described later in Section 6.2.1, allows us to generate these rules automatically. The evaluation of ConSpec is done in Section 7.1, identifying some areas of improvement, such as comparison of non-numerical elements. In the following section, a similar formalization for class diagrams is described.

5.1.2 Class Diagram Formalization

Class diagrams have a well-defined semantics to represent the structure of a system through the enumeration of its components and their relations. Class

diagrams are usually presented graphically. For our work, we use predicate logic to represent these elements mathematically, enabling the application of formal methods, which is not possible over graphical representations.

Different kinds of semantics have been proposed for class diagrams, each of them responding to a different kind of problem. For instance Cabot[59] formalizes class diagrams and OCL constraints into a Constraint Satisfaction Problem[187] in order to verify absence of constraint redundancies.

Berardi et al. proposed a formalization aimed for reasoning using first-order logic[41], wherein the name of the classes, attributes, and operators are treated as individual predicates in order to evaluate object instantiation. This approach results in an infinite set of predicate names because each diagram would have a different set of predicate names. Hence, establishing properties based on the type of predicate would not be possible.

Chanda's group has worked on the traceability of requirements and consistency of class diagrams [66]. They propose the use of grammars to formalize several diagrams and use transformation rules to maintain consistency.

Our approach targets a different problem from those previously mentioned, as it aims to establish the relationship between diagrams and specifications based on their constitutive elements. This problem is tackled by proposing a finite set of predicates and using specific names as their arguments as is shown next.

5.1.2.1 Elements

We present the elements required to formalize UML Class Diagrams in preparation for reasoning activities.

Before defining the elements of a UML Class Diagram, it is necessary to consider the basic building blocks that UML provides within this structural model.

The **UML Components** are:

- **Classifier types in UML**: $C_U = \{class, abstract, interface\}$
- **Visibility**: $V_U = \{+, -, \#, /, \sim, *\}$ where
  - $+$: Public
  - **-**: Private
  - $\#$: Protected
  - $/$: Derived
  - $\sim$: Package

  - *: Random
  - ϵ: Not specified

- **Scope**: $S_U = \{classifier, instance, \epsilon\}$
- **Primitive Types**: $P_U = \{Integer, Boolean, String, UnlimitedNatural, Real, \epsilon\}$
- **Cardinality Symbols**: $\#_U = \mathbb{N} \cup \{m, n, \epsilon, *, +, ?\}$
- **Relation Types**: $T_r = \{Association, Dependency, Aggregation, Composition, Realization, Relation, ClassedAssociation\}$

In what follows, assume given a dictionary D consisting of all words, and thus including all names of elements contained in a diagram. We hence define a class diagram formally as follows.

Class Diagram

A class diagram is a 6-tuple containing the following elements:

$$CD : (C, A, O, R, H, T) \tag{5.1}$$

Where

- C is a subset of the dictionary D containing the names of all the *classes* in the diagram, i.e.,
  $C \subseteq D$
- A is a set of attribute pairs *(class, attribute)*, where every class is an element of the set C, and every *attribute* is an element of the dictionary, i.e.,
  $A \subseteq (C \times D)$
- O is a set of operation pairs *(class, operation)*, where every class is an element of the set C, and every *operation* is an element of the dictionary, i.e.,
  $O \subseteq (C \times D)$
- R is a set of relation pairs *(source, target)*, where both source and target are elements of the set C, and the source class is in a relationship with target class, i.e.,
  $R \subseteq (C \times C)$

- H is a set of pairs *(super, sub)*, where both super and sub are elements of the set C, sub is a subclass of super, and hence *inherits* from super, i.e.,
  $H \subseteq (C \times C)$

- T is a set of *valid types* for the class diagram, it is the union of the standard primitive *types* defined in UML, and all classes C contained in the diagram, i.e.,
  $T \subseteq (C \cup P_U)$

From the last item, we note that C is used to represent both the set of class names used in a diagram and the type associated with it. It is generally clear from the context what we mean. We note that from the definition above classes, attributes and operations all have names and we are not imposing any constraints on their uniqueness, etc. For example, two different classes `c1` and `c2` may have an attribute with the same name `a`, and hence `(c1,a)` and `(c2,a)` both belong to the set of attribute pairs A. Furthermore, if `(a,a)` belongs to A, then this would mean that there is a class with name `a` and attribute `a`. Though this may be a poor design choice, we stress that our formalization does not restrict any possibly erroneous specification.

Predicates

Together with the sets enumerated above, the following predicates are used to define a Class Diagram:

- CLS(c, t) : indicates that the variable c (also a class name in C) is associated to a classifier type where t indicates the type of classifier, that is $t \in C_U$.

- ATR(c, a, t, v, s) : is used to express that class c has an attribute a of type $t \in T$, with visibility $v \in V_U$ and scope $s \in S_U$.

- OPR(c, o, t, v, s, P) : indicates that class c contains an operation o with visibility $v \in V_U$ and scope $s \in S_U$. This operation receives the set of parameters P and has return type $t \in T \cup void$.

  - $P \subseteq D \times T$: every element in P is a pair (n, t), where n is the name of the parameter, and t is its type.

- TYPE(t) : indicates that t is a type. These predicates are required to indicate the type for attributes, and the return type for the operations within a class.

- $\mathrm{REL}(s, d, t, n, r, \#_l, \#_u)$: indicates that there is a relationship (of type $t \in T_r$) between classes s and d. This relation has name n, role name r (at the d side), and $\#_*$ indicates the cardinality of the relationship such that $\#_l, \#_u \in \#_U$, and $\#_l$ is the lower boundary, and $\#_u$ is the upper boundary.

- $\mathrm{INH}(g, s)$: indicates that class g generalizes s, or conversely, that s is a specialization of (inherits from) g.

Not all elements from the above predicates have to be defined, and we either omit the undefined parameter when it is clear what we mean or write ϵ explicitly for clarity. When possible we do the former to keep the presentation more readable.

Though relations are currently not used in TOMM, they are included in here for syntactical completeness. Each relation predicate represents one single association from a source to a destination, which implies that in order to express bidirectional relations, it is necessary to decompose the relation into two separate predicates. This type of predicates can be used in the future for validation of cardinality constraints.

Axioms

Notice that in this section classifier types, visibility and scope are omitted to keep the predicates as short as possible, including only relevant elements to identity every predicate.

- $\forall x \in C | \, \mathrm{CLS}(x) \wedge \mathrm{TYPE}(x)$

- $\forall (x, y) \in A, \exists z \in T | \, \mathrm{ATR}(x, y, z) \wedge \mathrm{CLS}(x) \wedge \mathrm{TYPE}(z)$

- $\forall (x, y) \in O, \exists z \in T, \forall (a, b) \in P | \, \mathrm{OPR}(x, y, z, P) \wedge \mathrm{CLS}(x) \wedge \mathrm{TYPE}(x) \wedge \mathrm{TYPE}(b)$

- $\forall (x, y) \in R, t \in RT, n, r \in D, \#_l, \#_u \in S | \, \mathrm{REL}(x, y, n, r, \#_l, \#_u) \wedge \mathrm{CLS}(x) \wedge \mathrm{CLS}(y)$

- $\forall (x, y) \in H | \, \mathrm{INH}(x, y) \wedge \mathrm{CLS}(x) \wedge \mathrm{CLS}(y)$

Considering that there are no constraints when manually drawing diagrams, these predicates do not contain constraints either, making it possible to capture any drawn diagram. We will deal with the validity of diagrams in Section 5.3, where we check for properties such as circular inheritance.

Though these elements are sufficient to capture any diagram, some of them, such as visibility or interfacing, are not further expanded since they are irrelevant for the problem domain which we focus on.

5.1.2.2 Example

The formalization shown above is sufficient to capture the elements of any class diagram. We support this claim by showing how different diagrams are expressed using this formalism. Our examples are taken from different sources found in the literature, together with the NLRP benchmark repository[290].

Our first example is the diagram corresponding to the requirements for a library system found in appendix A.1. Several proposals for this diagram are found in the work of Callan[62], Harmain[142], Berardi[42] and Gelhausen[128], all of them with different strategies for abstraction and different level of details.

We use the original diagram developed by Callan[62] to demonstrate the basic features of our formalism.

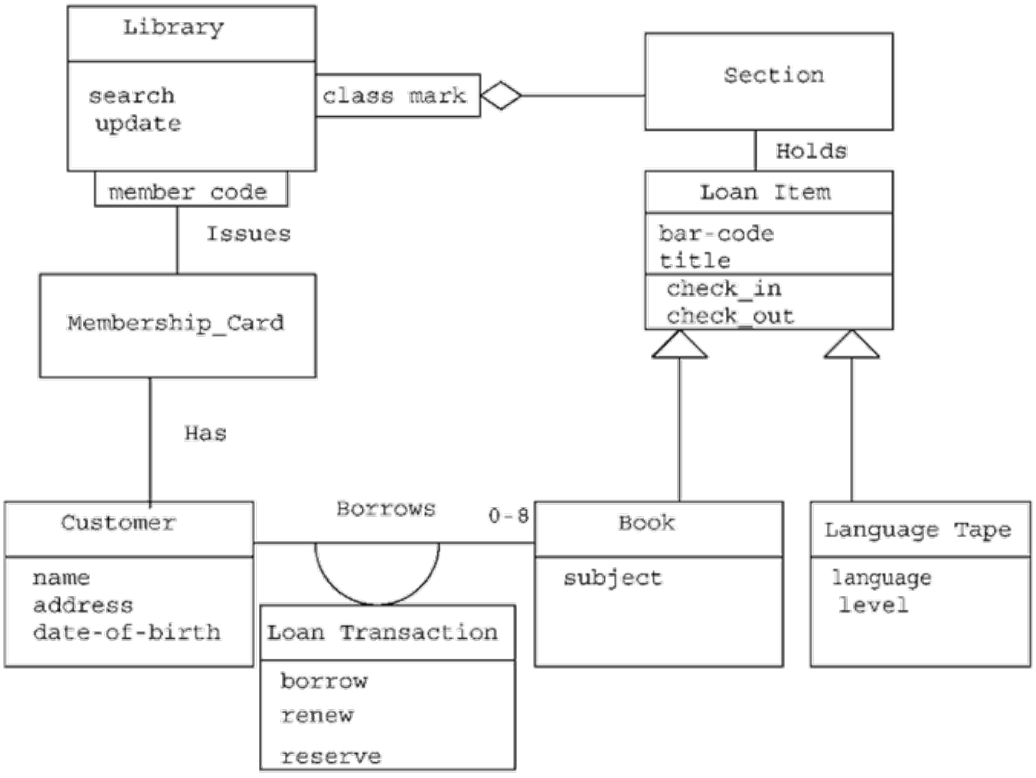


Figure 5.1: Callan's Class Diagram

This diagram encompasses examples of inheritance, aggregation, association, and one association class. Its formal representation done manually begins with the following sets.

```
D = {Library, search, update, member code, Issues, Membership_Card,
Has, Customer, name, address, date-of-birth, Loan Transaction,
```

```
borrow, renew, reserve, Borrows, Book, subject, Language Tape,
language, level, Loan Item, bar-code, title, check_in, check_out,
subsection, Hols, class mark}

C = {Library, Membership_Card, Customer, Loan Transaction,
Book, Language Tape, Loan Item, subsection}

A = {
  (Loan Item, bar-code),
  (Loan Item, title),
  (Customer, name),
  (Customer, address),
  (Customer, date-of-birth),
  (Book, subject),
  (Language Tape, language),
  (Language Tape, level)
}

O = {
  (Loan Transaction, borrow),
  (Loan Transaction, renew),
  (Loan Transaction, reserve),
  (Loan Item, check_in),
  (Loan Item, check_out)
}

R = {
  (Library, Membership_Card),
  (Membership_Card, Customer),
  (Customer, Loan Transaction),
  (Customer, Book),
  (Loan Transaction, Book),
  (subsection, Loan Item),
  (Library, subsection)
}

H = {
  (Loan Item, Book),
  (Loan Item, Language Tape)
}
```

With the previous sets, we generate the following predicates corresponding to the classes.

$\text{CLS}(Library) \wedge \text{TYPE}(Library)$
$\text{CLS}(Membership\_Card) \wedge \text{TYPE}(Membership\_Card)$
$\text{CLS}(Customer) \wedge \text{TYPE}(Customer)$
$\text{CLS}(LoanTransaction) \wedge \text{TYPE}(LoanTransaction)$
$\text{CLS}(Book) \wedge \text{TYPE}(Book)$
$\text{CLS}(LanguageTape) \wedge \text{TYPE}(LanguageTape)$
$\text{CLS}(LoanItem) \wedge \text{TYPE}(LoanItem)$
$\text{CLS}(subsection) \wedge \text{TYPE}(subsection)$

Some of the attributes of this class diagram are represented in the following predicates.

$\text{ATR}(LoanItem, title, \epsilon, \epsilon, instance) \wedge \text{CLS}(LoanItem) \wedge \text{TYPE}(\epsilon)$
$\text{ATR}(LoanItem, barcode, String, \epsilon, instance)$
$\text{ATR}(Customer, name, String, \epsilon, instance) \wedge \text{TYPE}(String)$
$\text{ATR}(Customer, address, Address, Private, instance) \wedge \text{TYPE}(Address)$
$\text{ATR}(Book, subject, \epsilon, Public, instance)$

Note that the first row above uses the three predicates of the rule: ATR, CLS and TYPE, however, in the following examples we skip them since some classes and types have already been defined, and we know that $P(x) \wedge P(x) \vdash P(x)$. Also, notice that we are referring uniquely to instance attributes. Even though the visibility is not specified in the original diagram, we stated that Customer.address is private and Book.subject is public to show the use of different visibility levels.

None of the attributes has a type defined, hence the use of ϵ in most of the cases. However, to extend this example, we provided Loan_Item.bar-code, Customer.name and Customer.address with specific types: a primitive String, and a complex type Address. With this extension, the *"Address"* type should also be defined as a class within a complete diagram, but this evaluation is not considered until Chapter 7.

Some examples of operations are:

$\text{OPR}(LoanTransaction, borrow, \epsilon, \epsilon, instance, \{\})$
$\text{OPR}(LoanItem, check\_in, void, Public, classifier, \{(barcode, String)\})$

In this case, we kept $Loan\_Transaction.borrow$ as in the diagram, where the type and visibility are not specified, and there are no parameters received.

However, we extended $Loan\_Item.check\_in$ in order to exemplify the remaining components of an operation. Thus we defined return type to be void, visibility to be Public, scope to be classifier, and parameters to be the String representing the bar code of the item to be checked in. This representation of the operation could be considered equivalent, for they solve the same problem differently. We properly define equivalence in Section 5.4.

In the original diagram we observe different types of relations, which are:

REL($Library, Membership_Card, Relation, Issues, membercode, \epsilon, \epsilon$)
REL($Library, subsection, Aggregation, \epsilon, classmark, \epsilon, \epsilon$)
REL($Customer, Book, ClassedRelation, borrows, \epsilon, 0, 8$)
REL($Customer, LoanTransaction, ClassedRelation, borrows, \epsilon, \epsilon, \epsilon$)
REL($LoanTransaction, Book, ClassedRelation, borrows, \epsilon, \epsilon, \epsilon$)

The first line captures the relation between Library and Membership_Card with the role name member code. The second entry captures an aggregation from Library to subsection, with the role name class mark. The last three entries represent the classed-relation borrows, between Customer and Book through the class Loan Transaction.

Inheritance in Callan's diagram exist between Loan Item, Book and Language Tape, and they are captured in the following statements.

INH($LoanItem, Book$)
INH($LoanItem, LanguageTape$)

Note that though these predicates capture the diagram in Figure 5.1, they do not provide any reasoning or verification on it. Reasoning rules will be described in detail in Section 5.2, Section 5.3 and Section 5.4.

5.2 Class Model Inference

In Section 5.1, we presented the formalization of ConSpec specifications and UML class diagrams. We defined predicates in first-order logic as the common language to integrate and enable formal reasoning over these artefacts. These predicates will be used here to describe a formal system that will enable us to infer the predicates of a class diagram from the predicates of a ConSpec specification.

5.2.1 Inference Calculus

In logics, rules of inference are expressed in the following form, where premises and conclusions are both valid formulas within the system. This representation is read as *"if premises 1, 2 and 3 hold, the conclusion also holds"* .

$$\frac{\text{Premise1} \qquad \text{Premise2} \qquad \text{Premise3}}{\text{Conclusion}}$$

The well-formed formulas within our formal system are those of the form of any of the predicates describing ConSpec specifications or class diagrams defined in Sections 5.1.1.1 and 5.1.2.1 respectively.

The problem to solve is phrased as *"Given the set of predicates describing a specification, what predicates of a class diagram can be inferred?"* From here, it is assumed that the predicates of the specification are known to be true; hence they are called axioms. The elements of the diagram are to be proven; hence they are theorems. Then, it is the case that.

1. Any rule within the system can only be applied over axioms, or over previously inferred theorems.

With the previous consideration, the following inference rules are defined.

Class Rule

This rule allows us to infer classes and types from the actors of a specification. It is directly applied over the *ACTOR* predicate, which means that the variable x can be either a qualified entity or any noun (plural or singular). In this rule it is assumed that every class is a *classifier* type of UML; hence it does not allow to infer abstract classes or interfaces.

$$\frac{\text{ACTOR(x)}}{\text{CLS(x, classifier)} \wedge \text{TYPE(x)}}$$

Non-parametric Operation Rule

In order to infer operations that do not take parameters, we have to apply this rule. The premises required are the predicates *CLS*, *ACTIVITY* and *VBB*. In here, the class is required to exist before inferring the operation, and the activity defines the name of the operation, to do so, this predicate must receive only one parameter, which must be a verb in the infinitive form. By definition, all operations are of type ϵ, which means that no return type is specified. Also all classes are *public (+)* by default and their scope is *instance*. Finally, the set of parameters is empty, which indicates that the operation receives no parameters.

$$\frac{\text{CLS(x, *)} \qquad \text{ACTIVITY(y)} \qquad \text{VBB(y)}}{\text{OPR(x, y, } \epsilon\text{, +, instance, \{\})}}$$

Uni-parametric Operation Rule

This is an extended version of the *non-parametric operation rule* to deal with operations that receive *only one* parameter. The difference is that this extension requires the *ACTIVITY* predicate to receive a pair of parameters, where the first one is a verb in the infinitive and the second one is a qualified entity. It is observed that the type, visibility and scope of the operation are the same as in the non-parametric version; however, the parameters are composed by a set containing the qualified entity of the activity.

$$\frac{\text{CLS(x, *)} \qquad \text{ACTIVITY(y, z)} \qquad \text{VBB(y)} \qquad \text{QUALIFIED\_ENTITY(z)}}{\text{OPR(x, y, } \epsilon\text{, +, instance, \{z\})}}$$

The current version of ConSpec only allows us to describe up to uni-parametric operations; hence only these rules are part of the inference system. Further research may extend ConSpec and the inference system to support a more general form of parametric operations.

Attribute Rule

This rule is applied over structural sentences in order to infer the attributes of a class. Notice that the application of this rule requires the class to exist before inferring the attribute, hence no attribute can be inferred for a non-existing class. The predicate for structural sentences requires three variables: the attribute x, the corresponding form of the verb *"to have"* expressed in the variable y, and the entity that has the attribute captured by the variable z. Similarly to the operation rules, the type of the parameter is not specified (ϵ), the visibility is *public* (+) and the scope is *instance*.

$$\frac{\text{CLS(z, classifier)} \qquad \text{STRUCTURAL\_SENTENCE(x,y,z)}}{\text{ATR(x, z, } \epsilon\text{, +, instance)}}$$

Inheritance Rule

This rule is required in order to infer the inheritance of the classes within the diagram from the type sentences of the specification. The predicate for type sentences here contains two parameters, the entity being the subtype, which is the variable x and the entity of the superclass, the variable y.

$$\frac{\text{TYPE\_SENTENCE(x, y)}}{\text{INH(y,x)}}$$

These rules provide a basic inference system to generate the predicates of a class diagram from a given specification. The consistency of our system is established in the following section.

5.2.2 Reliability

The diagrams generated by our inference calculus can only be as reliable as the rules used to generate it. To establish a measure of the reliability of our inference method, its *consistency*, *soundness* and *completeness* will be proved. To do so, a definition of each property is provided before the actual proof.

Definition 1. A formal system is **consistent** if and only if no contradictions are derived from the application of the inference rules.

A contradiction occurs when a formula ϕ and its negation $\neg\phi$ are both found within the axioms or the theorems of the system. The formulas for the system proposed are predicates for the elements of diagrams and specifications. The system is composed only by these predicates and the inference rules listed before. None of the inference rules produces the negation of a predicate. From which it follows that it is impossible to have a formula of the form $\neg\phi$. Thus the system cannot present any contradiction, that is, *the system is consistent.*

Definition 2. A formal system is **sound** if and only if every formula that can be proved in the system is valid with respect to the semantics of the system.

It is known that all formulas are predicates, and all the predicates proved (derived) correspond uniquely to the elements of the class diagram. In consequence, it is needed to establish the semantics of the predicates for the diagram, to do so, the following conditions must be established.

1. Every formula can only be be applied in the context of a given clause.

The following statements define the semantics of the predicates for class diagrams and ConsSpec specifications.

Definition 3. Semantics for diagram and specification predicates

1. *CLS* predicates are valid only if they are inferred from an *ACTOR* predicate.
2. *OPR* predicates are valid only if they are inferred from an *ACTIVITY* predicate.
3. *ATR* predicates are valid only if they are inferred from an *STRUCTURAL_SENTENCE* predicate.
4. *INH* predicates are valid only if they are inferred from an *TYPE_SENTENCE* predicate.

That means that all formulas have to be applied one by one to every clause in the contract. The class rule can only be applied in the actors section of the contract. Operation rules must be applied in the activity section. Attribute and inheritance rules can be applied in any of the condition fields, that is preconditions, activity conditions, and postconditions.

From the rules listed in the previous section, it is clear that these relations are satisfied, and since there are no other ways to generate predicates, it is proved that *the system is sound*.

Definition 4. A formal system is **complete** with respect to a given property if and only if every formula having that same property can be derived using that system.

In this particular system, completeness is evaluated in terms of the predicates of the class diagram, then the question to answer is *"can the inference rules proposed derive all the predicates associated with a class diagram?"* ; and the answer is *no*. It is enough to consider the diagram predicate *REL*, which captures the relationships between classes; this predicate is not the conclusion of any inference rule, and since these rules are the only way to generate predicates, then *REL* predicates cannot be generated. Thus it is proved that *the system is not complete*.

It has been proved that our inference system is consistent and sound, though not complete. With this consideration, we move forward to exemplify the use of our inference system in the following section.

5.2.3 Example

To demonstrate the use of these rules, the clause `C2` of the specification presented in Section 4.3 is used as a reference, which expresses the requirements for a basic library system. This clause is repeated in the following contract segment.

ConSpec 5.2: Library specification, clause C2

```
- C2:
       Activity: Borrow items
       Actors:
           - Customers
       Preconditions:
           - Customer's membership must be valid
           - Loaned-items must be less than 8
           - Book's bar-code must be read
       Postconditions:
           - The item is added to the customer's loaned-items
       Consequences:
           - Return the description of the unsatisfied-precondition
       Dependencies:
           - C5
```

As it was established in Section 5.2.2, the inference rules must be applied in the context of each clause to maintain soundness. The *class rule* is applied first as shown next to generate the class and type for the customer.

$$\frac{\text{ACTOR(Customers)}}{\text{CLS(Customers, classifier), TYPE(Customers)}}$$

The *uni-parametric operation rule* is then applied to the previously generated class and the activity of the clause. For clarity of the inference rule, the following formula assignments are done.

$$\begin{gathered}
\alpha = \text{CLS}(Customers, classifier) \\
\beta = \text{ACTIVITY}(Borrow, items) \\
\gamma = \text{VBB}(Borrow) \\
\delta = \text{QUALIFIED_ENTITY}(items) \\
\phi = \text{OPR}(Customers, Borrow, \epsilon, +, \text{instance}, \{\text{items}\})
\end{gathered} \tag{5.2}$$

$$\frac{\alpha \quad \beta \quad \gamma \quad \delta}{\phi}$$

In this clause there are no structural sentences or type sentences; hence there is no need to apply the attribute or inheritance rules. The partial class model generated by inference from the clause 2 of the library specification looks like follows:

CLS($Customers, class$)
TYPE($Customers$)
OPR($Customers, Borrow, \epsilon, +, instance, \{items\}$)

We just illustrated the use of the inference rules within a ConSpec clause for our simple library system. However, manual inference is time-consuming and error-prone, and it is hence crucial to develop tools that automate this process; this is the subject of Chapter 6.

In this section, we defined a formal system to infer class diagrams from requirements specifications, stated the inference rules, established its consistency and soundness, and demonstrated its application with an example. In the next section, we describe a formal system for class diagram validation.

5.3 Class Model Validation

In Section 3.1.7.3, existing approaches for model verification were introduced, together with the properties to be checked. A characteristic of these properties is that they are checked within the model itself, i.e., they do not maintain any relation with other artefacts, such as specifications. The novelty of our approach is that it aims to establish a relationship between the model and the specification, namely model validity.

This concept has been influenced by Somerville's definition of requirements validation[281] described in Section 3.1.5.4. Sommerville states that "requirements validation is the process of checking that requirements actually define the system that the customer really wants". In the same way, we state that *model validation is the process of checking that models define the requirements specification as intended*. Throughout this section, the elements class diagram validation will be defined as a formal system. Also, we establish its reliability and illustrate how it is used with an example.

5.3.1 Validation Calculus

In computer science, the correctness of a program is established through the application of inference rules to every instruction within the program. In this

way, the result of the program is inferred and compared with the expected result according to the specification. These inference rules belong to Hoare logic[150], and were introduced in Section 3.1.7.2.3.

Similarly, the validity of a model with respect to its specification is established through the application of inference rules over the elements of the model. In this case, the model corresponds to a class diagram, and the specification corresponds to a ConSpec contract. Their elements are expressed in the form of predicates, as described in Sections 5.1.1 and 5.1.2 respectively. Before proceeding to define the rules for this formal system, some definitions must be provided.

First, the relevant concepts must be defined as follows.

Definition 5. A model is **valid** if and only if it is sound and complete. This definition is formalized as:

$$Valid(M) \implies Sound(M) \wedge Complete(M)$$

Where M is any given model.

Definition 6. A model is **sound** if and only if all of its elements are derived from the specifications. It is formally expressed as:

$$Sound(M) \implies \forall \phi \in M, \exists \psi \in S \mid \psi \models_R \phi$$

In here, M is the set of predicate formulas describing the model, S is the set of predicate formulas describing the specification, ϕ is a predicate from the model, and ψ is a predicate from the specification. In this context the $\models_R$ symbol indicates that the predicate ϕ is inferred from the application of the rule R over the predicate ψ.

In this way, *soundness* establishes the relation from the predicates of the model M to the predicates of the specification S'.

Definition 7. A model is **complete** if and only if all the elements of the specification are related to an element in the model. Using the previous notation, we formally describe completeness as follows:

$$Complete(M) \implies \forall \psi \in S, \exists \phi \in M \mid \psi \models_R \phi$$

Inversely to soundness, *completeness* establishes the relation from the predicates of the specification *S'* to the predicates of the model *M*.

In Section 5.2.2 the completeness property for formal systems was discussed, and it was concluded that the absence of this property is to some extent expected. That is the case, for instance, of the sentences that do not contribute to a class diagram, such as *"the age must be less than 15"* . In its current version, our formalism for diagrams does not support OCL constraints, which would be required to capture this sentence. Another case occurs with predicates that contain repeated elements, such as *STRUCTURAL_SENTENCE(users, have, age)* and *NOUN(age)*; both predicates are part of *S*, but only one is used to derive an element of *M*.

With this consideration, it seems too restrictive to say that a model is not valid just because it is not complete, as a consequence, the decision was made to distinguish between two types of completeness, defined as:

Definition 8. A model presents **total completeness** if absolutely all the elements of the specification are related to the elements of the diagram. Hence the formal definition for completeness given before is the actual definitions of total completeness, which is:

$$Totally\_Complete(M) \implies \forall \psi \in S, \exists \phi \in M \mid \psi \models_R \phi$$

Definition 9. A model presents **partial completeness** if a well-defined subset of the elements of the specification is related to the elements of the diagram. Partial completeness is formally defined as:

$$Partially\_Complete(M) \implies \forall \psi \in S', \exists \phi \in M \mid \psi \models_R \phi$$

Where S' is the subset $S' \subset S$ containing all the specification predicates that must be used to infer elements for the diagram.

In the same way, validity cannot be just one property; it has to correspond to the type of completeness observed in the model; thus we define the following types of validity.

Definition 10. A model observes **strong validity** if it is sound and

presents total completeness.

$$Strong\_Validity(Model) \implies Sound(Model) \wedge Totally\_Complete(Model)$$

Definition 11. A model observes **weak validity** if and only if it is sound an presents partial completeness.

$$Weak\_Validity(Model) \implies Sound(Model) \wedge Partially\_Complete(Model)$$

The inference rules described in the previous section are applicable only over a subset of the predicates of specifications. The same subset of predicates will be used to establish *weak validity* in our validity calculus. This subset S' is defined as:

Definition 12. Subset of predicate statements for weak validity

$$S' = [ACTOR, ACTIVITY, VBB, QUALIFIED\_ENTITY, STRUCTURAL\_SENTENCE, TYPE\_SENTENCE]$$

In order to construct this validation calculus, the concept of *semantic equivalence* is required to establish that two words can have the same meaning in the context of the specification. For example, in a banking system, the words *"customer"* and *"client"* may be equivalent. This relation is particularly important when validating models against specifications because it has been observed that terminology may vary from one artefact to another. However, this variation should not affect the validity of the model. In here, semantic equivalence is represented with the symbol $\approx$. This relation has to be established manually before proving validity, so in the application of validity rules, it is assumed to exist. In concrete we have that:

Definition 13. Semantic equivalence is the relation $w_1 \approx w_2$, where w_1 and w_2 are words that have the same meaning in the context of the proof. If w_1 and w_2 are the same, then they are also semantically

equivalent, that is.

$$w_1 = w_2 \implies w_1 \approx w_2$$

The definitions stated before, the inference rules from Section 5.2.1 the semantics described in Section 5.2.2, and the semantic equivalence relation just presented, are the blocks required to define the following axioms for validity. These axioms are used to prove *weak validity* of a model M with respect to a specification S'.

Class Axiom

This axiom states that any class in M can be derived from S' using the Actor rule from the validation calculus.

$$\frac{\alpha_a \models_C \gamma_{x'} \qquad \gamma_x \qquad x \approx x'}{\top}$$

Where the predicate *ACTOR(a)* is represented by α_a, the application of the *class rule* is symbolised by $\models_A$, the predicate $CLS(x,*)is\gamma_x$, $\gamma_{x'}$ *is the predicate* $CLS(x', *)andtheexpressionx \approx x'$ *represents the semantic equivalence of* x *and* x'. *Note that the* $*$ *symbol in the CLS predicates represent any value. From these premises, we conclude* $\top$, *which means the axiom holds.*

Operation Axiom

This axiom establishes that any operation in M is derived from an activity in S'

$$\frac{\delta_a \models_O \omega_{x'} \qquad \omega_x \qquad x \approx x'}{\top}$$

In this case, the formula δ_a *represents the predicate ACTIVITY(a,* $*$*), the formulas* ω_x *and* $\omega_{x'}$ *represent OPR(*$*$*, x,* $*$*,* $*$*,* $*$*,* $*$*) and OPR(*$*$*, x',* $*$*,* $*$*,* $*$*) respectively, and* $\models_O$ *represents the application of the* operation rule *over* δ_a *to derive* $\omega_{x'}$ *The structure of this and next axioms follow the same pattern as the* class axiom.

Attribute Axiom

This axiom establishes the relationship between structural sentences and attributes generated with the attribute rule.

$$\frac{\sigma_{ab} \models_A \rho_{x'y'} \qquad \rho_{xy} \qquad x \approx x' \wedge y \approx y'}{\top}$$

.

In a structural sentence, two distinctive elements are required: the entity and the attribute of the entity. They are represented in the formula σ_{ab}. for the predicate STRUCTURAL_SENTENCE(a,∗,b). The attribute predicate ATR(x, y, ∗, ∗, ∗) is captured in the ρ formulas with parameters x, y and x', y'. This axiom holds if $x \approx x' \wedge y \approx y'$, which means that both attributes and both entities are semantically equivalent.

Inheritance Axiom

This axiom proves the relationship between inheritance predicates and type sentences through the inheritance rule*.*

$$\frac{\tau_{ab} \models_I \eta_{x'y'} \qquad \eta_{xy} \qquad x \approx x' \wedge y \approx y'}{\top}$$

.

In this axiom the formula τ represents the TYPE_SENTENCE predicate, and the formula η represents the INH predicate. As usual, $\models_I$ represents the application of the inheritance rule *over the type sentence.*

With these axioms, it is possible to determine the weak validity *of a model with respect to a specification based on the model* soundness *and* partial completeness*. In the next section, the reliability of our axiom system is discussed.*

5.3.2 Reliability

Similarly to the inference calculus, the reliability of this validation calculus depends on three properties: *consistency*, *soundness* and *completeness*.

First, we analyse *consistency* which is the lack of contradiction within the formal system. To evaluate this property, it is necessary to look at the formulas of the calculus and the inference rules. Notice that all the rules produce tautologies; hence, contradictions are unreachable, that is, *the system is consistent.*

To evaluate the *soundness* of the **system** it is necessary to differentiate it from **model soundness**, which aims to establish a relation between models and specifications. In contrast, *system soundness* establishes the relation between the inferred formulas and their truth value with respect to a given interpretation.

In Section 5.3.2 the interpretation for the formulas of the system was provided. With the newly introduced rules for validation no new formulas are generated, for all of them derive in a tautology. With this reasoning,

it is established that all existing formulas are axioms with properly-defined interpretations; hence, *the system is sound*.

A similar case occurs for *system completeness* which shall not be confused with *model completeness*. The system is complete if all the valid formulas can be generated from the axioms and the inference rules. As already mentioned, the inference rules only derive tautologies; hence no new formulas are generated. However, all possible formulas already exist as axioms; thus *the system is complete*.

In conclusion, *this formal system is consistent, sound and complete*. In the next section, we elaborate an example of model verification.

5.3.3 Example

The axioms for validity together with the definitions of model soundness and partial completeness have been introduced, and our validity calculus has been proven to be consistent, sound and complete. In this section, we demonstrate how to prove weak validity of a given model. Once more we make use of the library example. The predicates for the clause `C2` of the specification are shown below.

$ACTOR(Customers)$
$ACTIVITY(Borrow, items)$
$VBB(Borrow)$
$QUALIFIED\_ENTITY(items)$

Together with the predicates of the specification, an equivalence relation must be defined.

$Customer \approx Client$
$Customer \approx Customers$

For this example, the empty model denoted by $M = \{\}$ or $M = \emptyset$ is checked. To proceed, the class axiom is evaluated first over the actor clause, from which the premise

$$\text{ACTOR}(Customers, classifier) \models_C \text{CLS}(Customers)$$

is derived. However, the model M does not have a class predicate CLS(x) such that $x \approx Customer$ hence the axiom does not hold, and the model is incomplete.

Now assume the same model also has a predicate for the abstract class customer, that is $M = \text{CLS}(Customer, abstract)$. The first premise

$$\text{ACTOR}(Customers) \models_C \text{CLS}(Customers, classifier)$$

is known to exist, the second premise CLS($Customer, abstract$) also exists because it was just added to the model. The third premise $Customer \approx Customers$ from the equivalence relations also exists. Since all the formulas in M have an equivalent formula derived from the specification S, we infer that the model is sound.

However, the specification S contains an *ACTIVITY(Borrow, items)* whose production is not equivalent to any element in the model. To solve this problem, predicate OPR($*, Borrow, *, *, *, *$) is added to the model. In this way, the premise $\delta_a \models_O \omega_{x'}$ is applied, and the newly added predicate takes the place of the premise ω_x, with $x = Borrow$ and $x' = Borrow$ the premise $x \approx x'$ also holds. In this way, the new model M is both sound, and complete with respect to the predicates defined in *S'* before.

These examples show an invalid model due to lack of soundness, and invalid model due to lack of partial completeness, and finally a weakly valid model.

Two calculus have been proposed, one for model inference and another for model validation. In the following section, we provide a calculus for equivalence.

5.4 Class Model Equivalence

The number of class diagrams created throughout the development life cycle of a software application is considerable; this is usually the result of changes in the requirements or in the software that are not updated in previously existing diagrams. Another reason is that multiple stakeholders may generate diagrams from different perspectives (also known as viewpoints), for instance, there can be a diagram that describes only the problem domain, whereas there are others that also capture the solution domain. For this reason, there is a need to compare different class diagrams. Also, it makes it possible to compare different solutions generated by different teams.

The first approach for diagram comparison may be based on human judgement and perception. A stakeholder may, for instance, consider different diagrams and decide on how similar they are or whether they are equivalent. However, this manual approach is time-consuming and error-prone. Furthermore, it is not practical for large system models. Our approach instead relies on the formalization described in Section 5.1.2 and a formal system to establish the equivalence of two class diagrams formally.

This work should reduce human effort and error in the scenarios described before, at the same time that it is used to automatically check academic assignments, purge documentation of existing software systems, evaluate

designs and generate augmentation for machine learning.

In this chapter, our approach for model comparison is expressed in terms of a formal system to establish the equivalence of two models. An analysis of its reliability and an example are also shown.

5.4.1 Equivalence Calculus

In this section, a formal system to compare class diagrams is described. It takes two formal class models and establishes the equivalence relationship between them. Two types of equivalence are defined here, which we call *left equivalence* and *right equivalence*. They are determined based on a set of axioms presented below.

Let us introduce first de definition of model equivalence.

Definition 14. Two models M_1 and M_2 are equivalent if:

1. For every predicate in M_1 there is an equivalent predicate in M_2.

$$\forall \phi_{lX} \in M_1, \exists \psi_{mY} \in M_2, \phi_{lX} \approx \psi_{mY}$$

2. For every predicate in M_2 there is an equivalent predicate in M_1.

$$\forall \psi_{mY} \in M_2, \exists \phi_{lX} \in M_1, \psi_{mY} \approx \phi_{lX}$$

Where ϕ and ψ are formulas representing predicates of type l and m applied to the tuple of parameters X and Y. Also, M_1 and M_2 are equivalent if both are empty:

$$M_1 = \emptyset \wedge M_2 = \emptyset$$

In the previous definition, the concept of *predicate equivalence* has been mentioned, and now it is defined as follows.

Definition 15. Two predicates ϕ_{lX} and ψ_{mY} with predicate names l and m and parameters X and Y are equivalent if:

1. The predicate names of ϕ and ψ are the same.

$$l = m$$

2. The number of parameters is the same for both predicates.

$$|X| = |Y|$$

3. The i-th parameter in ϕ is equivalent to the i-th parameter in ψ.

$$\forall x_i, y_i \; x_i \approx y_i$$

Where $x_i \approx y_i$ is the same equivalence relation from Definition 13.

The previous axioms allow us to establish the equivalence of two class diagrams through their models. However, it is the case that some diagrams can be partially equivalent as well, that is to say, that they contain some similar elements but not all of them. To differentiate these cases, two types of equivalence are proposed: *left equivalence* and *right equivalence*, which are defined as follows:

Definition 16. Left equivalence occurs when all the elements of M_1 have an equivalent element in M_2. This corresponds to the first rule of Definition 14.

Definition 17. Right equivalence occurs when all the elements of M_2 have an equivalent element in M_1. This corresponds to the second rule of Definition 14.

In this way, it is possible to establish a more accurate comparison between the two class models. The following rules of inference allow us to reason about the equivalence of every predicate ϕ_{lX} and ψ_{mY} within models M_1 and M_2.

Class Axiom

This axiom allows us to establish the equivalence between two classes.

$$\frac{\gamma_x \qquad \gamma_y \qquad x \approx y}{\top}$$

Where γ_x is a formula of the type $CLASS(x)$ and γ_y is a formula of the type $CLASS(y)$. In here, x and y represent the name of the classes; hence, the type of class (classifier, abstract or interface) does not affect their equivalence.

Operation Axiom

This axiom allows us to establish the equivalence between two operation predicates.

$$\frac{\omega_{ab} \qquad \omega_{xy} \qquad a \approx x \wedge b \approx y}{\top}$$

For these formulas, the predicate name is OPR, and the variables a, x represent the class for the operation and the variables b, y represent the name of the operation. Other parameters, such as visibility and scope are not checked.

Attribute Axiom

This axiom allows us to establish the equivalence between two attribute predicates.

$$\frac{\rho_{ab} \qquad \rho_{xy} \qquad a \approx x \wedge b \approx y}{\top}$$

The ATR predicates are checked here, where the variables a and x represent the class of the attribute, and variables b and y represent the name of the attribute.

Inheritance Axiom

The equivalence of the class inheritance captured in INH predicates is established with this axiom.

$$\frac{\eta_{ab} \qquad \eta_{xy} \qquad a \approx x \wedge b \approx y}{\top}$$

In here, a and x are the names of the superclasses, and b and y are the subclasses.

To this point, model equivalence has been defined, and two types of equivalence have been discussed. The rules to prove equivalence have also been presented. As it has been done with our inference and validation calculus, we proceed now to evaluate the reliability of our equivalence calculus.

5.4.2 Reliability

All the formal systems previously proposed have been evaluated in term of consistency, soundness and completeness, not being this an exception, we proceed to discuss these properties.

The inference rules defined within the equivalence calculus result all in tautologies. As a result, no contradictions are derived; hence it is demonstrated that *the system is consistent*.

The predicates of the class models are always true, no other predicates are possible, and hence *the system is sound and complete*. With these properties being demonstrated, an application of this calculus is given below.

5.4.3 Example

To demonstrate the usage of the equivalence calculus, we need two class diagrams which we would like to compare and check for equivalence. Similarly to the example of the validation calculus, we start with the empty model. Let $M_1 = \emptyset$ and $M_2 = \emptyset$. By definition, these two models are equivalent.

Now assume that the predicate $\text{CLS}(Customer, classifier)$ is added to M_1. There is not a premise of the type $\text{CLS}(x, *)$ in M_2 such that $Customer \approx x$, hence the class axiom does not hold over M_1 and the model does not satisfy *left equivalence*.

In addition, assume now that $\text{CLS}(Client, abstract) \in M_2$ and that $Client \approx Customer$. In this case, the class axiom holds over M_1, and it is also the case for M_2, hence both models are equivalent.

Now assume that $\text{INH}(User, Client) \in M_1$. It is known that the class axiom holds over M_1 and M_2, since there are no more elements in M_2 then the *right equivalence* is satisfied, however, the *left equivalence* is not, due to the newly added class. These examples cover the evaluation of left and right equivalence to determine model equivalence.

5.5 Summary

A framework for formal reasoning over class models was presented. The framework encircles the formalization of ConSpec specifications and Class Diagrams using predicates, and a calculus for model inference, validation and comparison (aka. equivalence). Every calculus is accompanied by a set of inference rules, an evaluation of its consistency, soundness and completeness, and an example to illustrate its application. Though still limited, these formal systems set the ground for new approaches to reason about models, and their relation with requirements specifications. More importantly, the presented

framework underlies the proof-of-concept that we have developed and which is discussed in detail in Chapter 6.

Chapter 6

T4TOMM: a proof-of-concept for TOMM

In this chapter, we describe the implementation of a proof-of-concept that supports our formal framework developed through Sections 5.2, 5.3, and 5.4. We have named this implementation T4TOMM, which stands for *Tool 4 Thinking of Models and More*, and in its current stage, it supports model inference, validation and comparison. In order to implement these theories, we have resorted to existing developments, such as languages, solvers and processors, all of which will be discussed within the chapter. Towards the end of the chapter, we will have introduced the elements of our implementation, the justification of the decisions made, and the illustration of the usage of T4TOMM.

T4TOMM currently supports the following functions:

- Automatic formalization of YAML-based ConSpec specifications into SMT-LIB models.
- Automatic formalization of JSON-based class models into SMT-LIB models.
- Automatic integration of SMT-LIB models to infer JSON class diagrams from YAML ConSpec specifications.
- Automatic integration of SMT-LIB models to validate JSON class diagrams against YAML ConSpec specifications.
- Automatic integration of SMT-LIB models to compare JSON class diagrams.

- Processing of SMT-LIB models using CVC4 to generate, validate and compare classes, attributes, operations and inheritance in class models.
- Partial JSON Class model extraction (mainly attributes and operations) from image-based class diagrams.

6.1 Resources

In this section, we introduce the existing resources we have used in the development of T4TOMM. They are separated into two categories, those related to natural Language processing, and those related to formal reasoning with satisfiability modulo theories.

Based on our experience with programming languages, we considered both Java and Python as candidates for the development of our proof-of-concept. We noticed a growing tendency in the use of Python in industry and open source projects, and, hence, our choice of Python as the primary programming language to develop T4TOMM. As a consequence, the selection of NLP and SMT were also influenced by this decision.

6.1.1 Natural Language Processing

This category enumerates libraries that aid the processing of English texts, in particular, those used to process ConSpec requirements specifications written in SpeCNL. Two processing tasks are required within the implementation of T4TOMM; they are parts-of-speech (POS) tagging, and sentence parsing using Context Free Grammars (CFG).

The Natural Language Toolkit (NLTK)[44] was the most suitable candidate for both the tasks, primarily due to its compatibility with the leading Operative Systems and the amount of data available thanks to its open-source nature; this makes it a perfect fit for our proof-of-concept. Additionally, it is also developed for Python, our choice of programming language.

Parts of speech tagging[206] is the activity that consists of identifying the grammatical category of a word, i.e. Parts of Speech (POS) within a sentence. Examples of POS in English are nouns, verbs and prepositions. The most commonly used tags are appropriately defined in the Penn Treebank tag set[20], which is also followed by most of the POS taggers. However, these tags are not the same as the ones required for TOMM, as defined in Section 4.1.1. To solve this problem, we have defined the equivalences captured in table 6.1

NLTK allows performing POS tagging in Python by providing an interface to different taggers. For the implementation of T4TOMM, the off-the-shelf

Table 6.1: Equivalence of POS tag

SpeCNL tag	Treebank tag
NOS	NN
NOP	NNS
NOS	NP
NOP	NPS
VB	VV
VBP	VVD
VBI	VBN
VBI	VVN
VBB	VVP
VB	VVZ
VBB	VB
ADJ	JJ
INT	CD

English tagger is sufficient to perform the desired task. In the current implementation, we assume that the tagger returns the correct tags for each sentence. However, we are aware that this is not always the case. Therefore, we envision a future version where more than one tagger can be combined with the user input in order to ensure that each word is correctly tagged. This limitation does not affect the theory proposed in TOMM due to the assumption mentioned before.

Sentence parsing consists of identifying the sequence of grammatical rules necessary to the construction of a given sentence. The set of all possible rules to enable this construction is called a Context Free Grammar (CFG) and is composed of terminal and non-terminal symbols. Non-terminal symbols are related to the words in the sentence, whereas non-terminal are treated as placeholders in the parsing tree. For instance, in the sentence "dogs run", both words "dogs" and "run" are terminals, and we define "SENTENCE", "NOUN", "VERB" to be non-terminals. Finally, the following CFG is used to parse sentences that have a noun and a verb, like the one mentioned earlier.

```
SENTENCE -> NOUN VERB
NOUN -> dogs
VERB -> run
```

T4TOMM requires sentence parsing to validate that a specification is properly written i.e. that it follows the rules described in SpeCNL and ConSpec. T4TOMM also makes use of sentence parsing to formalize a contract automatically.

This task is achieved utilizing NLTK parsing capabilities. In our current implementation, we use the efficient Chart Parser, which generates a growing

chart with all possibilities for each subtree[44]. We parse individually every element of the specification with its corresponding CFG, which then reuses the elements of SpeCNL. These grammars are presented in Section 6.2 of this chapter.

6.1.2 Satisfiability Modulo Theories

We introduced SMT solvers in Section 3.1.7.4 to evaluate logic formulas with respect to a combination of the proposed theories. In this section, we will discuss the elements used in the implementation of T4TOMM to support the theories described as part of TOMM for model inference, validation and comparison.

6.1.2.1 SMT-LIB

SMT-LIB[33, 34] is an international effort to provide a common language to express SMT theories that to be used by different solvers. Together with the input and output language, it describes theories for common types, such as integer, real and floating-point numbers, and a set of logics to deal with different elements, such as quantifiers and arrays. We opted for SMT-LIB because it has been accepted as a standard by the major players in the SMT community, and its documentation is accessible and clear.

The elements of SMT-LIB used within T4TOMM are constants, functions and datatypes, together with quantifiers, logic operators, assertions, queries and model descriptors. Though these elements are conceptually similar to the ones found in major programming languages, writing SMT-LIB models demands a change in the way we think about them. First of all, it follows Polish (prefix) notation[141] i.e. the operator comes at the beginning, followed by the operands. For example, instead of $a + b$ one must write $+ab$; in small examples, this may seem trivial, but it becomes more complex when extended to represent sufficiently complicated models. Another consideration is that SMT-LIB is not a programming language, but a formal reasoning language, which has no instructions or procedures are coded. Instead, constraints and questions are expressed. These difficulties may seem overwhelming for any software developer. Hence, the need for T4TOMM to ease the use of SMT solvers with user-friendly models and specification.

6.1.2.2 CVC4

CVC4[35] is an SMT solver that supports both, its own language, and SMT-LIB. Its more relevant features include support for datatypes, strings and

finite sets. These features justify our decision of CVC4 over other solvers, such as Z3[96]. In addition to these features, CVC4 provides binaries for most major operating systems, which eases the installation and configuration process.

Unlike NLTK, CVC4 itself does not integrate with Python, and though there are official interfaces for C++ and Java, the Python API is neither complete nor adequately documented. For this reason, CVC4 has to be executed from a terminal session initiated from our python application, which is in charge of generating the corresponding SMT-LIB models, as we will be describing it in the next section.

SMT-LIB and CVC4 were chosen due to our familiarity with these tools, in addition to their ability to deal with formal theories and to generate proofs. SMT solvers also represent a potential for future support of OCL constraints.

6.1.3 Image processing

As part of T4TOMM, we have incorporated a module to extract information from the graphic representation of class models. This is described in Section 6.6. This particular module requires the usage of several libraries for image processing. Since our core application is developed in Python, we favoured the usage of libraries written for Python. In particular, we have used scikit-image, scikit-learn and pillow.

Scikit-image is a library that implements various Image Processing algorithms. We made use of it in order to perform various image manipulations, including colour transformations (RGB, grey scale and black and white), and basic image filtering (line segmentation, skeletonization and labelling). These algorithms are required to perform image segmentation.

Scikit-learn is a machine learning library that provides optimal implementations of the conventional algorithms within the field. In particular, we used it to perform numeric clustering and segmentation. Both the image and the learning libraries make use of the numpy library that contains mathematical functions for data manipulation.

Pillow is a top-level implementation of the underlying Python Imaging Library, which eases image generation and handling. We made use of this library to generate intermediate representations during the segmentation process, and to produce the images of individual segments.

In order to perform Optical Character Recognition (OCR), the task of extracting text from images, we evaluated python-tesseract and google vision API. We opted to use tesseract because it is free and performed better than the Google vision API when extracting the information of the classes. We also determined that from the different segmentation modes supported by

tesseract, the one that assumes that assume the image is a block of vertically aligned text works the best to extract attributes and operations, though it struggles to deal with class names. We accepted this trade-off because intuitively we know that modifying one class name requires less effort than modifying more than one attribute or operation.

These libraries allow us to extract the textual components for each class, as described and Section 6.6.2.

6.2 Meta-Models

In this section, we present the SMT-LIB components that are reused for the following tasks: inference, validation and comparison. We also present the strategy used to instantiate each of these components with their specific values for each of those tasks.

6.2.1 Specification

This meta-model contains the definition of all the used components of a ConSpec requirements specification.

6.2.1.1 Datatypes

In Sections 5.2 and 5.3, we introduced a collection of rules used to derive models from specifications and to validate models against specifications. These rules require a formal representation in SMT-LIB using clauses, actors, structural sentences and type sentences. There are in turn captured by the following datatypes.

SMTLib 6.1: Specification Metamodel

```
; Spec datatypes
; Actor
(declare-datatypes ((Actor 0))
    (((ACTOR (actor_name String)))))
; Clause
(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (c_actors (Set Actor))
        ))))
; Sentence
(declare-datatypes ((Sentence 0))
    (((SENTENCE
        (words (Set String))
        ))))
; Structural Sentence
(declare-datatypes ((Structural_S 0))
```

```
    (((STRUCT
        (entity String)
        (modal String)
        (have String)
        (property String)
        ))))
; Type Sentences
(declare-datatypes ((Type_S 0))
    (((TYPE_S
        (subtype String)
        (isa String)
        (type String)
        ))))

; CSD sets
(declare-fun clauses () (Set Clause))
(declare-fun actors () (Set Actor))
(declare-fun actors () (Set Actor))
(declare-fun struct_sents () (Set Structural_S))
(declare-fun type_sents () (Set Type_S))
```

These datatypes are reused in model inference and model validation as well. Note that we have additionally added a *Sentence* datatype, something not explicitly defined in our theories. However, this datatype helps the implementation of the inference rules in SMT-LIB.

Also notice that in Section 5.2.1, we defined the structural sentence as `STRUCTURAL_SENTENCE(a,b,c)` where a is an entity, b is a modal and c is a property. Whereas in our SMT-LIB representation, we use the "have" element with the intention of having a more readable model, of the type `(STRUCT "entity" "must" "have" "property")` , which makes it more readable than `(STRUCT "entity" "must" "property")` . The same reasoning is applied to type sentence, which in our theory are captured as `TYPE_SENTENCE(a, b)` but in our SMT-LIB meta-model are expressed as `(TYPE_S "Admin" "is a" "User")` .

6.2.1.2 Automatic Formalization

In order to formalize contract specification, POS tagging and sentence parsing are required. Each of the individual elements of a contract is parsed using a specific grammar, as defined in Section 4.2.1 and built on top of the elements of SpeCNL (Section 4.2). Each Context-Free Grammar is captured in a modular file, and our NLPHandler builds dynamically the required grammar based on the element of a clause to be parsed. For example, in order to parse an action, the handler loads the specific grammar for actions, sentences, concepts, and parts of speech.

The following CFGs exemplify the representation of the ones described in Chapter 4 as read by NLTK.

```
    MODAL -> 'can' | 'must' | MD

    COMPARATOR -> INEQUALITY 'or' EQUALITY | INEQUALITY | EQUALITY |
↪   COMPARATOR_SYMBOL
    INEQUALITY -> 'greater' 'than' | 'less' 'than'
    EQUALITY -> 'equal to'
    COMPARATOR_SYMBOL -> '>' | '<' | = | <= | >=

    ENTITY -> PLURAL_ENTITY | SINGULAR_ENTITY | QUALIFIED_ENTITY | NON |
↪   ATTRIBUTE
    SINGULAR_ENTITY -> SINGULAR_INDICATOR QUALIFIER NOS
    PLURAL_ENTITY -> PLURAL_INDICATOR QUALIFIER NOP
    QUALIFIED_ENTITY -> QUALIFIER NON
    SINGULAR_INDICATOR -> 'a' | 'an' | 'one' | '1' | 'the'
    PLURAL_INDICATOR -> INT | 'the'
    QUALIFIER -> ADJ | NN | VBI | QUALIFIER'-'QUALIFIER | LAMBDA

    ATTRIBUTE -> ENTITY APOS NON | NEUTRAL_INDICATOR NON 'of' 'the' NON

    ACTION -> VB | MODAL VB
```

Grammar 6.1: SpeCNL concepts

```
    SENTENCE -> STRUCTURAL_SENTENCE | COMPARISON_SENTENCE |
    ↪  CARDINALITY_SENTENCE | CONDITIONAL_SENTENCE | TYPE_SENTENCE |
    ↪  CONSTRAINT_SENTENCE

    CONSTRAINT_SENTENCE -> ATTRIBUTE 'must' 'be' EXPECTATION
    EXPECTATION -> VBI | JJ

    STRUCTURAL_SENTENCE -> NON MODAL 'have' STRUCTURAL_ITEM
    STRUCTURAL_ITEM -> ENTITY | ENTITY',' STRUCTURAL_ITEM | ENTITY', and
↪   'STRUCTURAL_ITEM

    COMPARISON_SENTENCE -> ATTRIBUTE COMPARISON_OPERATOR NUM
    COMPARISON_OPERATOR -> 'must' 'be' COMPARATOR

    CARDINALITY_SENTENCE -> ENTITY ACTION LIMIT NUM NON
    LIMIT -> 'up' 'to' | 'at' 'least' | 'maximum' | 'minimum' |
↪   'exactly'

    CONDITIONAL_SENTENCE -> 'if' CASE 'then' CONSEQUENCE
    CASE -> ENTITY CONDITION_MODE CONDITIONAL | CASE 'and' CASE | CASE
↪   'or' CASE
    CONDITION_MODE -> 'is' | 'is' 'not'
    CONDITIONAL -> COMPARATOR NUM | VBP | ADJ | NN
    CONSEQUENCE -> VBB ENTITY

    TYPE_SENTENCE -> SUBTYPE 'are' TYPE
    SUBTYPE -> PLURAL_ENTITY
    TYPE -> PLURAL_ENTITY
```

Grammar 6.2: SpeCNL sentences

In addition to the elements of ConSpec and SpeCNL, it is necessary to add rules to parse the terminals each sentence. However, it is technically infeasible to list all possible terminal words as part of one single grammar. Hence, we generate terminal rules dynamically for every individual element to be parsed by tagging the words and adding the rules of the form `POST_TAG -> 'word'` . For example, if the element to be parsed is the action "save document", tagging this sentence we obtain (W save) and (NN document). Thereby, we generate the rules `W -> 'save'` and `NN -> 'document'` .

In this implementation, we deal with two types of semantic equivalence based on word variations. The first type is *synonyms*, for which we make use of WordNet[222] and its python API in order to query the synonyms for the corresponding tagged word. The second type has to do with plurals, and the third one with capitalization. These two types of word variation are tackle with the python library call inflection[1], which enables to generate different

[1]https://inflection.readthedocs.io/en/latest/

capitalization options and pluralization.

Additional word variations, such as domain-dependant semantics are not deal with automatically; hence, they have to be explicitly stated in the formal models. An example of this case is the concept *"Customer"* , which in a banking domain can be equivalent to *"Client"* ; however, this is not the case for marking companies, in which *"Customer"* refers to the users that access the marking campaigns, while *"Client"* refers to the sponsors of such campaigns.

In the formal models, equivalence is captured in the form of set membership, due to the constraints of SMT-LIB. That is, if two words *"w1"* and *"w2"* are equivalent, then they are members of the set same set. In this way, we can find equivalent words by exploring the set of synonyms. An example is shown in SMTLib 6.2

SMTLib 6.2: Example of sets for semantic equivalence and function to determine if two words are equivalent

```
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Paper"  "paper"  "PAPER"  "Papers"  "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert  "submit"  "SUBMIT"  "Submit"  "submits"
    "subject"  "SUBJECT"  "Subject"  "subjects"
    (singleton "submit"))) syns_dict))

(assert (= syns_dict (as univset (Set Entry))))

define-fun is_syn ((w1 String) (w2 String)) Bool
    (or
        (= w1 w2)
        (exists
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))
```

In this way, we dynamically generate complete grammars that allow parsing every individual element of the contract: actors, clauses numbers, activities, conditions, etc..

One consideration is that the grammar for activity defined in Section 4.2.1 makes it difficult for the tagger to tag the corresponding verb properly. To solve this problem, during the automatic formalization we prepend the preposition "to" to the sentence before tagging. By performing this, the verb is identified with the tag "VB". This addition is only applicable to aid the tagger, hence no need to add "to" in corresponding grammar.

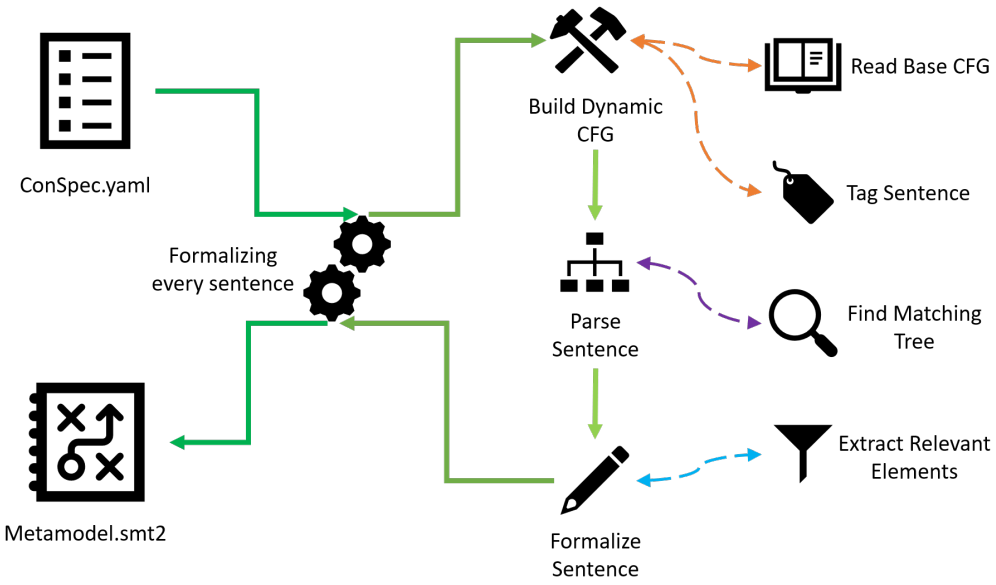


Figure 6.1: Contract Formalization Pipeline

Figure 6.1 illustrates the pipeline followed to formalize a specification into its SMT-LIB representation. The dashed arrows represent activities that have to be performed before moving to the next processing stage.

Notice that the input is a YAML file similar to what was depicted in 6.1. We have chosen a YAML file for the contract because its structure based on indentation is simple and does not require any additional symbols. With this format, the specification document is written using only our CNL.

ConSpec 6.1: Specification structure

```
Title: Sample Contract
Version: 1

Clauses:
  - C1:
      Action: register user
      Actors:
          - Admins
      Preconditions:
          - User's name must be provided
          - User's address must be provided
          - User's date-of-birth must be provided
      Action conditions:
          - The system must generate a user id
      Postconditions:
          - The user id must be unique
  - C2:
      Action: delete user
      Actors:
          - Admins
```

```
    Preconditions:
        - User's name must be provided
        - User's address must be provided
        - User's date-of-birth must be provided
    Consequences:
        - Consequence 1
        - Consequence 2
    Dependencies:
        - Dependency 1
        - Dependency 2
```

6.2.2 Class Diagram

Similarly to the previous section, here, we present the SMT-LIB elements that compose the meta-model of a Class Diagram. In addition to inference and validation, this meta-model is used to compare two given class diagrams.

6.2.2.1 Datatypes

The following STM-LIB datatypes represent the predicates described in Section 5.1.2.1.

SMTLib 6.3: Class Diagrams Datatypes

```
; UML datatypes
(declare-datatypes ( (Class     0) ) ( ((CLASS
                                        (cls_nme String)
                                        (cls_typ String)
                                    ))))
(declare-datatypes ( (Type      0) ) ( ((TYPE
                                        (typ_nme String)
                                    ))))
(declare-datatypes ( (Attribute 0) ) ( ((ATR
                                        (atr_cls String)
                                        (atr_nme String)
                                        (atr_typ String)
                                        (atr_vis String)
                                        (atr_sco String)
                                    ))))
(declare-datatypes ( (Param     0) ) ( ((PARAM
                                        (prm_nme String)
                                        (prm_typ String)
                                    )
                                    (nil))))
(declare-datatypes ( (Operation 0) ) ( ((OPR
                                        (opr_cls String)
                                        (opr_nme String)
                                        (opr_typ String)
                                        (opr_vis String)
                                        (opr_sco String)
                                        (opr_prm Param)
                                    ))))
(declare-datatypes ( (Relation 0) ) ( ((REL
                                        (rel_src String)
                                        (rel_des String)
```

```
                                        (rel_typ String)
                                        (rel_nme String)
                                        (rel_rol String)
                                        (rel_c_l String)
                                        (rel_c_u String)
                                    ))))
(declare-datatypes ( (Inheritance     0) ) ( ((INH
                                        (inh_sup String)
                                        (inh_sub String)
                                    ))))
```

In addition to the predicates, we need to capture the sets described in Section 5.1.2. However, instead of using the representation described in the theory, the use of SMT-LIB demands an alternative declaration of the sets by using the datatypes described in 6.3. This adaptation is depicted in 6.4.

SMTLib 6.4: Class Diagram Sets

```
; UML sets
(declare-fun classes    () (Set Class))
(declare-fun types      () (Set Type))
(declare-fun attributes () (Set Attribute))
(declare-fun operations () (Set Operation))
(declare-fun relations  () (Set Relation))
(declare-fun inheritances  () (Set Inheritance))
```

6.2.2.2 Automatic Formalization

In order to input a class diagram to T4TOMM, we have proposed a JSON structure that enables us to capture the relevant elements of a diagram. The OMG has defined the XML Metadata Interchange (XMI)[11] as a standard to represent UML diagrams using tags. However, JSON documents have over the years become predominant, especially in web technologies, due to their simplified syntax. For this reason, we have considered JSON over XMI to represent class diagrams, as a future version of T4TOMM is envisioned to be accessed from any web browser and to provide web-services. The structure proposed looks as follows.

JSON Model 6.1: Class Diagram structure

```
{
    "classes": {
        "classa": {
            "name": "ClassA",
            "attributes": {
                "attra": {
                    "name": "attrA",
                    "type": "epsilon",
                    "visibility": "-",
                    "scope": "instance"
```

```
                },
                "attrb": {
                    "name": "attrB",
                    "type": "epsilon",
                    "visibility": "-",
                    "scope": "instance"
                }
            },
            "operations": {
                "opa-epsilon": {
                    "name": "opA",
                    "type": "epsilon",
                    "visibility": "+",
                    "scope": "instance",
                    "parameters": {}
                },
                "opb-epsilon": {
                    "name": "opB",
                    "type": "epsilon",
                    "visibility": "+",
                    "scope": "instance",
                    "parameters": {}
                }
            }
        },
        "classb": {
            "name": "ClassB",
            "attributes": {},
            "operations": {}
        }
    },
    "associations": {
        "classa-classb-association": {
            "source_class_name": "ClassA",
            "destination_class_name": "ClassB",
            "type": "association",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        }
    }
}
```


The JSON representation is read by T4TOMM Python's implementation producing the corresponding classes that are later translated into the SMT elements described above. Notice that unlike the formalization of the specification, this process does not require any natural language processing.

6.3 Inference

In Section 5.2.1, we described a set of rules that allows us to formally infer a class model from a Activity requirements specification. In this section, we provide the representation of these rules as SMT-LIB functions that enable such inference from the meta-model described in Section 6.2.1.

The following function captures the *class rule*, which allows us to infer classes and types from the actors of a contract.

SMTLib 6.5: Class Rule

```
(define-fun infer-classes () Bool
    (forall
        ((x Actor))
        (=>
            (member x actors)
            (exists
                ((y Class))
                (and
                    (member y classes)
                    (and
                        (=
                            (cls_nme y)
                            (actor_name x))
                        (=
                            (cls_typ y)
                            "classifier")))))))

(define-fun infer-types () Bool
    (forall
        ((x Actor))
        (=>
            (member x actors)
            (exists
                ((y Type))
                (and
                    (member y types)
                    (=
                        (actor_name x)
                        (typ_nme y)))))))
```

In order to express the inference rules as constraints to generate the model, we declare boolean functions. In our theory, we express the rule in terms of an individual actor. However, in the function `infer-classes`, we use the `forall` quantifier to assert that there is a class for every single actor, which ensures completeness.

The `infer-class` function is read as "For every actor that is a member of the set of actors of a specification, there exists a class that is member of the set of classes of a diagram, such that the name of the class is the same as the name of the actor, and that class is 'classifier"'.

The membership constraints `(member x actors)` and `(member y classes)` are significant because SMT solvers reason about the universe of elements. That is, if we were reasoning about the integers `1, 2, 3`, CVC4 would try to build a model based on the universe of integers, which would include other integers besides `1, 2, 3`. Similarly, if we do not assert the membership of the variable `(x Actor)` to the set `(declare-fun actors () (Set Actor))` defined in the meta-model, we may be inferring classes for actors which do not

belong to the given specification. In this case, we used the "classifier" type for all classes, because no further information can be automatically extracted from the specification alone.

Note that unlike the *class rule*, our SMT-LIB implementation has a separated inference for types; this is to help CVC4 by reducing complexity with several quantifiers in one same formula. Additionally, the `infer-types` function only states that the name of the type must be the same as the actor.

The operation rules used to infer simple and parameterized operations presented in Section 5.2.1 are implemented in the following SMT-LIB model.

SMTLib 6.6: Class Rule

```
(define-fun infer-operations () Bool
    (forall
       ((x Clause))
        (=>
            (member x clauses)
            (infer-operation x))))

(define-fun infer-operation ((c Clause)) Bool
    (forall
        ((a Actor))
        (=>
            (and
                (member a actors)
                (member a (g_actors c))
            )
            (member (mk-operation a (activity c)) operations))))

(define-fun mk-operation ((a Actor) (activity String)) Operation
    (OPR (actor_name a) (get-activity activity) "e" "+" "instance" (get-param activity)))

(define-fun get-activity ((activity String)) String
    (ite
        (str.contains activity " ")
        (str.substr activity 0 (str.indexof activity " " 0))
        activity))

(define-fun get-param ((activity String)) Param
    (ite
        (str.contains activity " ")
        (PARAM (str.substr activity (str.indexof activity " " 0) (str.len activity)) "e")
        nil))
```

In this model, the function `infer-operations` is used to initialize the inference process. This function is used to assert that for every clause that belongs to the set of clauses of a specification, an operation has to be inferred using the `infer-operation` function. The helper constructing function `mk-operation` receives the actor and the activity of the clause and instantiates a new instance operation.

To achieve this construction, the helper function `get-activity` parses the activity string of the clause, so that it only takes the first word before a

space. Besides, the `get-param` helper finds a possible parameter within the activity string, assuming the parameter is found after the first space between two words. The current implementation infers operations from activities composed by only one verb and one parameter at most, for example, *"register user"* and *"login"* .

This implementation allows identifying the actor of the activity as the class of the operation, the first word of the activity as the name of the operation, and the second word of the activity as the parameter, as required by the corresponding inference rule.

Similar to the operation rule are the implementations of the attribute, inheritance, and attribute rules. They all have initializer functions that "iterates" over the elements of each set of the specification in order to infer its corresponding element of the class diagram. The complete implementation of the inference rules together with an example are provided in Appendix B.1

The output of the SMT-solver is an SMT-LIB model like the one shown in CVC4 Output for inferred class model. In it, it is observed that all classes, attributes, operations and inheritances are described as singletons. Our python implementation parses this answer in order to generate the corresponding python object, which, if needed, can be derived into the formal representation of a diagram. A desirable extension would be a visualization of the model generated in the form of a diagram.

SMTLib 6.7: CVC4 Output for inferred class model

```
(singleton (CLASS "Library" "classifier"))
(singleton (CLASS "Loan-Item" "classifier"))
(singleton (OPR "Library" "loans" "e" "+" "instance", (as emptyset (Set Params))))
(singleton (OPR "Library" "issues" "e" "+" "instance", (as emptyset (Set Params))))
(singleton (ATR "Books" "title" "e" "e" "e"))
(singleton (ATR "Members", "date-of-birth" "e" "e" "e")))
(singleton (INH "Loan-items" "Books"))
```

6.4 Validation

In Section 5.3.1, the rules necessary to validate a class model against a given specification were discussed. Throughout this section, we will discuss the structure of the SMT-LIB implementation of most relevant rules.

Both sections, validation and diagram equivalence, make use of the definition of concept equivalence. For T4TOMM, two words equivalent if they have the same meaning, or if they have the same base word. For example, *"customer"* and *"Customer"* are equivalent words varying only in capitalization, *"customer"* and *"customers"* are equivalent varying in the number,

and *"customer"* and *"client"* have an equivalent meaning. In most general cases, any synonyms of a word are its equivalent.

In order to define word equivalence, we provide the following SMT-LIB model.

SMTLib 6.8: Word Equivalence

```
(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                        (synonyms (Set String))
                                    ))))

(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "w1"  "word1"  "Word1"  "Word 1"
    (singleton "w 1"))) syns_dict))


(assert (member (mk-entry (insert
    "word2"  "w2"
    (singleton "W 2"))) syns_dict))

(assert (= syns_dict (as univset (Set Entry))))
```

In this model, the `Entry` datatype represents an element of the dictionary of synonyms. Every entry is a set of equivalent strings. From the example above, it is observed that *"w1"* , *"word1"* , *"Word1"* , *"Word 1"* , and *"w 1"* are equivalent. Every set of equivalences is added to the dictionary, which will then be used in the validation and equivalence tasks.

The current python implementation generates these sets automatically by varying each of the words found in the class diagrams and the specifications. However, further semantic equivalence is required in order to enable the introduction of exceptional cases. For example, in a bank system, the terms *"customer"* and *"cardholder"* may be equivalent, which cannot be automatically identified without user input.

The *actor rule* $\exists CLS(x, e) \in M, \exists ACTOR(y) \in S \mid x \approx y$
expresses the basic condition that there is a class and a actor whose names are *equivalent*.

In order to prove the soundness of the diagram, we have to prove that this rule is *true for all the classes*, which we write in SMT-LIB as follows.

SMTLib 6.9: Class Soundness

```
; There is an actor thar derives the class
(define-fun check_class ((c Class)) Bool
    (exists
        ((a Actor))
        (and
            (member a actors)
```

```
            (is_syn
                (cls_nme c)
                (actor_name a)
            )
        )
    )
)

; All the classes come from an actor
(define-fun actors_rule () Bool
    (forall
        ((c Class))
        (=>
            (member c classes)
            (check_class c)
        )
    )
)
```

The function `actors_rule` allows to reason about all classes, while the function `check_class` verifies that there is at least one actor whose name is equivalent to the name of the class being analysed.

Inversely, we need a rule to verify completeness of the model, i.e. to check that all the actors in the specification have an equivalent class in the diagram. We use the following functions to prove class completeness.

SMTLib 6.10: Class completeness

```
(define-fun check_actor ((a Actor)) Bool
    (exists
        ((c Class))
        (and
            (member c classes)
            (is_syn
                (actor_name a)
                (cls_nme c)
            )
        )
    )
)

(define-fun inverse_actors_rule () Bool
    (forall
        ((a Actor))
        (=>
            (member a actors)
            (check_actor a)
        )
    )
)
```

In this model, the function `inverse_actors_sule` allows to reason about all the actors of the specification, while the function `check_actor` is responsible for proving that there is a class whose name is equivalent to the name of the actor.

A similar implementation is made for the remaining rules, which are fully described in Appendixes B.2 and B.3. Note that these rules also depend on the meta-model described in Section 6.2.

In order to check either completeness or soundness, we must evaluate (assert) each of the rules, and we must assign the output to a variable that can be queried in SMT-LIB to conclude the validity of the model. We demonstrate these steps in the following model.

SMTLib 6.11: Checking for Soundness

```
;-----------------------------------
; Validating model
;-----------------------------------
(declare-const actors_validation Bool)
(declare-const operations_validation Bool)
(declare-const attributes_validation Bool)
(declare-const inheritances_validation Bool)
(assert (= actors_validation actors_rule))
(assert (= operations_validation operations_rule))
(assert (= attributes_validation attributes_rule))
(assert (= inheritances_validation inheritances_rule))


;-----------------------------------
; Checks
;-----------------------------------
(check-sat)
(get-value (actors_validation))
(get-value (operations_validation))
(get-value (attributes_validation))
(get-value (inheritances_validation))
(exit)
```

Notice that the boolean constants named as `*_validation` store the result of the evaluation of the corresponding rule. For instance, `actors_validation` stores the result of the boolean function `actors_rule` .

At the bottom of the model, we have the `(check-sat)` command which effectively executes the solver in order to prove our rules. After checking for the satisfiability of the model, we query the value of these variables. Should all of them be true, we conclude that the diagram is sound with respect to the specification. In these SMT-LIB models, the return value of the SMT solver is *unknown*, because when using universal quantifiers, the solver cannot entirely derive conclusions about the universe of elements. For this reason, we only take into consideration the results for the specific rules, which constraint the reasoning domain to the elements that are members of the sets that describe the specification and the diagram.

The following model depicts the output of the solver when validating a diagram against a specification using our implementation of the validation rules for soundness.

SMTLib 6.12: Results for Soundness

```
((actors_validation true))
((operations_validation true))
((attributes_validation true))
((inheritances_validation true))
```

Notice that in this example, all variables are true, so we conclude that diagram is sound.

6.5 Equivalence

In this section, we will discuss the SMT-LIB implementation of the theory described in Section 5.4, related to model equivalence. The implementation makes use of the class diagram meta-model described in Section 6.2 to instantiate the components of the diagrams to be compared.

In our theory, we signalled semantic equivalence as the key to comparing two class diagrams. In our current implementation, semantic equivalence is implemented as the similarity in form and meaning between two words as described in the previous section. That is, two words are equivalent if they refer to the same base word and vary in number, spacing or capitalization, for example, *"word1"* and *"Words 1"* . The implementation of an SMT-LIB function that handles such equivalences is described in the model 6.8. With this function, we can formally compare two models.

Our theory separates two types of partial equivalence. *Left equivalence* occurs when all the elements of the first model have an equivalent element in the second one. Inversely, *right equivalence* is satisfied when all the elements of the second diagram have an equivalent in the first one. If both, left and right equivalences are satisfied, then we conclude that both models satisfy *total equivalence*.

The following model shows the function used to verify that for all the classes in the first diagram, for which there is an equivalent class in the second diagram.

SMTLib 6.13: Class Equivalence Checking

```
(define-fun check_classes ((s1 (Set Class)) (s2 (Set Class))) Bool
    (forall
        ((x Class))
        (=>
            (member x s1)
            (exists
                ((y Class))
                (and
```

```
            (member y s2)
            (is_syn (cls_nme x) (cls_nme y))
        )))))
```

This function, in particular, checks that for all the classes that belong to the set of classes of the first diagram, for which exists another class that belongs to the set of classes of the second diagram, such that the name of the first class is equivalent to the name of the second class.

The implementation of the model to check the equivalence of dictionaries, operations, attributes, relations and inheritances are found in Appendix B.4.

Similar to the validation task, the equivalence task depends on the result of individual operations, which are also stored in boolean variables. The definition of these variables are provided in Appendix **??**

In order to verify left and right equivalence, our python program dynamically generates two separate sub-models alternating the order of the diagrams. After these sub-models are run, we establish the type of equivalence existing, as well as identify which elements of the diagrams are not equivalent.

6.6 Class Model Extraction

At this point, TOMM provides the means to reason about class diagrams formally. In the case of T4TOMM, we have required the diagram to be described as a JSON file. However, this requirement imposes an extra effort on the potential users of the T4TOMM, for diagrams typically exist as image files. Intending to eliminate this additional effort, we have prototyped an algorithm capable of extracting the information contained in class diagrams, which generates the JSON description for the graphical representation of the diagrams.

The extraction of class model elements is divided into two tasks. The first one is to identify the graphical components of the diagrams in the image, commonly known in the image processing community as *image segmentation*. The second task requires *optical character recognition* (OCR) algorithms to extract the relevant information from every image segment. Figure 6.2 depicts the interaction between these tasks and will be described further next.

6.6.1 Image Segmentation

A class diagram is described as a composition of boxes and arrows, wherein every box represents a class, and every arrow represents a relation between classes. Employing image segmentation, we aim to divide the whole image into

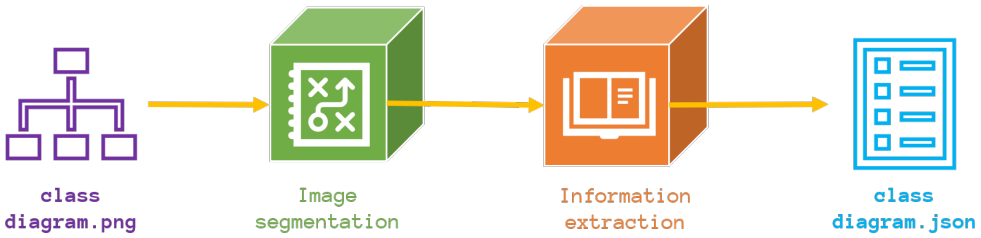


Figure 6.2: Class Diagrams Extraction Process

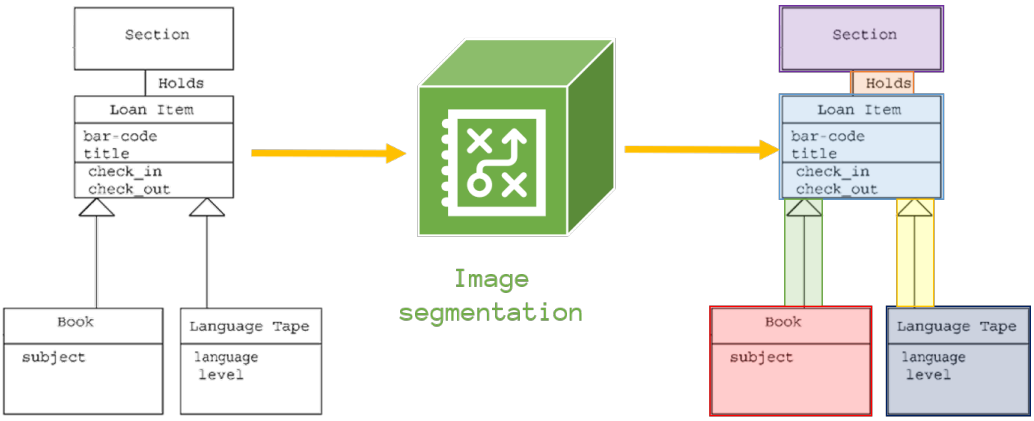


Figure 6.3: Class Diagram Segmentation

clusters of pixels that belong to either a class, an arrow, or the background. Figure 6.3 illustrates the goal of this task.

For instance, if a given class diagram is composed of 5 classes and 3 relations, then we have 9 possible labels (clusters) to be assigned to every pixel. In general, the number of labels is defined as:

$$l = c + r + 1 \tag{6.1}$$

Where c is the number of classes, r the number of relations, and 1 represents the background of the diagram or any other relevant information. Our current implementation of T4TOMM requires the user input to specify the number of classes and relations.

Class diagrams tend to be drawn differently depending on the tool used to create them. Variations include lines and background colours, line widths, connective shapes, fonts, amongst other features; this raises the need to perform pre-processing to normalize the images to be analysed. After the noise has been removed, images are segmented accordingly. Figure 6.4 depicts the stages of image segmentation, which are line recognition, boxes recognition

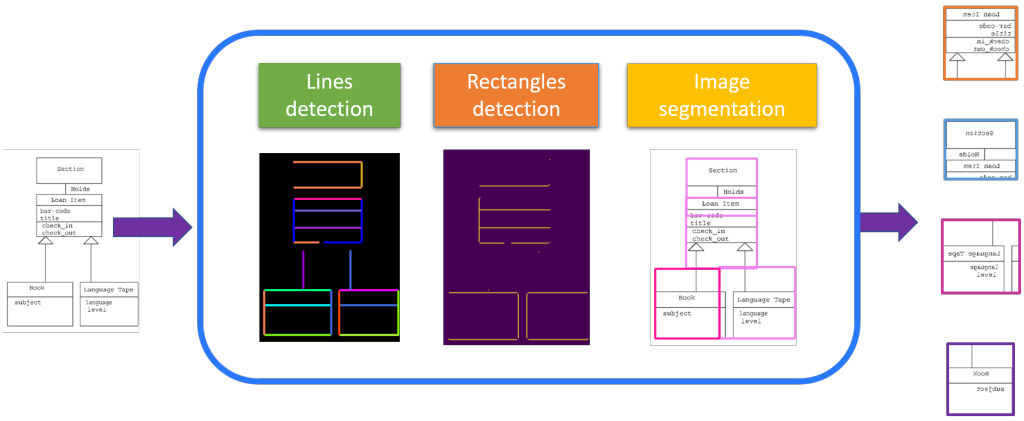


Figure 6.4: Segmentation Process

and finally image segmentation.

During the **lines detection** stage, only relevant lines are selected, in order to avoid the noise caused by the text within the boxes. It is assumed that input diagrams are given in colour (RGB), which adds extra complexity to the image. Hence, the first step of the pre-processing requires to transform the image to greyscale, which then is binarized to have only two possible values for each pixel, either black or white. The algorithm works better on images with a black background and white lines, so if required, the colours of each pixel are inverted so that non-background pixels are white. Binary images are then skeletonized so that only pixels representing the contours of the objects are kept. Finally the probabilistic Hough line transformation method[174, 283] is used to recognize valid lines. The resulting image is exported as a PNG file to be used in the following stage of the process. The different operations performed over the input image in this stage are shown in Figure 6.5.

Similarly to the lines detection stage, the **rectangles detection** stage requires to remove colour and to binarize the image as shown in Figure 6.6. The goal of this stage is to produce rectangular shapes based on the lines obtained in the previous stage, by building intricate skeletons from the relevant individual lines. These skeletons are exported as well and used in the next stage to identify classes.

Finally, in the *segmentation* stage, the rectangles formed before are used to cluster pixels that represent classes. In Figure 6.7 two approaches are compared: *labelling* based on neighbour pixels, and *k-means*[143, 202]. For small diagrams, the accuracy of both is almost the same, but on bigger

diagrams, k-means is more precise, for it is indicated exactly how many clusters are there. This difference is shown in Figure 6.8. Once all segments have been appropriately identified, every region is saved as an individual image that is used later to extract the information about the class.

6.6.2 Information Extraction

With the image segmented using k-means (or any other similar method), we feed every individual class to the text processor. It will retrieve the contents of the box line by line. This text must be processed in order to identify the type and name of the component. The following basic rules are followed to identify the individual elements of a class:

- **Class name** if the text belongs to the first or second line, and starts with a capital letter.
- **Attribute** every line starting from the second, that starts with lower case or a visibility symbol, and not containing parenthesis.
- **Operator** every line starting from the second, that starts with lower case or a visibility symbol and containing parenthesis.

We these rules, we can parse the contents of every class image as shown in Figure 6.9. Note that this implementation only works on classes that have been written following the standard UML notation.

The prototype implementation currently only works for a limited set of ideal class diagrams, particularly those that are very clear with classes that are sufficiently spaced. However, as a proof-of-concept, it certainly illustrates how our formal framework can be extended with the usage of image processing in order to reduce human effort in favour of better automation.

6.7 Summary

In this chapter, a proof-of-concept to support TOMM was proposed, namely T4TOMM. In order to make TOMM easily accessed, automatic formalization of ConSpec specifications is performed. Similarly, class models are partially obtained from the images of class diagrams, which are then formalized. SMT-LIB models generated from the processes described above are then and checked using SMT-Solvers to determine model validity and equivalence, together with the model inference from specifications. Some of the limitations of T4TOMM are briefly mentioned here and further explored in the following chapter.

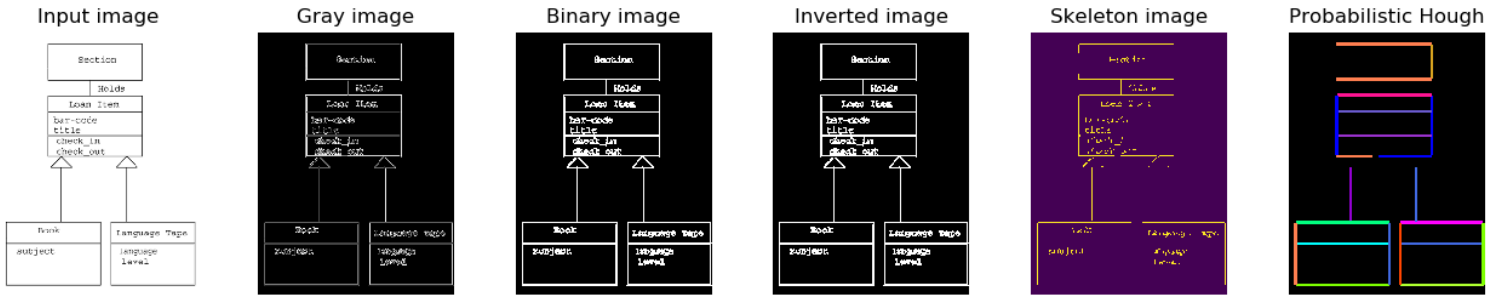


Figure 6.5: Lines detection process

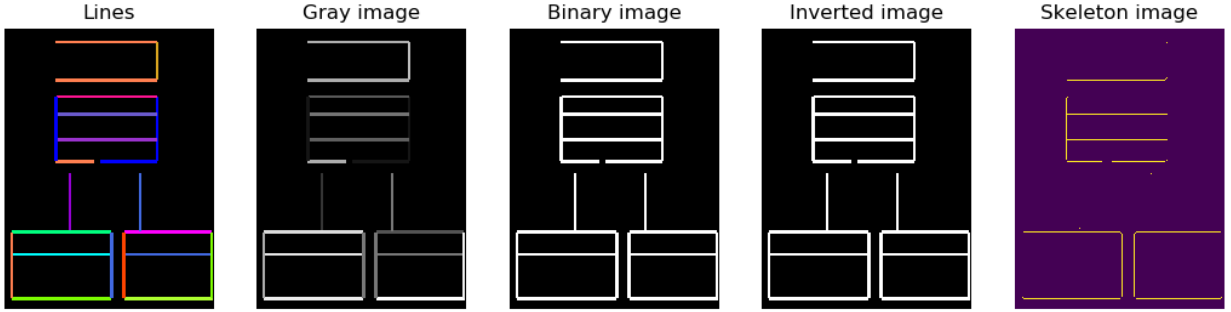


Figure 6.6: Rectangles detection process

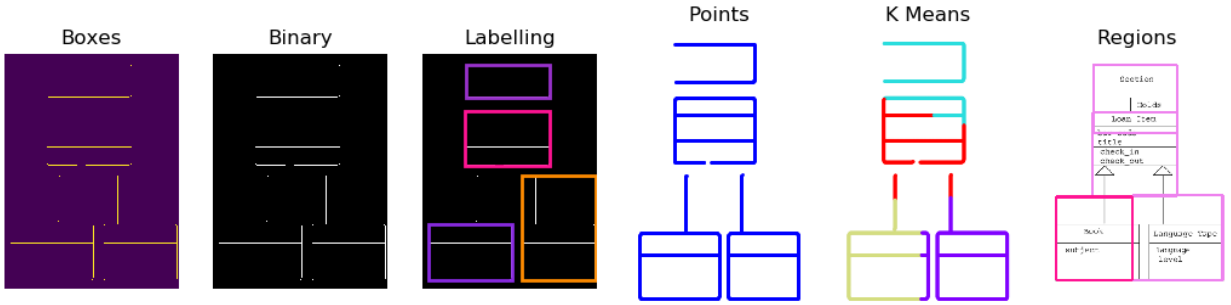


Figure 6.7: Segments detection process

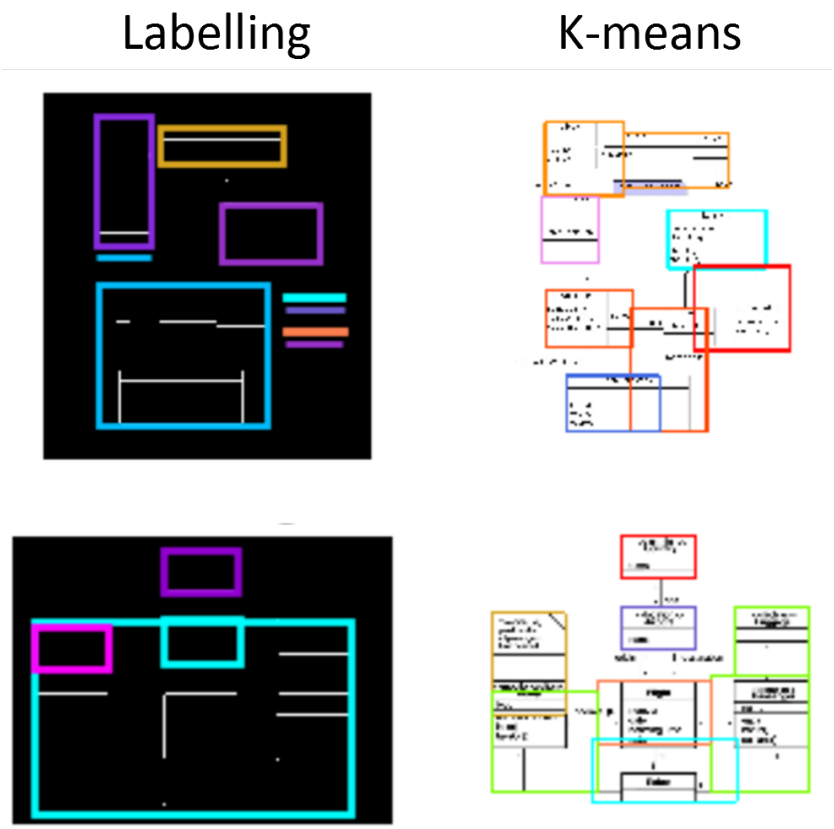


Figure 6.8: Comparison of labelling vs k-means

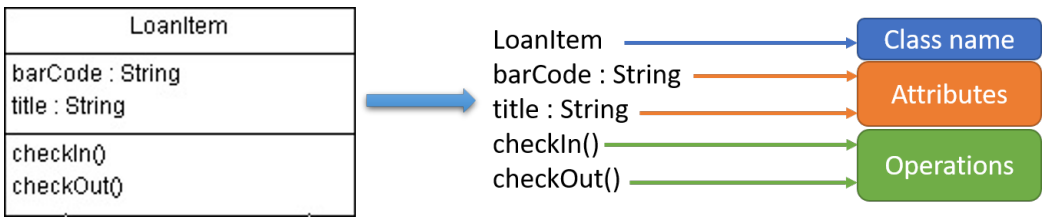


Figure 6.9: Class information extraction

Chapter 7

Evaluation

Through this chapter we describe the different aspects involved in the evaluation of our contributions, including the evaluation methodology, the cases to be evaluated, the actual evaluation and the results for each of the items enumerated in Section 1.4

First we evaluate our specification format for functional requirements (ConSpec + SpeCNL). This format is evaluated by manually translating existing requirements from different sources and domains into SpeCNL sentences within a ConSpec specification. For each of these requirements, we discuss the extension of the requirements that we were able to express in SpeCNL and the limitations that we came across. We highlight the results regarding functional requirements, and we discuss future work to improve and extend this specification format.

For model generation we refer to the requirements of the library system introduced in Chapter 2 restructured as a ConSpec specification, and we use T4TOMM to infer its corresponding class model. We then manually check that all the classes, attributes, operations and inheritances expected are present in the inferred model.

For model validation, we evaluate four individual cases. First, we check an invalid mode, to make sure our theory is capable of determining when a diagram is neither sound nor complete with respect to a given specification. Then we check a model that we know is sound, but not complete, that is, that it lacks some elements that exist in the specification. Then we check model completeness alone in a diagram that has all the elements of the specification, but also additional elements. We finally check a valid model, that is, a model that is sound and complete. All these checks are done using our proof-of-concept and manually modified diagrams that satisfy each case.

For model comparison, we follow a similar approach to the one of model validation. We use T4TOMM to check for two models that are different, then

we check for left, right and total equivalence of manually modified diagrams that satisfy these scenarios.

The evaluation of these theories includes first a demonstration of their manual application, and then we proceed to evaluate the specific cases using T4TOMM. All these theories are evaluated with respect to the current capabilities of TOMM, that is, reasoning about classes, attributes, operations and inheritances. Additional elements such as OCL constraints or associations are not evaluated. The obtained results are manually compared against the expected ones, and then, they are discussed individually in each subsection. The threats for validity are discussed towards the end of the corresponding section.

In addition, we evaluate T4TOMM for model extraction, which is a secondary contribution. We extract class models from existing class diagrams from different sources. For each existing class diagram, we manually count the number of classes, attributes and operations, and then we compare those results with the ones obtained from our proof-of-concept.

7.1 ConSpec and SpeCNL

In Chapter 4 we presented our document structure, ConSpec, and our controlled natural language SpeCNL used to specify functional requirements. In this section, we discuss the process followed to evaluate their capabilities and limitations in comparison with requirements documents written in English, with no pre-defined structure. We present here a selection of requirements from a collection of publicly available requirements documents, and we analyse the process required to rewrite these original requirements into ConSpec specifications.

7.1.1 Evaluation Methodology

The evaluation of ConSpec and SpeCNL was done by comparing existing requirements documents with their corresponding ConSpec representation. First, we investigated and identified sources for requirements documents in English. Then we curated a subset of requirements to be used in our evaluation, by defining a representative combination of features, which are: source, length, and domain. The *source* feature represents the origin of the requirements, which we classified into research publications, academic materials and technical documentation for real systems. The *length* of each original document was determined by the number of words if it does not exceed 1000 words or the number of pages in the case it does exceed 1000

words. Finally, the *domain* feature refers to the business sector related to the system being specified, such as finance or transportation.

Using boundary and combinatorial analysis[56], we chose the shortest and longest requirements from each source, and then we made sure to include samples for several domains, the resulting set is shown in Table 7.1. These requirements were then *manually* translated into SpeCNL sentences within their corresponding ConSpec document. The translation process allowed us to identify several sentences that could not be represented in our proposed format. We then analysed and discussed the structure and meaning of these sentences, together with the limitations of our specification format that prevented us from capturing them properly. These activities are further discussed in Section 7.1.2 for each original requirements document.

7.1.2 Evaluation Cases

While class diagrams can be obtained from different sources, such as examples in books, assignments from educational courses, and peer-reviewed publications, requirements specifications or requirements documents are not equally available. However, we managed to find an attempt to collect requirements documents aimed to benchmark tools that generate UML diagrams from natural language requirements[289].

This collection goes by the name NLRP-Bench and is available as an online repository of classified requirements and diagrams. It contains several requirements documents from different sources in German and English, together with a smaller set of class and activity diagrams, associated with some of these requirements.

After inspecting this collection, and following our evaluation methodology proposed in Section 7.1.1, we chose a subset of requirements to be used in our evaluation, which is shown in Table 7.1. These requirements define the cases discussed in this section.

7.1.2.1 Ships Description

The first requirements to be analysed are named *Ships Description*[1], which describe a minimal version of the components of ships. This example has been used in exam questions and has been developed by Prof. Walter F. Tichy from the Karlsruher Institut für Technologie (KIT), who is also one of the authors of NLRP-Bench. The original text is shown Text 7.1.

[1] `http://nlrp.ipd.kit.edu/index.php/Ships`

Table 7.1: Evaluation Cases for ConSpec and SpeCNL

Requirements	Source	Length	Domain
Ships Description	Academic	49 words	Transportation
Trains Description	Academic	78 words	Transportation
ATM Simulation	Academic	750 words	Banking
ACME Library	Academic	16 pages	Business
Simplified Library	Research	217 words	Business
Steam-Boiler	Research	10 pages	Hardware
Laws of Chess	Research	18 pages	Gaming
Whois Protocol	Technical	100 words	Protocol
Light Control System	Technical	13 pages	Hardware

Ships are either passenger ships or container ships. A passenger ship may have one or more restaurants, which each are equipped with a certain number of seats. Each ship has at least one diesel engine. A diesel engine comprises several cylinders. Each cylinder has valves and a combustion chamber.

Text 7.1: Ships Description Requirements

Table 7.2 shows the translation of the original sentences into SpeCNL. In the original text, it is noted that most of the verbs, coloured in blue, refer to the properties of the ships. Thus they can be rewritten as *structural sentences* defined in Section 4.1.3, that is the case of sentences *s2*, *s4* and *s5*. This is possible because the expressions *"are equiped with"* and *"compromises"* are equivalent of the verb *"to have"* for this particular example. Sentence *s1* is shows the usage of *type sentence*, described in Section 4.1.3, which generate sentences *s1.1* and *s1.2*. Finally, *s3* is translated into a cardinality sentence to indicate the minimum number of cylinders for each engine. Notice that though *s2* also mentions the cardinality of the number of restaurants in a passenger ship, the modal *may* supersedes the role of the cardinality.

Our ConSpec format expects every requirement to be expressed in terms of actions, but these requirements do not describe any action to take place within the system, which implies that no clauses can be written. In order to deal with this problem, we introduced a *neutral action* to indicate the existence of something, regardless of whether there are actions to be taken or not. The neutral action corresponds to sentence *"somethign exists"* , which is supported by SpeCNL; however, its usage would indicate that in the class diagram related this specification (whether generated or validated), there

Table 7.2: Translation of Ships Description requirements into SpeCNL sentences

Original sentence	SpecCNL translation	Element
s1 Ships are either passenger ships or container ships	s1.1 Passenger-ships are ships	Type sentence
	s1.2 Container-ships are ships	Type sentence
s2 A passenger-ship may have one or more restaurants, which each are equipped with a certain number of seats	s2.1 Passenger-ships may have restaurants	Structural sentence
	s2.2 Restaurants have seats	Structural sentence
s3 Each ship has at least one diesel engine	s3.1 Ships have at least one diesel-engine	Cardinality sentence
s4 A diesel engine comprises several cylinders	s4.1 Diesel-engines have cylinders	Structural sentence
s5 Each cylinder has valves and a combustion chamber	s5.1 Cylinders have valves and combustion chambers	Structural sentence

must be a class named *"Somethign"* with a method named *"exists"* . This peculiarity has to be taken into consideration in future versions of ConSpec, and specific ways of dealing with the neutral action will have to be further defined in TOMM as well.

For this particular example, the neutral action corresponds to the sentence *"ships exist"* , where ships are the actors and exist is the action for the only clause of the contract in which the properties of the ships will be specified. Because the neutral action is a special case, given that it is not executed, the components of the clause are expressed within the action conditions section, as shown in ConSpec 7.1.

ConSpec 7.1: Ships Description ConSpec

```
- C1:
    Action: exist
    Actors:
        - Ships
    Action conditions:
        - Passenger-ships are ships
        - Container-ships are ships
        - Passenger-ships may have restaurants
        - Restaurants have seats
        - Ships have at least 1 diesel-engine
        - Diesel-engines have cylinders
        - Cylinders have valves and combustion-chambers
```

From this example, we concluded that SpeCNL is sufficient to express these

requirements with the sentences provided, but ConSpec requires to handle the existential neutral action. This observation also opens the question about the existence of other neutral actions that should be considered; however, this question will be left for future work, as discussed in Section 8.2.

7.1.2.2 Trains Description

The next example is the *trains description*[2] system, which is another exam question from Tichy. Unlike the example of the ships, this example does contain some functionality to be specified, and its structures are more complex. The full requirements are described in Text 7.2.

> A freight train is a train whose carriages are only freight cars. The cars of a passenger train are at least one passenger coach and a maximum of one dining car. Each train has one or two locomotives. Coaches are composed of up to one big room and one or more compartments. Each coach has air conditioning which can be turned on or off. Each air conditioner can only be operated up to a specified maximum ambient temperature.

Text 7.2: Trains Description Requirements

Table 7.3 shows the translation of these requirements into SpeCNL. This example consists of more cardinality and type sentences, together with actions and comparison sentences. Sentence *s6* compares the temperature of the air conditioner with the maximum ambient temperature, which is another variable. However, comparison sentences in SpeCNL do not allow comparison with variables. Instead, an actual numeric value must be provided; for this reason, we had to assume a value for the maximum ambient temperature. We also had to assume that cars, carriages, coaches are all equivalent in some way and that they all are carriages with specific sub-types.

With this translation, we observed the possible limitation in comparison sentences concerning the usage of variables. However, although this feature reduces the flexibility of SpeCNL, it also assures that the corresponding values are always provided, and in fact, it forces the users to think of these values when writing requirements.

The specification showed in ConSpec 7.2 corresponds to these requirements. Notice that alike ships specification; we again need to write a *neutral action* for the existence of trains an all of their components. An alternative would have been to write these sentences as preconditions for the switch-on action. However, that would require us to either include them also in the switch-off action, which is redundant or to establish a dependency between switch-on and switch-off only to indicate the existence of the components of the system,

[2] `http://nlrp.ipd.kit.edu/index.php/Trains`

Table 7.3: Translation of Trains Description requirements into SpeCNL sentences

Original sentence	SpecCNL translation	Element
s1 A freight train is a train whose carriages are only freight cars	s1.1 Trains have carriages	Structural sentence
	s1.2 Freight-trains are trains	Type sentence
	s1.3 Freight-cars are carriages	Type sentence
	s1.4 Freight-trains have freight-cars	Structural sentence
s2 The cars of a passenger train are at least one passenger coach and a maximum one diner coach	s2.1 Passenger-trains are trains	Type sentence
	s2.2 Passenger-coaches are carriages	Type sentence
	s2.3 Dining-cars are carriages	Type sentence
	s2.4 Passenger-trains must have at least 1 passenger-coach	Cardinality sentence
	s2.5 Passenger-trains must have at most 1 dining area	Cardinality sentence
s3 Each trains has one or two locomotives	s3.1 Trains must have at least one locomotive	Cardinality sentence
	s2.2 Trains must have at most two locomotives	Cardinality sentence
s4 Coaches are composed of up to two big room and one or more compartments	s4.1 Coaches can have one big-room	Cardinality sentence
	s4.2 Coaches have at least 1 compartment	Cardinality sentence
s5 Each coach has air-conditioning which can be switched on or off	s5.1 Coaches have air-conditioning	Structural sentence
	s5.2 Switch-on air-conditioning	Action
	s5.2 Switch-off air-conditioning	Action
s6 Each air conditioner can only be operated up to a specified ambient ambient temperature	s6.1 Air-conditioning's temperature must be less than or equal to 32	Comparison sentence

which may lead to unexpected behaviour. For these reasons, it was determined that having a neutral action to be reused as a dependency for switch-on and switch-off actions was the best alternative. Though situations like this one do not affect the generation and validation of class models, better handling procedures are required when dealing with other types of models, such as sequence diagrams, or OCL constraints. However, these models are not part of the scope of our current research and will be part of future work.

ConSpec 7.2: Trains Description ConSpec

```
- C1:
    Action: exist
    Actors:
        - Trains
    Action conditions:
        - Trains have carriages
        - Freight-trains are trains
        - Freight-cars are carriages
        - Freight-trains have freight-carriages
        - Passenger-trains are trains
        - Passenger-coaches are carriages
        - Dinning-cars are carriages
        - Passenger-trains must have at least 1 passenger-coach
        - Passenger-trains must have at most 1 dinning-car
        - Trains must have at least 1 locomotive
        - Trains must have at most 2 locomotives
        - Coaches can have 1 big-room
        - Coaches have at least 1 compartment
        - Coaches have air-conditioning
        - Air-conditioning's temperature must be less or equal than 32
- C2:
    Action: Switch-on
    Actors:
        - Air-conditioning
    Postconditions:
        - Air-conditioning's temperature must be less or equal than 32
    Dependencies:
        - C1
- C3:
    Action: Switch-off
    Actors:
        - Air-conditioning
    Dependencies:
        - C2
```

7.1.2.3 ATM Simulation

The length of the requirements document for the *ATM Simulation*[3] system and the *ACME University Library Information System*[4] makes it unreasonable

[3] http://nlrp.ipd.kit.edu/index.php/ATM_Simulation

[4] http://nlrp.ipd.kit.edu/index.php/ERS_ACME_-_University_Library_Information_System

to add them within this dissertation. However, they are available for reference in their corresponding URLs found in the footnotes. We present and discuss now only the elements that are relevant to this evaluation.

The ATM simulation is a peculiar case, for it defines the interaction of hardware and software. For this reason, we do not expect SpeCNL and ConSpec to be prepared to deal with the entire set of requirements. Nonetheless, this exercise helps to provide an insight into their potential to deal with more complex systems.

The original requirements document is verbose, thus translating sentence by sentence is not the best approach. Instead, the specification was written from scratch with a simplified interpretation of the requirements. The specification segment showed in ConSpec 7.3 demonstrates the support of preconditions, conditional sentences, consequences, and dependencies.

ConSpec 7.3: ATM Simulation ConSpec

```
- C1:
    Action: exist
    Actors:
        - ATM
    Action conditions:
        - ATMs have card-reader
        - ATMs have keyboard
        - ATMs have display
- C2:
    Action: exist
    Actors:
        - Customer
- C3:
    Action: read card
    Actors:
        - card-readers
- C4:
    Action: insert PIN
    Actors:
        - Customer
    Dependencies:
        - C3
- C5:
    Action: validate PIN
    Actors:
        - Bank
    Preconditions:
        - if attempts is >= 3 then retain card
    Dependencies:
        - C4
        - C6
    Consequences:
        - re-enter PIN
- C6:
    Action: communicate bank
    Actors:
        - ATM
    Preconditions:
        - ATMs have communication link
```

```
- C7:
    Action: withdraw amount
    Actors:
        - Customer
    Preconditions:
        - amount must be valid
    Dependencies:
        - C5
        - C6
- C8:
    Action: get balance
    Actors:
        - Customer
    Dependencies:
        - C5
```

When writing these specifications, some issues arose, including the loss of certain information that cannot be expressed in SpeCNL. One example is the expression *"multiples of $20.00"* as part of the precondition for cash withdrawal. The *comparison sentences* in SpeCNL are, once again, not prepared to deal with comparisons other than numeric in adjectives. Hence, advance mathematical comparisons like this one are considered as future work.

7.1.2.4 ACME Library

The requirements for the *ACME University Library Information System* are written in an extensive document that conforms to the guidelines of the IEEE discussed in Section 3.2.1. We pay particular attention to the section of functional requirements which lists 32 requirements. In order to demonstrated the derivation of its specification, some of the requirements are shown in Text 7.3, and their representation is shown in ConSpec 7.4.

- Req(01) Users shall be able to reserve books, irrespective of whether or not the requested book is on loan.
- Req(02) Users shall not be able to reserve reference works.
- Req(04) The reservation shall be valid for any copy of a particular book. For example, if there are five copies of Don Quixote, any one of these copies shall be considered to be reserved.
- Req(13) Library users shall be able to borrow books.
- Req(14) The loan period shall be 7 days for students and administrative and support staff and 15 days for faculty.
- Req(25) Library users shall have to return the books before the loan period expires.
- Req(26) If a book is returned late, the system shall suspend the user for two days for every day the book is overdue.

Text 7.3: ACME Library Requirements

ConSpec 7.4: ACME Library ConSpec

```
- C1:
    Action: reserve book
    Actors:
        - User
    Preconditions:
        - Book has copies
    Action conditions:
        - if book is not reference-work then reserve book
        - if book is loaned then reserve book
        - if book is not loaned then reserve book
        - if book has copies then reserve copies
- C2:
    Action: borrow book
    Actors:
        - User
    Preconditions:
        - loan has period
    Action conditions:
        - if user is not faculty then loan's period is 7
        - if user is faculty then loan's period is 15
- C3:
    Action: return book
    Actors:
        - User
    Action conditions:
        - if loan's period is expired then suspend user
```

In this particular example, some information is lost due to the structure of the *conditional sentences*. The requirement *Req(26)* estates that the suspension period must be two times the number of days overdue. However, this statement cannot be expressed as a consequence of the form `CONSEQUENCE -> VBB ENTITY` . A possible solution would be to express *"user suspend"* as an additional clause and then break down the constraints in simpler ones. However, the current structure of ConSpec does not keep any relation between clause actions and consequences in conditional sentences. This discovery points us out to an important aspect that ought to be considered in the next version of our specification formats.

In the next section, we discuss the particularities found when SpeCNL is being used to specify the requirements found in publications.

7.1.2.5 Simplified Library

The *library* example has been published with the work of Callan[62], Harmain[142], Kim[171], and Bajwa[26]. This text was also used in Section 4.3 as an example of the refinement process used to generate ConSpec specifications. The complete text and its specification are found in Appendix A.1 and Appendix section A.2.

7.1.2.6 Steam Boiler

The *Steam-Boiler Control Specification Problem* [5] is a document that describes a system to control the level of water in a steam-boiler. It was initially taken from the International Software Safety Symposium organised by the Institute for Risk Research, where it was used as a competition problem. However, the current requirements publicly available are an adaptation for a particular boiler system published by Abrial[7].

The whole system is integrated by the steam-boiler, the water level measurement device, the pumps, the control device and the steam measurement device. The main program's behaviour, the subject of our specification, has five operation modes: initialisation, normal, degraded, rescue and emergency stop. In ConSpec 7.5, we illustrate the specification of the normal mode.

ConSpec 7.5: Steam Boiler ConSpec

```
- C1:
    Action: operate normal-mode
    Actors:
        - Boiler
    Action conditions:
        - If water-level is equal to 15 then activate pumps
        - If water-level is more than 95 then deactivate pumps
        - If water-level-measuring-unit is failing then operate rescue-mode
        - If water-level is equal to 10 then operate emergency-stop
        - If water-level is equal to 100 then operate emergency-stop
```

When translating these requirements, similar assumptions to the train specification had to be made. There is a need to compare water levels with normal and maximum values, which are not explicitly indicated in the requirements. Hence, we indicated 15 and 95 for normal values, and 10 and 100 for extreme values. This assumption not only solves the problem partially but also helped to discover that the current specification for comparators does not allow to express comparison with ranges, for example, *"x is between a and b"* . It is also interesting to note the relation of these requirements with state diagrams, which are not currently supported by TOMM. These extensions will be considered in a future version of SpeCNL.

7.1.2.7 Laws of Chess

FIDE laws of Chess[6] specification has been published in the handbook of the World Chess Federation. It contains all the rules and movements allows in chess, described in a structured and illustrated manner. The document

[5] `http://nlrp.ipd.kit.edu/index.php/Steam_Boiler`

[6] `http://nlrp.ipd.kit.edu/index.php/FIDE_laws_of_chess`

is divided into articles describing the goals of the game, the description of the pieces, their starting configuration, the moves allowed for each piece, the sequence of actions and the termination conditions, together with some additional particularities. Unlike previous cases, this document does not describe the requirements of a system on itself, but the rules of a game, which are specified in ConSpec 7.6.

ConSpec 7.6: Laws of Chess ConSpec

```
- C1:
    Action: set-up
    Actors:
        - Game
    Preconditions:
        - Kings are pieces
        - Queens are pieces
        - Rooks are pieces
- C2:
    Action: check move
    Actors:
        - Piece
- C3:
    Action: move
    Actors:
        - Piece
- C4:
    Action: start
    Actors:
        - Game
    Action conditions:
        - if piece is moved then check move
    Postconditions:
        - check end game
    Dependencies:
        - C1
```

In this example, only kings, queens and rooks were specified as pieces, but it had to be done in separate statements; this can be improved by adding several subtypes into one single statement like *"Kings, Queens and Rooks are Pieces"* . Besides, it was noticed that the sequence of actions for a game could be better described through referencing other clauses within the pre and postconditions.

7.1.2.8 Whois Protocol

Academic and published requirements often tend to be related to common scenarios, things that one can come across daily, and whose elements result familiar to most of the people. In contrast, the real systems explored to be explored now require knowledge of the particular domain they belong to; they use specific terminology and are typically defined in terms of several components.

The *Whois Protocol*[7] is characterised by the sequence of steps that must be followed. The description text is rather short and is shown in Text 7.4.

> A WHOIS server listens on TCP port 43 for requests from WHOIS clients. The WHOIS client makes a text request to the WHOIS server, then the WHOIS server replies with text content. All requests are terminated with ASCII CR and then ASCII LF. The response might contain more than one line of text, so the presence of ASCII CR or ASCII LF characters does not indicate the end of the response. The WHOIS server closes its connection as soon as the output is finished. The closed TCP connection is the indication to the client that the response has been received.

Text 7.4: Whois Protocol Requirements

The steps for the protocol are properly described in the ConSpec 7.7. However, the termination for the requests could not be specified due to the limitations of comparison sentences. *C5* expresses the sequence to be followed in the form of *conditional sentences* and this sequence is again established through the dependencies in *C3* and *C4*. This duplication of information can be removed by allowing ConSpec to use clauses within the conditions of an action.

ConSpec 7.7: Whois Protocol ConSpec

```
- C1:
    Action: listen request
    Actors:
        - Server
    Preconditions:
        - Server has ports
        - Listening-port is 43
- C2:
    Action: make request
    Actors:
        - Client
    Action conditions:
        - Send text
    Postconditions:
        - Receive response
- C3:
    Action: reply request
    Actors:
        - Server
    Action conditions:
        - send text
    Postconditions:
        - Close connection
    Dependencies:
        - C1
        - C2
- C4:
    Action: receive response
    Actors:
```

[7] http://nlrp.ipd.kit.edu/index.php/Whois_Protocol

```
        - Client
    Dependencies:
        - C3
- C5:
    Action: execute
    Actors:
        Protocol
    Action conditions:
        - if protocol started then make request
        - if server is listening then reply request
        - if request replied then receive response
```

7.1.2.9 Light Control System

Next example is the *RE UTS Light Control System*[8]. The purpose of the system is to update the control of the lights of the 4th floor, in building 32 of the University of Kaiserslautern, Germany. The document describes the schematics for the sensors and actuators on the floor, together with a description for every room. Additionally, the functional needs are listed, which are used to generate the specification for our benchmarking.

The specification showed in ConSpec 7.8 illustrates some of the functionality specified. Though this example describes a real-life system, there are only a number of functionalities that can be specified using SpeCNL. The reason is that this document describes mainly the features, locations and behaviour of the physical components (hardware).

ConSpec 7.8: Light Control System ConSpec

```
- C1:
    Action: activate safe-illumination
    Actors:
        - Light
- C2:
    Action: change scene
    Actors:
        - Light
- C2:
    Action: set default-scene
    Actors:
        - Light
- C2:
    Action: set transition-time
    Actors:
        - Light
- C3:
    Action: change light
    Actors:
        - Room
    Preconditions:
        - Rooms have lights
    Action conditions:
```

[8] http://nlrp.ipd.kit.edu/index.php/RE_UTS_Light_Control_System

```
- if room is recently-occupied then activate safe-illumination
- if light-scene is changed then keep scene
- if room is unoccupied then reset lights
```

To this point requirement for academic, published and real systems have been presented together with their corresponding ConSpec specification. The summary of our findings is presented in Section 7.1.3.

7.1.3 Summary of Evaluation for ConSpec and SpecCNL

Through the manual translation of existing requirements into ConSpec specifications, it was possible to evaluate the strengths and limitations of SpeCNL and ConSpec. The limitations observed and areas of improvement are now discussed.

7.1.3.1 Areas of improvement for SpeCNL

Regarding the SpeCNL language, areas of opportunity were identified. One of them is *comparison sentences*, which allow only single numeric values and adjectives, leading to loss of certain information, and hence the need to extend them to support ranges, and variables. It was also observed that *type sentences* could be simplified, including several subtypes in one single statement. Another enhancement has to be done in *conditional sentences* to support more complex structures, for example, alternative conditions and simultaneous conditions.

A more general aspect to be considered has to do not with the language itself, but with the way it is used. Currently, it is needed to identify *manually* equivalent concepts within the domain of each specification. However, further research on natural language processing and discourse analysis can help to automate this process[25, 82, 267, 305, 306, 311]. Besides, we envision that future developments of deep learning in machine translation[23, 69, 284] will allow us to translate requirements documents into ConSpec specifications automatically.

7.1.3.2 Areas of improvement for ConSpec

Concerning ConSpec, we observed the constant need for a *neutral existential action* to specify structural components of a system with no actions, which could potentially lead to entity-relationship diagrams[67, 287]. This observation also raises concerns about the possible existence of other neutral

actions, which will have to be investigated further. Though SpeCNL does allow to express neutral existential actions, ConSpec does not provide a proper definition or semantics for it, and TOMM is not equipped to deal with these actions.

Another area of improvement for ConSpec is the relationship between clauses. In particular, we observed the need to relate the conditions of one clause to other clauses. This extension will be crucial for TOMM to support other than class diagrams, in particular, sequence and state diagrams.

Finally, we propose the inclusion of an open field within each clause, that allows documenting in common English aspects that cannot be expressed in SpeCNL. This feature will allow minimising loss of information caused due to unseen limitations of ConSpec and SpeCNL.

7.1.3.3 Conclusion

At the beginning of Chapter 4 the desired properties for requirements specifications were described. Now we provide a summary of their presence or absence in our proposed requirements specification.

The precise definition of the parts for every clause (see Section 4.2) and their application is shown in this evaluation, allows us to argue that ConSpec specifications are indeed *structured*. In this evaluation, we also demonstrated the use of SpeCNL sentences to describe existing requirements. The simplicity of these rules demonstrates *clarity* as a feature present in ConSpec specifications.

We also observed that ConSpec specifications can be *redundant* and *ambiguous*. The former is needed in order to enable a relationship with class diagrams, while the later will have to be dealt with in future work. Also, *verifiable* and *correctness* features are not an explicit part of ConSpec specification, however, they that can be achieved in combination with TOMM.

In spite of the areas of improvement identified, which will be addressed in the future developments, through this exercise we have demonstrated that SpeCNL and ConSpec are capable of capturing various existing requirements from different sources, with different lengths and for different domains, as it was defined in our evaluation methodology (Section 7.1.1). We now proceed to discuss the evaluation of TOMM in Section 7.2

7.2 TOMM and T4TOMM

In this section, we describe the methodology used to evaluate TOMM as a formal framework to generate, validate and infer class models. Also, we define

a series of cases to be used in the evaluation of each of these activities. The results obtained from the evaluation are summarised in Section 7.2.6.

7.2.1 Evaluation Methodology

TOMM is a formal framework that contains theories to conduct specific reasoning tasks over class models. We evaluate these theories over properly defined cases that represent the expected outputs for each theory, which will be compared against the actual outputs of reasoning using TOMM. In order to assist in the evaluation of these scenarios, we make use of T4TOMM, which, as described in Chapter 6, automates the reasoning activities when the appropriate model representations are provided. Both, manual application of the reasoning rules, and automated reasoning using T4TOMM are included in this evaluation.

Table 7.4 shows the different cases to be evaluated for each reasoning activity. Specific needs for each evaluation are described in the corresponding case section.

Table 7.4: Evaluation cases for TOMM and T4TOMM

Activity	Case
Generation	- Inferring model manually - Inferring model with T4TOMM
Validation	- Manual validation - Checking invalid model using T4TOMM - Checking sound model using T4TOMM - Checking complete model using T4TOMM - Checking valid model using T4TOMM
Comparison	- Manual comparison - Comparing not equivalent models using T4TOMM - Comparing models with left equivalence using T4TOMM - Comparing models with right equivalence using T4TOMM - Comparing equivalent models using T4TOMM

These cases are organised in such a way that resources can be used progressively. For example, a class model inferred from a given specification can also be used to check for validity, and to be compared with other existing models.

7.2.2 Evaluation of Model Generation

In this section, we evaluate the generation of class models using the inference rules defined in TOMM. First, we demonstrated the manual applications of

the rules, and then we execute the inference rules encoded in T4TOMM.

7.2.2.1 Inferring model manually

For this case, it is required to have a ConSpec specification that will generate a model for its corresponding class diagram. The example of the library proposed by Callan[62] will be used; this is due to the number of diagrams available and its tendency to be used in evaluations within this area of research.

To discuss the evaluation of our framework the segment of the original requirements shown in Text 7.5 will be used.

> A library issues loan items to customers. Each customer is known as a member and is issued a membership card that shows a unique member number. Along with the membership number, other details on a customer must be kept such as a name, address, and date of birth. A loan item is uniquely identified by a bar code. There are two types of loan items, language tapes, and books. A language tape has a title language (e.g. French), and level (e.g. beginner. A book has a title, and author(s).

Text 7.5: Segment of the requirements for the Library system described by Callan[62]

In order to infer a class diagram, the ConSpec specification and its corresponding formalisation are required. To present only the relevant information, the ϵ elements of the predicates through this section have been omitted, this, however, does not represent a change in the original formalisation.

The corresponding ConSpec specification shown in ConSpec 7.9 was manually generated by us, following the steps exemplified in Section 7.1.2.1. The predicates showed in Predicates 7.1 were also generated manually and will be used to demonstrate the manual application of the inference rules.

ConSpec 7.9: Specification of the Library system proposed by Callan[62]

```
Title: Callan Library System
Version: 1.0

Clauses:
      - C1:
            Action: exists
            Actors:
                  - Customers
                  - Members
                  - Books
                  - Language-tapes
            Activity Conditions:
                  - Customers are members
                  - Members have member-id
                  - Members have name
```

```
                - Members have address
                - Members have date-of-birth
    - C2:
          Action: loans loan-item
          Actors:
                - Library
          Preconditions:
                - Items must have bar-code
                - Books are items
                - Books must have title
                - Books must have author
                - Tapes are items
                - Tapes must have language
                - tapes must have level
    - C3:
          Action: issues membership-card
          Actors:
                - Library
          Action conditions:
                - if id is valid then generate id
          Dependencies:
                - C2
```

Predicates 7.1: Predicates for the ConSpec specification of the Library system shown in ConSpec 7.9

```
S_1 = {
  ACTOR(Library)
  ACTOR(Customer)
  ACTOR(Member)
  ACTOR(Loan-items)
  ACTOR(Language-tapes)
  ACTOR(Books)
  ACTIVITY(loans. loan-item)
  ACTIVITY(issues. membership-card)
  TYPE_SENTENCE(books, are, loan-items)
  TYPE_SENTENCE(language-tapes,
    are, loan-items)
  TYPE_SENTENCE(Customers, are, Members)
  STRUCTURAL_SENTENCE(Books, have, title)
  STRUCTURAL_SENTENCE(Books, have, author)
  STRUCTURAL_SENTENCE(Language-tapes,
    have, title-language)
  STRUCTURAL_SENTENCE(Language-tapes,
    have, level)
  STRUCTURAL_SENTENCE(Members,
    have, member-id)
  STRUCTURAL_SENTENCE(Members,
    have, name)
  STRUCTURAL_SENTENCE(Members,
    have, address)
  STRUCTURAL_SENTENCE(Members,
    have, date-of-birth)
}
```

The model M_v shown in Predicates 7.2 was generated manually applying the inference rules proposed in Section 5.2.1. Each element of the model is generated by applying its corresponding rule, for example, the following applications allow to infer $CLS(Library)$ and $INH(Members, Customers)$

$$\frac{\text{ACTOR(Library)}}{\text{CLS(Library, classifier), TYPE(Library)}}$$

$$\frac{\text{TYPE\_SENTENCE(Customers, Members)}}{\text{INH(Members,Customers)}}$$

Predicates 7.2: Predicates for the class model manually inferred from Predicates 7.1

```
M_v = {
  CLS(Library)
  CLS(Customer)
  CLS(Member)
  CLS(Loan-items)
  CLS(Language-tapes)
  CLS(Books)
  TYPE(Library)
  TYPE(Customer)
  TYPE(Member)
  TYPE(Loan-items)
  TYPE(Language-tapes)
  TYPE(Books)
  OPR(Library, loans, {Loan-item})
  OPR(Library, issues. {Membership-card})
  INH(Loan-items, Books)
  INH(Loan-items, Language-tapes)
  INH(Members, Customers)
  ATR(Books, title)
  ATR(Books, author)
  ATR(Language-tapes, title-language)
  ATR(Language-tapes, level)
  ATR(Members, member-id)
  ATR(Members, name)
  ATR(Members, address)
  ATR(Members, date-of-birth)
}
```

The manual application of all the inference rules is rather verbose; hence it is not included in here. However, an important observation is that for classes existing in other predicates, they always have to be declared within the actors section of a class. This can be avoided by extending the other rules (inheritance, attribute and operation) to infer the classes they refer to automatically.

7.2.2.2 Inferring model with T4TOMM

The current implementation of T4TOMM was run from the terminal, with the file containing the ConSpec specification above. Before inferring the class model, the specification shown in ConSpec 7.9 was automatically translated by T4TOMM in an SMT-LIB model that includes the corresponding SMT-LIB representation of the inference rules. SMTLib 7.1 contains a snippet of the encoding of the inference rules, together with the SMT-LIB formalization

of the ConSpec specification, the full model is provided in Appendix C.1 as SMTLib C.1. This model is processed by CVC4 and infers the corresponding SMT-LIB model representing the inferred class model, which is shown in SMTLib 7.2.

SMTLib 7.1: Segment of SMT-LIB model to infer class model from ConSpec 7.9

```
;---------------------------------
; Specification elements
;---------------------------------

(declare-datatypes ((Structural_S 0))
    (((STRUCT
        (entity String)
        (modal String)
        (have String)
        (property String)
        ))))


(declare-datatypes ( (Attribute 0) ) ( ((ATR
                                        (atr_cls String)
                                        (atr_nme String)
                                        (atr_typ String)
                                        (atr_vis String)
                                        (atr_sco String)
                                    ))))


(define-fun get-param ((activity String)) Param
    (ite
        (str.contains activity " ")
        (PARAM (str.substr activity (str.indexof activity " " 0) (str.len activity)) "e")
        nil))


(define-fun infer-classes () Bool
    (forall
        ((x Actor))
        (=>
            (member x actors)
            (exists
                ((y Class))
                (and
                    (member y classes)
                    (and
                        (=
                            (cls_nme y)
                            (actor_name x))
                        (=
                            (cls_typ y)
                            "class")))))))


(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(assert (member (ACTOR "Members") C1a))
(assert (member (ACTOR "Books") C1a))
(assert (member (ACTOR "Language-tapes") C1a))
(declare-const C1 Clause)
```

```
(assert (= C1 (CLAUSE "exists" C1a )))
(assert (instantiate-clause C1))

(declare-const C2a (Set Actor))
(assert (member (ACTOR "Library") C2a))
(declare-const C2 Clause)
(assert (= C2 (CLAUSE "loans loan-item" C2a )))
(assert (instantiate-clause C2))

(declare-const C3a (Set Actor))
(assert (member (ACTOR "Library") C3a))
(declare-const C3 Clause)
(assert (= C3 (CLAUSE "issues membership-card" C3a )))
(assert (instantiate-clause C3))


(assert (member (STRUCT "Items" "must" "have" "bar-code") struct_sents))
(assert (member (TYPE_S "Books" "are" "items") type_sents))
(assert (member (STRUCT "Books" "must" "have" "title") struct_sents))
(assert (member (STRUCT "Books" "must" "have" "author") struct_sents))
(assert (member (TYPE_S "Tapes" "are" "items") type_sents))
(assert (member (STRUCT "Tapes" "must" "have" "language") struct_sents))
(assert (member (STRUCT "tapes" "must" "have" "level") struct_sents))
```

SMTLib 7.2: SMT-LIB code representing the class model inferred from SMTLib C.1 using CVC4

```
((classes (union (union (union (union (singleton (CLASS "Customers" "class")) (singleton
↪  (CLASS "Members" "class"))) (singleton (CLASS "Books" "class"))) (singleton (CLASS
↪  "Language-tapes" "class"))) (singleton (CLASS "Library" "class")))))

((types (union (union (union (union (singleton (TYPE "Customers")) (singleton (TYPE
↪  "Members"))) (singleton (TYPE "Books"))) (singleton (TYPE "Language-tapes")))
↪  (singleton (TYPE "Library")))))

((operations (union (union (union (union (singleton (OPR "Customers" "exists" "e" "+"
↪  "instance" nil)) (singleton (OPR "Members" "exists" "e" "+" "instance" nil)))
↪  (singleton (OPR "Books" "exists" "e" "+" "instance" nil))) (singleton (OPR
↪  "Language-tapes" "exists" "e" "+" "instance" nil))) (singleton (OPR "Library" "loans"
↪  "e" "+" "instance" (PARAM " loan-item" "e"))))))

((relations (singleton (REL "Library" " loan-item" "Association" "loans" "e" "*" "*"))))

((attributes (union (union (union (union (singleton (ATR "Items" "bar-code" "e" "e" "e"))
↪  (singleton (ATR "Books" "title" "e" "e" "e"))) (singleton (ATR "Books" "author" "e"
↪  "e" "e"))) (singleton (ATR "Tapes" "language" "e" "e" "e"))) (singleton (ATR "tapes"
↪  "level" "e" "e" "e")))))

((inheritances (union (singleton (INH "items" "Books")) (singleton (INH "items"
↪  "Tapes")))))
```

By looking at the segment of predicates in Predicates 7.2 and the SMT-Lib code in SMTLib 7.2 we can notice that they do contain similar elements. Though the notation of the predicates is simplified for the sake of clarity, the names of classes, types, attributes, operations and inheritances are explicitly stated, and they match the ones obtained with the inference rules.

T4TOMM also allows generating automatically the JSON representation of the SMT-LIB class model inferred, which is shown in JSON Model 7.1.

JSON Model 7.1: JSON representation of the model inferred in SMTLib 7.2


```
{
    "classes": {
        "customers": {
            "name": "Customers",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "members": {
            "name": "Members",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "books": {
            "name": "Books",
            "class_type": "class",
            "attributes": {
                "title": {
                    "name": "title"
                },
                "author": {
                    "name": "author"
                }
            },
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "language-tapes": {
            "name": "Language-tapes",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "library": {
            "name": "Library",
```

```
            "class_type": "class",
            "attributes": {},
            "operations": {
                "loans-epsilon-loan-item-epsilon": {
                    "name": "loans",
                    "parameters": {
                        "loan-item-epsilon": {
                            "name": "loan-item",
                        }
                    }
                }
            }
        },
        "items": {
            "name": "Items",
            "class_type": "class",
            "attributes": {
                "bar-code": {
                    "name": "bar-code"
                }
            },
            "operations": {}
        },
        "tapes": {
            "name": "Tapes",
            "class_type": "class",
            "attributes": {
                "language": {
                    "name": "language"
                },
                "level": {
                    "name": "level"
                }
            },
            "operations": {}
        }
    },
    "associations": {
        "items-books-inheritance": {
            "source_class_name": "items",
            "destination_class_name": "Books",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        },
        "items-tapes-inheritance": {
            "source_class_name": "items",
            "destination_class_name": "Tapes",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        }
    }
}
```


In this way, we demonstrate that the formal system proposed in Sec-

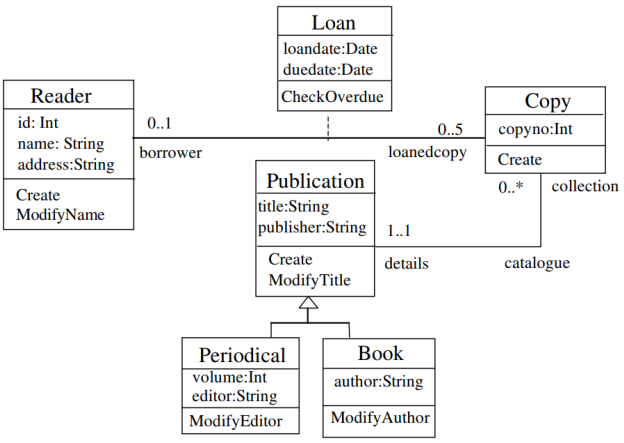


Figure 7.1: Kim's class diagram for the Library system

tion 5.2.1 and the implementation described in Section 6.3 are indeed capable of inferring class models from ConSpec specifications as expected.

7.2.3 Evaluation of Model Validation

In this section, we describe the evaluation of our theory to validate class diagrams with respect to ConSpec specifications. To do so, we demonstrate the manual process of validation, and subsequently, we use T4TOMM to evaluate invalid, sound, complete and valid model.

7.2.3.1 Manual validation

For this case, we refer again to the library system described in Text 7.5. In particular we will check the class diagram proposed by Kim[171] in Figure 7.1 against the specification shown in ConSpec 7.9, whose predicate formalization has been manually done in Predicates 7.1. The predicate formalisation for Kim's diagram has been done manually, and a simplified version of some of its predicates is shown in Predicates 7.3.

Predicates 7.3: Predicate formalization of Kim's class diagram[171] shown in Figure 7.1

```
M_k = {
  CLS(Reader)
  ATR(Reader, id, Int)
  ATR(Reader, name, String)
  ATR(Reader, address, String)
  OPR(Reader, Create)
  OPR(Reader, ModifyName)
  CLS(Loan)
  ATR(Loan, loanDate, Date)
  ATR(Loan, dueDate, Date)
  OPR(Loan, CheckOverdue)
  CLS(Copy)
  ATR(Copy, copyNo, Int)
  OPR(Copy, Create)
  CLS(Publication)
  ATR(Publication, title, String)
  ATR(Publication, publisher, String)
  OPR(Publication, Create)
  OPR(Publication, ModifyTitle)
  CLS(Periodical)
  ATR(Periodical, volume, Int)
  ATR(Periodical, editor, String)
  OPR(Periodical, ModifyEditor)
  CLS(Book)
  ATR(Book, author, String)
  OPR(Book, ModifyAuthor)
}
```

In order to evaluate this model, a semantic equivalence relation is required to be able to distinguish which elements have the same meaning. 7.1 list the semantic equivalent words for this diagram.

$$LoanTransaction \approx Loan$$

$$Reader \approx Customer \approx Customers \quad (7.1)$$

Semantic equivalences required to validate Predicates 7.3

Now the soundness of M_k, shown in Predicates 7.3 is evaluated by applying the validation rules to all the elements of the model. The first predicate to be evaluated is *CLS(Reader)*, which requires a clause of the type $ACTOR(x) \models CLS(x)$ such that $x \approx Reader$, by assigning $x = Customer$, the following application of the *class axiom* is obtained. The variable x is used instead of *Customer* to keep the application within the margins of the page.

$$\frac{ACTOR(x) \models CLS(x) \qquad CLS(Reader) \qquad x \approx Reader}{\top}$$

Because the axiom holds, it is concluded that the class *"Reader"* is valid. The same process is applied to the classes *"Loan"* , *"Copy"* , *"Publication"*

, *"Periodical"* , and *"Book"* , from which only *"Loan"* and *"Book"* are valid. Because the other three are not, we conclude that the model is not sound with no need for checking other predicates.

In order to validate completeness the predicates from S_1 are checked one by one. Starting with $ACTOR(Library)$, there should exist a predicate $CLS(x)$ in M_k such that $x \approx Library$. Because the set of semantic equivalences does not have an equivalent for *"Library"* , this axiom requires $x = Library$, that is, to have $CLS(Library)$ in M_k, which is not the case, hence the model is also incomplete. Since the model is neither complete nor sound, then it is proved to be invalid with respect to S_1.

7.2.3.2 Checking invalid model using T4TOMM

In Section 7.2.3.1 we proved manually that the diagram from Figure 7.1 is invalid with respect to the specification ConSpec 7.9. In here we demonstrate how to reach the same conclusion using T4TOMM.

First, we had to manually generate the JSON class model representing the Figure 7.1. A segment of this JSON model is shown in JSON Model 7.2, and the full model (JSON Model C.1) is provided in Appendix C.2.

JSON Model 7.2: JSON segment of a invalid model with respect to ConSpec 7.9


```
{
    "classes": {
        "loan": {
            "name": "Loan",
            "class_type": "class",
            "attributes": {
                "loandate": {
                    "name": "loandate",
                    "type": "Date"
                },
                "duedate": {
                    "name": "duedate",
                    "type": "Date"
                }
            },
            "operations": {
                "check-overdue": {
                    "name": "CheckOverdue",
                    "parameters": {}
                }
            }
        },
        "copy": {
            "name": "Copy",
            "attributes": {},
            "operations": {}
        },
        "reader": {
            "name": "Reader"
        },
```

```
        "publication": {}
    },
    "associations": {
        "publication-periodical-inheritance": {
            "source_class_name": "Publication",
            "destination_class_name": "Periodical",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        }
    }
}
```

This model was then automatically translated using T4TOMM into SMT-LIB, illustrated in SMTLib 7.3.

SMTLib 7.3: Segment of SMT-LIB model to check an invalid class model (JSON Model C.1) against ConSpec 7.9

```
(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (c_actors (Set Actor))
        ))))


; Class Diagram
(declare-datatypes ( (Class     0) ) ( ((CLASS
                                        (cls_nme String)
                                        (cls_typ String)
                                    ))))


;---------------------------------
; Sets
;---------------------------------
(declare-const clauses (Set Clause))
(assert (= clauses (as univset (Set Clause))))
; Class Diagram
(declare-const classes (Set Class))


; There is an actor thar derives the class
(define-fun check_class ((c Class)) Bool
    (exists
        ((a Actor))
        (and
            (member a actors)
            (is_syn
                (cls_nme c)
                (actor_name a)
            )
        )
    )
)

; All the classes come from an actor
```

```
(define-fun actors_rule () Bool
    (forall
        ((c Class))
        (=>
            (member c classes)
            (check_class c)
        )
    )
)

;---------------------------------
; Specification
;---------------------------------
(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(declare-const C1 Clause)
(assert (= C1 (CLAUSE "exists" C1a )))
(assert (instantiate-clause C1))

(assert (member (STRUCT "Items" "must" "have" "bar-code") struct_sents))


;---------------------------------
; Class model
;---------------------------------

; Classes
(assert (member (CLASS "Loan" "classifier") classes))
(assert (member (CLASS "Copy" "classifier") classes))


; Operations
(assert (member (OPR "Loan" "CheckOverdue" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Copy" "Create" "" "" "" (PARAM "" "")) operations))

; Attributes
(assert (member (ATR "Loan" "loandate" "" "" "") attributes))
(assert (member (ATR "Loan" "duedate" "" "" "") attributes))

;Inheritances
(assert (member (INH "Publication" "Periodical") inheritances))
(assert (member (INH "Publication" "Book") inheritances))
(assert (= (card inheritances) 2))


;---------------------------------
; Validating model
;---------------------------------
(declare-const classes_validation Bool)
(declare-const operations_validation Bool)
(declare-const attributes_validation Bool)
(declare-const inheritances_validation Bool)
(declare-const relations_validation Bool)
(assert (= classes_validation actors_rule))
(assert (= operations_validation operations_rule))
(assert (= attributes_validation attributes_rule))
(assert (= inheritances_validation inheritances_rule))


;---------------------------------
; Checks
;---------------------------------
```

```
(check-sat)
(get-value (classes_validation))
(get-value (operations_validation))
(get-value (attributes_validation))
(get-value (inheritances_validation))
(exit)
```

We know this diagram is invalid because the classes and the attributes it contains cannot be generated from ConSpec 7.9. The solution generated by the SMT solver is captured in SMTLib 7.4, and it shows that the validation of actors, operations, and attributes does not hold for soundness and completeness, while inheritance does hold for soundness, but not for completeness. These results match the observations made in the manual validation described in Section 7.2.3.1. Thus T4TOMM exhibits the adequate expected behaviour for model validation.

SMTLib 7.4: SMT-LIB code resulting from the validation of the invalid model SMTLib C.2

```
; Soundness
((classes_validation false))
((operations_validation false))
((attributes_validation false))
((inheritances_validation false))

; Completeness
((inverse_classes_validation false))
((inverse_operations_validation false))
((inverse_attributes_validation false))
((inverse_inheritances_validation false))
```

The complete JSON class model, and the SMT-LIB representation generated by T4TOMM are available in Appendix C.2 as JSON Model C.1 and SMTLib C.2 respectively.

7.2.3.3 Checking sound model using T4TOMM

For the evaluation of this case, we use a *sound* but incomplete class model with respect to ConSpec 7.9. We achieved this by removing one attribute and one operation from the class *"Books"* in the inferred model shown in JSON Model 7.1. A snippet of the modified class is presented in JSON Model 7.3.

JSON Model 7.3: JSON segment of a sound but incomplete model with respect to ConSpec 7.9

```
{
    "classes": {
        "books": {
```

```
            "name": "Books",
            "class_type": "class",
            "attributes": {
                "author": {
                    "name": "author"
                }
            },
            "operations": {}
        }
    },
    "associations": {
        "items-books-inheritance": {
            "source_class_name": "items",
            "destination_class_name": "Books",
            "type": "inheritance"
        }
    }
}
```

A portion of the SMT-LIB model generated by T4TOMM can be observed in SMTLib 7.5. This model is checked by CVC4, and generates the outputs shown in SMTLib 7.6. From here, it is observed that the model soundness holds for all the elements, and completeness fails for attributes and operations. The complete JSON and SMT-LIB models are included in Appendix C.2.0.2

SMTLib 7.5: Segment of SMT-LIB model to check a sound but incomplete class model (JSON Model C.2) against ConSpec 7.9

```
;---------------------------------
; Sets
;---------------------------------
; Specification
(declare-const clauses (Set Clause))
(assert (= clauses (as univset (Set Clause))))
; Class Diagram
(declare-const classes (Set Class))

;---------------------------------
; Synonyms
;---------------------------------
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Customers"   "customers"   "CUSTOMERS"   "Customer"   "customer"
    "CUSTOMER"   "client"   "CLIENT"   "Client"   "clients"
    (singleton "Customers"))) syns_dict))

(assert (= syns_dict (as univset (Set Entry))))

;---------------------------------
; Constructors and helpers
;---------------------------------

(define-fun is_syn ((w1 String) (w2 String)) Bool
    (or
        (= w1 w2)
        (exists
```

```
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))

(define-fun instantiate-clause ((c Clause)) Bool
    (and
        (member c clauses)
        (forall
            ((a Actor))
            (=>
                (member a (c_actors c))
                (member a actors)
            )
        )
    )
)

;---------------------------------
; Inference rules
;---------------------------------

; There is an actor thar derives the class
(define-fun check_class ((c Class)) Bool
    (exists
        ((a Actor))
        (and
            (member a actors)
            (is_syn
                (cls_nme c)
                (actor_name a)
            )
        )
    )
)

;---------------------------------
; Specification
;---------------------------------
(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(assert (instantiate-clause C1))

(declare-const C2a (Set Actor))
(assert (member (ACTOR "Library") C2a))
(assert (member (TYPE_S "Books" "are" "items") type_sents))

(assert (= (card struct_sents) 5))
(assert (= struct_sents (as univset (Set Structural_S))))

;---------------------------------
; Class model
;---------------------------------

; Classes
(assert (member (CLASS "Customers" "classifier") classes))

; Operations
(assert (member (OPR "Customers" "exists" "" "" "" (PARAM "" "")) operations))
```

```
; Attributes
(assert (member (ATR "Books" "author" "" "" "") attributes))

;Inheritances
(assert (member (INH "items" "Books") inheritances))

;---------------------------------
; Validating model
;---------------------------------
(declare-const classes_validation Bool)
(assert (= classes_validation actors_rule))

;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (classes_validation))
(get-value (operations_validation))
(get-value (attributes_validation))
(get-value (inheritances_validation))
(exit)
```

SMTLib 7.6: SMT-LIB code resulting from the validation of the model SMTLib C.3

```
; Soundness
((classes_validation true))
((operations_validation true))
((attributes_validation true))
((inheritances_validation true))

; Completeness
((inverse_classes_validation false))
((inverse_operations_validation false))
((inverse_attributes_validation false))
((inverse_inheritances_validation true))
```

7.2.3.4 Checking complete model using T4TOMM

We evaluated this case similarly to Section 7.2.3.3, by using a complete but not sound model. We modified the diagram inferred in Section 7.2.2.2, this time to add a random class *"RandomClass"* with a random attribute *"randomAttribute"* , which cannot be related to any element in ConSpec 7.9.

The class added to the original diagram is shown in JSON Model 7.4, and a segment of its formal representation is included in SMTLib 7.7. The complete models are available in Appendix C.2.0.2.

JSON Model 7.4: JSON segment of a complete but not sound model with respect to ConSpec 7.9

```
{
    "classes": {
        "randomclass": {
            "name": "RandomClass",
```

```
            "class_type": "class",
            "attributes": {
                "randomattribute": {
                    "name": "randomAttribute"
                }
            },
            "operations": {}
        }
    }
}
```

SMTLib 7.7: Segment of SMT-LIB model to check a complete but not sound class model (JSON Model C.3) against ConSpec 7.9

```
(define-fun inverese_operations_rule () Bool
    (forall
        ((c Clause))
        (=>
            (member c clauses)
            (check_activity (activity c))
        )
    )
)

(define-fun check_structures ((s Structural_S)) Bool
    (exists
        ((a Attribute))
        (and
            (member a attributes)
            (is_syn
                (property s)
                (atr_nme a)
            )
        )
    )
)

;---------------------------------
; Specification
;---------------------------------
(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(assert (member (ACTOR "Members") C1a))
(assert (member (STRUCT "Tapes" "must" "have" "language") struct_sents))
(assert (member (STRUCT "tapes" "must" "have" "level") struct_sents))

;---------------------------------
; Class model
;---------------------------------
; Classes
(assert (member (CLASS "RandomClass" "classifier") classes))
; Attributes
(assert (member (ATR "RandomClass" "randomAttribute" "" "" "") attributes))
(assert (member (ATR "Books" "title" "" "" "") attributes))

;---------------------------------
; Validating model
;---------------------------------
(declare-const inverse_classes_validation Bool)
(declare-const inverse_attributes_validation Bool)
```

```
(assert (= inverse_classes_validation inverse_actors_rule))
(assert (= inverse_attributes_validation inverese_attributes_rule))

;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (inverse_classes_validation))
(get-value (inverse_attributes_validation))
```

From the results shown in SMTLib 7.8 it can be seen that the model is complete, but not sound with respect to classes and attributes, since there are classes and attributes that do not belong to the specification. This is the expected result, confirming that our framework behaves as expected.

SMTLib 7.8: SMT-LIB code resulting from the validation of the model SMTLib C.4

```
; soundness
((classes_validation false))
((operations_validation true))
((attributes_validation false))
((inheritances_validation true))

; completeness
((inverse_classes_validation true))
((inverse_operations_validation true))
((inverse_attributes_validation true))
((inverse_inheritances_validation true))
```

7.2.3.5 Checking valid model using T4TOMM

In this section we demonstrate the validation of the class model inferred in JSON Model 7.1. This diagram is known to be valid because it was in fact generated directly from the specification.

By running T4TOMM validation function we obtain the formal model represented in SMTLib 7.9, which after being evaluated by CVC4 generates the output shown in SMTLib 7.10. From this output, it is concluded that the diagram is sound, and complete with respect to specification ConSpec 7.9 as expected.

SMTLib 7.9: Segment of SMT-LIB model to check an invalid class model (JSON Model C.1) against ConSpec 7.9

```
(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (c_actors (Set Actor))
        ))))
```

```
; Class Diagram
(declare-datatypes ( (Class     0) ) ( ((CLASS
                                        (cls_nme String)
                                        (cls_typ String)
                                    ))))


;---------------------------------
; Sets
;---------------------------------
(declare-const clauses (Set Clause))
(assert (= clauses (as univset (Set Clause))))
; Class Diagram
(declare-const classes (Set Class))


; There is an actor thar derives the class
(define-fun check_class ((c Class)) Bool
    (exists
        ((a Actor))
        (and
            (member a actors)
            (is_syn
                (cls_nme c)
                (actor_name a)
            )
        )
    )
)

; All the classes come from an actor
(define-fun actors_rule () Bool
    (forall
        ((c Class))
        (=>
            (member c classes)
            (check_class c)
        )
    )
)

;---------------------------------
; Specification
;---------------------------------
(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(declare-const C1 Clause)
(assert (= C1 (CLAUSE "exists" C1a )))
(assert (instantiate-clause C1))

(assert (member (STRUCT "Items" "must" "have" "bar-code") struct_sents))


;---------------------------------
; Class model
;---------------------------------

; Classes
(assert (member (CLASS "Loan" "classifier") classes))
(assert (member (CLASS "Copy" "classifier") classes))
```

```
; Operations
(assert (member (OPR "Loan" "CheckOverdue" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Copy" "Create" "" "" "" (PARAM "" "")) operations))

; Attributes
(assert (member (ATR "Loan" "loandate" "" "" "") attributes))
(assert (member (ATR "Loan" "duedate" "" "" "") attributes))

;Inheritances
(assert (member (INH "Publication" "Periodical") inheritances))
(assert (member (INH "Publication" "Book") inheritances))
(assert (= (card inheritances) 2))


;---------------------------------
; Validating model
;---------------------------------
(declare-const classes_validation Bool)
(declare-const operations_validation Bool)
(declare-const attributes_validation Bool)
(declare-const inheritances_validation Bool)
(declare-const relations_validation Bool)
(assert (= classes_validation actors_rule))
(assert (= operations_validation operations_rule))
(assert (= attributes_validation attributes_rule))
(assert (= inheritances_validation inheritances_rule))


;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (classes_validation))
(get-value (operations_validation))
(get-value (attributes_validation))
(get-value (inheritances_validation))
(exit)
```

SMTLib 7.10: SMT-LIB code resulting from the validation of the invalid model SMTLib C.2

```
; Soundness
((classes_validation false))
((operations_validation false))
((attributes_validation false))
((inheritances_validation false))

; Completeness
((inverse_classes_validation false))
((inverse_operations_validation false))
((inverse_attributes_validation false))
((inverse_inheritances_validation false))
```

Through these sections, we demonstrated how TOMM is capable of successfully checking invalid, incomplete, not sound and valid class models with respect to classes, attributes, operations and inheritances. Other types of associations (aggregation, composition, etc.) are not currently supported,

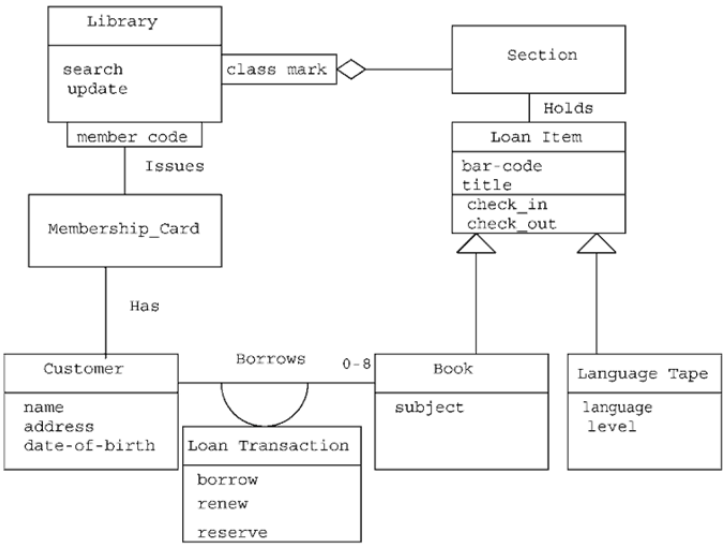


Figure 7.2: Class Diagrams for Library Example

hence are not included in this evaluation. Threats to validity include the validation of complex and existing diagrams which have not been properly generated. In addition, T4TOMM relies entirely on CVC4; hence, problems related to this SMT solver will be inhered by our tool.

7.2.4 Evaluation of Model Comparison

In this section we evaluate four possible scenarios, being a comparison of not equivalent models the first one. Left and right equivalent models are evaluated by modifying a base model used for the comparison. Finally, providing two equivalent models, we evaluate the overall approach. These comparisons are made through T4TOMM; however, we initially describe the manual operation of our comparison theory.

7.2.4.1 Manual Comparison

For this case, the diagrams developed by Callan[62] and Kim[171] are to be manually formalised and compared. The former is depicted in Figure 7.2 and the later was introduced in Section 7.2.3.1 as Figure 7.1.

Predicates 7.4 are used to formalize Callan's diagram as M_c with the ϵ parameters omitted. The predicates for Kim's model are presented in Predicates 7.3 as M_k

Predicates 7.4: Predicate formalization of Callan's class diagram shown in Figure 7.2

```
M_c={
  CLS(Library),
  OPR(Library, search),
  OPR(Library, update),
  CLS(Membership_Card),
  CLS(Customer),
  ATR(Customer, name),
  ATR(Customer, address),
  ATR(Customer, date-of-birth),
  CLS(Section),
  CLS(Loan Item),
  ATR(Loan Item, bar-code),
  ATR(Loan Item, title),
  OPR(Loan Item, check_in),
  OPR(Loan Item, check_out),
  CLS(Book),
  ATR(Book, subject),
  CLS(Language Tape),
  ATR(Language Tape, language),
  ATR(Language Tape, level),
  CLS(Loan Transaction),
  OPR(Loan Transaction, borrow),
  OPR(Loan Transaction, renew),
  OPR(Loan Transaction, reserve)

}
```

The equivalence of these two class models will be checked now. First left equivalence will be evaluated, that is for all elements in Predicates 7.4 there is an equivalence element in Predicates 7.3.

The class *CLS(Library)* is evaluated first through the *class axiom* of Section 5.4.1 must be checked. There is not a predicate *CLS(x)* such that $Library \approx x$, hence the predicate does not hold for this class. In consequence, we conclude that *left equivalence* does not hold.

Right equivalence is checked now, starting with the predicate *CLS(Reader)* from M_k. In this particular case, the class axiom holds, as it is shown in the following application of the corresponding inference rule, given that all the premises are available.

$$\frac{\text{CLS(Reader)} \qquad \text{CLS(Customer)} \qquad Reader \approx Customer}{\top}$$

However, this axiom is not applicable for all classes in M_k, as it happens with the class *CLS(Copy)* from M_k, which has no equivalent class in M_c. Hence, *right equivalence* is also not satisfied. In conclusion, it has been manually proved that these two diagrams are not equivalent according to our comparison theory.

7.2.4.2 Comparing not equivalent models using T4TOMM

For this evaluation case, we use the same class models from Section 7.2.4.1 (Figure 7.2 and Figure 7.1). T4TOMM requires to provide these models in a

JSON format. fragments of these JSON files are shown in JSON Model 7.1 and JSON Model 7.2 respectively, and their corresponding SMT-LIB models are combined in SMTLib C.2.

The output of T4TOMM is shown in SMTLib 7.11. In it, it is observed that neither left nor right equivalence are satisfied, which corresponds to the expected behaviour.

SMTLib 7.11: SMT-LIB model with the solution of not equivalent models

```
; Left equivalence
((equiv_classes false))
((equiv_attributes false))
((equiv_operations false))
((equiv_inheritances false))
((are_equivalent false))

; Right equivalence
((equiv_classes false))
((equiv_attributes false))
((equiv_operations false))
((equiv_inheritances false))
((are_equivalent false))
```

7.2.4.3 Comparing models with left equivalence using T4TOMM

In order to evaluate left equivalence alone, we have taken the class model shown in JSON Model 7.1 and its modified version show in JSON Model C.3, which includes an additional class. We expect this model to be left equivalent because we know that all elements in JSON Model 7.1 are definitely present in JSON Model C.3.

SMTLib 7.12: SMT-LIB model with the solution of left equivalent models

```
; Left equivalence
((equiv_classes true))
((equiv_attributes true))
((equiv_operations true))
((equiv_inheritances true))
((are_equivalent true))

; Right equivalence
((equiv_classes false))
((equiv_attributes true))
((equiv_operations true))
((equiv_inheritances true))
((are_equivalent true))
```

7.2.4.4 Comparing models with right equivalence using T4TOMM

We evaluate right equivalence by comparing the base class model show in JSON Model 7.1 against the reduced model shown in JSON Model C.2. The

model SMT-LIB model shown in SMTLib C.7 is evaluated by CVC4 which generates the solution shown in SMTLib 7.13. From this solution is it observed that left equivalence fails for the classes, while right equivalence holds for all the attributes, as it is expected.

SMTLib 7.13: SMT-LIB model with the solution of right equivalent models

```
; Left equivalence
((equiv_classes false))
((equiv_attributes true))
((equiv_operations true))
((equiv_inheritances true))
((are_equivalent true))

; Right equivalence
((equiv_classes true))
((equiv_attributes true))
((equiv_operations true))
((equiv_inheritances true))
((are_equivalent true))
```

7.2.4.5 Comparing equivalent models using T4TOMM

In this final evaluation, we compare a base model (JSON Model 7.1) against itself using some substitutions in the name of the classes, by semantically equivalent words. The complete SMT-LIB model (SMTLib C.8) is checked and the results shown in SMTLib 7.14 are obtained. From these results, we conclude that both models are proved to be equivalent as expected.

SMTLib 7.14: SMT-LIB model with the solution of equivalent models

```
; Left equivalence
((equiv_classes true))
((equiv_attributes true))
((equiv_operations true))
((equiv_inheritances true))
((are_equivalent true))

; Right equivalence
((equiv_classes true))
((equiv_attributes true))
((equiv_operations true))
((equiv_inheritances true))
((are_equivalent true))
```

7.2.5 Class Model Extractions with T4TOMM

We define model extraction as the recognition of elements from graphical representations of a class model (diagrams), Its implementation, presented in

Section 6.6, is evaluated by providing different class diagrams as images, and measuring the number of elements extracted correctly.

The set of diagrams used in the evaluation were drawn from two different sources. The first corresponding to diagrams that were already available in research publications and websites. The second one corresponding to diagrams generated by us using CASE tools explicitly for this evaluation.

We aimed to cover various types of diagrams as much as possible. These include partial class diagrams that have classes without attributes or classes with attributes but without any operations and complete diagrams containing classes, attributes and operations. Table 7.5 lists all the diagrams used in our evaluation, alongside their source, and the main features they cover. Due to space constraints (and also avoid information overload), we include only a subset of the diagrams along with their individual evaluation and the corresponding extracted JSON Class Model, containing only relevant fields such as the name for the class, attributes and operations. However, the remaining diagrams and their extracted models in entirety are available in Appendix C.4.

Table 7.5: Set of Class Diagrams used to evaluate model extraction

Id	Source	Feature
Figure C.1	generated	attributes
Figure C.2	generated	operations
Figure 7.3	generated	complete
Figure C.3	generated	coloured
Figure C.4	existing	attributes
Figure C.5	existing	classes
Figure C.6	existing	complete
Figure C.7	existing	complex
Figure C.8	existing	hand

We do not expect our current implementation to cover all these scenarios satisfactorily since it is not the purpose of our research. However, T4TOMM includes this feature as an additional contribution to aid the usage of TOMM, hence the need to evaluate its capabilities and limitations as well.

7.2.5.1 Extraction of complete diagram generated by us

In here we evaluate an ideal diagram for T4TOMM, which contains classes, attributes and operations. This diagram has been generated using CASE tools which results in more regular shapes for the classes (boxes) in comparison with diagrams drawn by hand. The diagram corresponds to Figure 7.3 and the results are shown in Table 7.6.

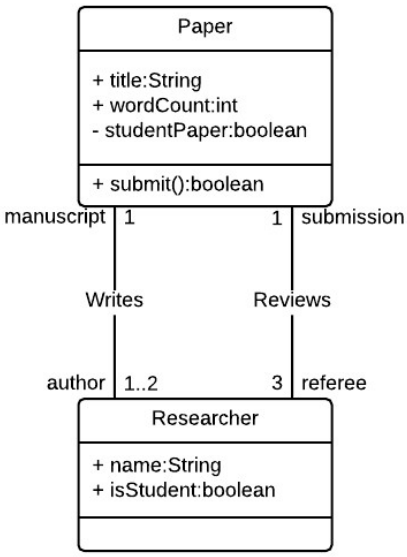


Figure 7.3: Class diagram generated by us containing classes, attributes and operations

This diagram has 2 classes: *"Paper"* and *"Researcher"* . The 3 attributes for each paper are *"title"* , *"wordCount"* and *"studentPaper"* , plus 2 attributes for each researcher, *"name"* and *"isStudent"* , for a total of 5 attributes. This diagram contains 1 method for the paper: *"submmit"* .

The output of T4TOMM is shown in JSON Model 7.5.

JSON Model 7.5: Extracted JSON class model from a complete class diagram generated by us


```
{
    "classes": {
        "class15567096207013591": {
            "name": "Class15567096207013591",
            "class_type": "class",
            "attributes": {
                "wordcount": {
                    "name": "wordCount"
                },
                "studentpaper": {
                    "name": "studentPaper"
                }
            },
            "operations": {
                "submit-epsilon": {
                    "name": "submit",
                    "parameters": {}
                }
            }
        },
        "class1556709621266361": {
```

```
            "name": "Class1556709621266361",
            "class_type": "class",
            "attributes": {
                "name": {
                    "name": "name"
                },
                "isstudent": {
                    "name": "isStudent"
                }
            },
            "operations": {}
        }
    },
    "associations": {}
}
```


It is observed that the extraction algorithm implemented in T4TOMM is capable of extracting the names of two attributes for the papers, namely *"wordCount"* and *"studenPaper"* , and 2 attributes for the researchers, namely *"name"* and *"isStudent"* . In addition one operation for papers is extracted: *"submit"* . Note that for this case, the name of the classes was not identified, hence T4TOMM assigned unique names to the assume classes, in order to group the attributes and operations extracted. These results are summarised in Table 7.6

Table 7.6: Evaluation of model extraction for complete class diagram generated by us

Item	Expected	Obtained
Classes	2	0
Attributes	5	4
Operations	1	1

7.2.5.2 Extraction of complete existing diagram

In here we evaluate an existing diagram which satisfies the same criteria of the one generated by us in Section 7.2.5.1. This diagram is shown in Figure 7.4, the extracted model is shown in JSON Model 7.6, and the results are summarised in Table 7.7

JSON Model 7.6: Extracted JSON class model from an existing complete class diagram


```
{
    "classes": {
        "class1556712268236669": {
            "name": "Class1556712268236669",
```

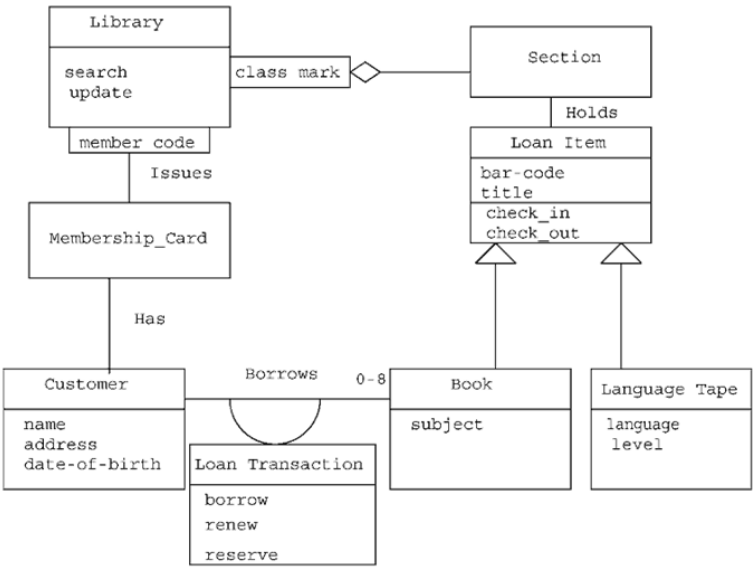


Figure 7.4: Existing class diagram containing classes, attributes and operations


```
        "class_type": "class",
        "attributes": {
            "update": {
                "name": "update"
            }
        },
        "operations": {}
    },
    "section": {
        "name": "Section",
        "class_type": "class",
        "attributes": {
            "holds": {
                "name": "Holds"
            },
            "loanitem": {
                "name": "LoanItem"
            },
            "title": {
                "name": "title"
            }
        },
        "operations": {}
    },
    "issues": {
        "name": "Issues",
        "class_type": "class",
        "attributes": {},
        "operations": {}
    },
    "languagetape": {
        "name": "LanguageTape",
        "class_type": "class",
```

```
            "attributes": {
                "language": {
                    "name": "language"
                },
                "level": {
                    "name": "level"
                }
            },
            "operations": {}
        },
        "class1556712270651666": {
            "name": "Class1556712270651666",
            "class_type": "class",
            "attributes": {
                "subject": {
                    "name": "subject"
                },
                "tion": {
                    "name": "tion"
                }
            },
            "operations": {}
        },
        "class1556712271316668": {
            "name": "Class1556712271316668",
            "class_type": "class",
            "attributes": {
                "customer": {
                    "name": "customer"
                },
                "name": {
                    "name": "name"
                },
                "address": {
                    "name": "address"
                }
            },
            "operations": {}
        },
        "bostows": {
            "name": "BOSTOWS",
            "class_type": "class",
            "attributes": {
                "borrow": {
                    "name": "borrow"
                },
                "renew": {
                    "name": "renew"
                },
                "reserve": {
                    "name": "reserve"
                }
            },
            "operations": {}
        }
    },
    "associations": {}
}
```


We observed that *"update"* was extracted as an attribute, but it should be

a method. However, this problem is attributed to the original class diagram, which is unclear even for a person. Individual observation has to be done to decide whether *"search"* and *"update"* are attributes or operations a *"Library"* , because there is no proper sectioning in the class box, and there is no use of parenthesis for any methods in the diagram.

It is also interesting to note that the class *"Loan Item"* is being extracted as an attribute for the class *"Section"* , this has to do with the alignment of the boxes in the diagram. Another noticeable problem is that the class *"Customer"* was not identified on its own, but it was assigned as an attribute for the randomly guessed class *"Class1556712271316668"* . The class that was extracted the best is *"Language Tape"* with two attributes, *"language"* and *"level"* .

Table 7.7: Evaluation of model extraction for existing complete class diagram

Item	Expected	Obtained
Classes	7	3
Attributes	8	6
Operations	7	0

In summary, only 3 out of 7 classes were extracted correctly, only 6 out of 8 real attributes were extracted, but additional attributes were generated either by mistaking class names or operations with attributes or by addling misidentified text. No operations were identified property because the diagram does not use parenthesis for the operations.

7.2.5.3 Results

Based on the evaluation of the extraction feature of T4TOMM, we concluded that though it does indeed help to extract some information from class diagrams, there is still a long way to go so that it will be entirely usable. The results are for each case are summarised in Table 7.8. Not that in this table we only count the items that are extracted correctly; for instance, if there is a random class or a miss-constructed attribute, it will not be considered in the count.

A particular case arose when extracting information from the existing diagram containing only classes (Figure C.5). This particular diagram causes the image processing library to crash, resulting in an empty class model to be generated. Based on this problem we reflect upon the decency on external libraries, which can always have some impact on the performance of T4TOMM.

Table 7.8: Summary of results for Class Model extraction with T4TOMM

	Classes		Attributes		Operations	
Diagram	Expected	Obtained	Expected	Obtained	Expected	Obtained
Figure C.1	2	0	5	3	0	0
Figure C.2	2	0	0	0	5	2
Figure 7.3	2	0	5	4	1	1
Figure C.3	6	4	9	6	3	2
Figure C.4	15	5	7	3	0	0
Figure C.5	18	0	0	0	0	0
Figure C.6	7	3	8	6	7	0
Figure C.7	29	0	11	3	26	0
Figure C.8	4	0	13	1	8	0

Regarding the graphical features of class diagrams, we noticed that extraction works better over diagrams that make use of proper sectioning for operations and attributes. While colouring diagrams does not have any effect on the current extraction, bigger spacing between boxes does help to identify classes better, In addition to the graphical properties, we observed that natural language processing could be used to identify operations and attributes based on POS tagging. This evaluation shows that indeed T4TOMM helps to extract some information from class diagrams in order to enable processing within the implementation of our theories.

7.2.6 Summary of Evaluation for TOMM and T4TOMM

In Section 7.2.2, Section 7.2.3 and Section 7.2.4 we evaluated of formal theories for model generation, validation, and comparison by defining specific cases that cover the possible outcomes for the theories. The manual application of each theory was initially demonstrated, and T4TOMM was used to automate the evaluation process of the cases identified.

The evaluation of our theory for *model generation* covers the case of formal inference, which showed to work as expected regarding inference of classes, attributes, operations and inheritances.

The evaluation of *model validation* was done by checking an invalid model, a sound model, a complete model, and a valid model. Once again, the theory and its implementation on T4TOMM showed to derive the expected results.

Model comparison was evaluated by checking not equivalent class models, as well as models presenting right equivalence, left equivalence, and total

equivalence. T4TOMM was able to perform these comparisons satisfactorily.

In addition to our theories, we evaluated *model extraction* using T4TOMM. We demonstrate that despite its limitations, it is nonetheless capable of extracting some elements of class diagrams, mostly attributes and operations. It showed to perform better over diagrams with clear layouts that follow the conventional notations of UML.

The application of our approach is evident here, for one might be tempted, based on the graphic representation of the class diagrams, to say that they *seem* valid, or *equivalent*. However, this observation has no proper grounds to be accepted, which is the problem addressed with TOMM.

7.2.6.1 Threats to validity

Our theories are shown to work properly over a subset of elements that are part of class diagrams, namely classes, attributes, operations and inheritance relations. However, this does not guarantee that these theories will be able to support other elements, such as aggregations or compositions as they are. Some extensions or modifications may be required to support not only new relations in class diagrams, but also additional diagrams, such as use cases, sequence diagrams, and other not-UML diagrams.

Regarding model comparison, we currently deal with models that have the same structure with variations on the nomenclature of their individual elements. Comparison based on functionality and design patterns will have to be included as future work.

In addition to the limitations of our theories proposed in TOMM, we reflect on the limitations of our proof-of-concept. The current implementation of T4TOMM requires processing using a Python interpreter[263], the NLTK library[199] to process ConSpec specification, CVC4[35] to solve the SMT-LIB models, and pytesseract[225] to extract models from diagrams. These dependencies constitute an external threat to validity for T4TOMM.

Though or theories have been evaluated against existing class diagrams and specifications, the number of possibilities out there makes us aware of unseen potential limitations that may arise when dealing with more complex diagrams and specifications alike.

Although the current versions of TOMM and T4TOMM are considerably restrictive and present some threats to validity, they also demonstrate how to establish a formal foundation to reason about class diagrams, which also motivate further research.

7.3 Summary

In this chapter, the applicability of SpecCNL and ConSpec was evaluated within several domains and requirements with different complexities. They demonstrated to be functional to some extent, but areas of improvement present, especially with comparison sentences, and the relation between the elements of a contract.

The usage of TOMM was discussed with several cases applicable, and the results were compared with those obtained from T4TOMM. Alike our specification format, our formal framework and our tool to support it are limited yet; however, their core components have been demonstrated to be useful and extensible.

Chapter 8

Conclusions

Our research has been motivated by the need to provide better support for software developers to improve the quality of software produced more generally in a variety of non-critical domains. This support comes in the form of integrated automated tools that make it easier to validate and verify the software as it is being developed. Verification is often done through the use of formal methods that are difficult to use or understand. In this research, we aimed to provide a framework that would facilitate the use of specific formal method techniques, including very scalable techniques such as constraint solvers, outside their usual area of application. The core of our framework combines the following formal reasoning activities: generation of models by inference, validation of existing models and comparison of models to determine their equivalence.

These reasoning activities are achieved employing deductive systems that are currently still relatively simple, supporting only basic types of constructs within class diagrams, that is, classes, attributes, methods and inheritance. However, the theories proposed here demonstrate how these diagrams can be made more reliable, and the same theories can be extended to cover more complex structures, such as OCL constraints and relations, and also other types of diagrams including use cases and activity diagrams. The novelty of our work has also been to embrace the formulation of requirements, and how through the use of NLP we can improve the generated models and hence tackle a significant source of ambiguity and failure in software development. A further novel contribution of our work concerns the analysis and comparison of models for their equivalence. There are many ways in which such notions of model equivalence can be used to improve models. It can, for instance, be used to clarify how a model may compare to other possible alternative models to the same approach. Moreover, we have shown the strength and potential of our work to combine constraint solvers with machine learning techniques

to enable even more varied tool support and tool integration. Even if done at a smaller scale, we showed evidence of how our approach could facilitate development if extended further.

Within our formalization, we have shown how several predicates can capture requirement specifications and class diagrams. Through the examples and the evaluation, we showed how these predicates facilitate three reasoning activities over class models: validation, generation and comparison. In addition, we introduced our specification document structure based on contract clauses using our proposed version of English sentences named SpeCNL. Encouraging results have been obtained from the evaluation of our controlled language and specification document, for they have shown their plausible application in real life requirements. We have also identified a few extensions that can be added to enhance their applicability.

Our theories for model validation, generation and comparison were explicitly evaluated by successfully comparing the outcome of our proof-of-concept against the expected output. We showed that for all the different scenarios the theories hold for relatively simple models containing classes, attributes, operations and inheritance. Further evaluation to determine its validity regarding more complex diagrams is not included in this research. T4TOMM showed to implement these theories properly and to aid in their usage since formal STM-LIB models are automatically generated and checked.

In order to use T4TOMM, however, class models have to be represented as JSON structures. This representation is partially supported by our implementation for model extraction, which is also part of T4TOMM. However, these models still require considerable manual corrections before being used within our formal theories.

8.1 Threats to Validity

We have evaluated our theories through the proof-of-concept represented by T4TOMM. However, this implementation has several external dependencies that limit the validity of our tool, and thus the validity of our theories. For instance, we assume that the results of the SMT solver are correct, and then we use that information to conduct our evaluation. However, should the SMT solver misbehave, the trust in our theories would have to be reassessed.

Besides, we assume that most of the functional requirements follow a similar structure and thus they can be expressed using SpeCNL. However, we advert the complexity of natural language, and we are aware that this assumption may not always hold.

Our theories are applicable for a limited set of activities (generation, vali-

dation and comparison) over a small set of artefacts (functional requirements and class models); however, little can be adequately concluded about its application over other activities, such as model checking or model refinement. We are also limited regarding the conclusions for other diagrams, though we argue that these foundations can be extended to other UML models, we have thus far no evidence that this is effectively the case, and further research is needed to validate this claim.

Given the uniqueness of our approach, in which we deal with the semantic equivalence of elements alone. It was challenging to find similar work that defines evaluation metrics suitable for our approach. As we progress with this research, we will be able to generate a more precise set of metrics for more standard evaluation.

8.2 Future Work

The work we proposed to do was very ambitious, and we have described before the contributions we have made as well as the novelty of our approach. There are also some elements that can be explored further and in particular elements of our framework that can be extended accordingly. In this section, we discuss some of the possible areas of future work.

Automatic requirements translation

Currently, in order to use TOMM or T4TOMM, the specifications have to be given in ConSpec format, which requires a manual translation of the original requirements. Further research about machine translation would lead to a partially-automated process in which existing requirements can be converted into ConSpec specifications without human effort.

The set of ConSpec specifications generated manually as part of the evaluation could also be used in a probabilistic approach for automated translation of specifications. This task would be done by generating all the possible SpeCNL alternatives for each sentence in the original text, and then finding the most likely to be the correct translation in comparison with the examples on the training set and in relation with the SpeCNL grammar.

Another possible approach for requirements translation could be to use Recurrent Neural Networks (RNNs) with encodings to detect translation patterns (rules) to be applied. Though these algorithms may help this task, there would still be (in principle) some pre-processing required by the user, who is the expert in the domain of the specification. For instance, the machine could assume that *"customer"* and *"client"* are the same, but

the correctness of such an assumption depends on the context of the given application. In order to avoid introducing mistakes, there should be an option for the user to decide/override such assumptions. In any case, either of these approaches can be implemented in order to automate the translation of requirements into ConSpec specifications supported by our framework.

ConSpec editor

In addition to the translation of existing requirements, a ConSpec editor would also aid the creation of specifications. This editor would contain the contract structure of the specifications so that the user would be able to enter details more easily, possibly also making use of visual blocks. Real-time checking of specifications could also be considered in order to make sure that errors can be found as early as when the user adds a requirement. Finally, further features such as automatic word completion and auto-correction based on our grammar defined in Section 4.1 could be added to enrich the ConSpec editor and make the approach more user-friendly.

Reasoning framework

The current version of TOMM contains theories to reason about structural properties of class diagrams. However, we expect that these foundations make it possible to develop and extend the theory further, in order to consider behavioural aspects, and additional diagrams.

An important observation about our inference calculus defined in Section 5.2 is that classes used in attribute, operation and inheritance rules must be first declared as actors within the specification, resulting in an unnecessary number of actors. This situation can be improved by extending the inference rules so that they also generate all the referred classes.

Our validity calculus allows us to establish *weak validity* and *partial completeness* properly. However, a more comprehensive set of inference rules would allow us to extend this calculus in order to support strong validity and total completeness, which requires us to weaken existing rules and to extend our definitions for validity.

T4TOMM

Our present implementation of T4TOMM can be extended further through a web-based interface that eases diagram visualization, generation and interaction, in addition to an editor for ConSpec and support for SMT-solvers other

than CVC4, and exploration of model size, and space limitations, which has not been done yet.

The approach used for class model extraction using k-means can be developed by further investigating other machine learning techniques. In particular, our comparison of class diagrams can be used in Generative Adversarial Networks to generate learning data.

8.3 Final Remarks

Our work and proposed framework were inspired by real needs in practice to verify the validity of software models more formally. Our framework is a step in that direction and has given us valuable insights into what can be done. We developed a framework TOMM which aimed to introduce novelty and clear benefits mainly at the requirements engineering stage of the development process. Some of our earlier work had worked on other stages of development, and it was an area which we perceived was lacking solutions.

Ours is an approach towards an automated reasoning framework that tackles requirements specification and engineering more systematically. Our evaluation has, nonetheless, shown us that there are still various areas of improvement that need to be researched further to meet our overall vision of tools for seamless verification of software at all stages of the development process.

Possible extensions for the formal systems were discussed to support other diagrams and other elements within class diagrams. Enhancements to our specification language and document structure can be pursued further, both at the foundational level and for the tools to support them. Improvements and extensions for T4TOMM have been discussed and will be explored further, specifically those related to techniques from machine learning to explore further the benefits that such integration can make for software development.

Appendices

Appendix A

Library Example

A.1 Requirements

A library issues loan items to customers. Each customer is known as a member and is issued a membership card that shows a unique member number. Along with the membership number, other details on a customer must be kept such as a name, address, and date of birth. The library is made up of a number of subject sections. Each section is denoted by a classification mark. A loan item is uniquely identified by a bar code. There are two types of loan items, language tapes, and books. A language tape has a title language (e.g. French), and level (e.g. beginner). A book has a title, and author(s). A customer may borrow up to a maximum of 8 items. An item can be borrowed, reserved or renewed to extend a current loan. When an item is issued the customer's membership number is scanned via a bar code reader or entered manually. If the membership is still valid and the number of items on loan less than 8, the book bar code is read, either via the bar code reader or entered manually. If the item can be issued (e.g. not reserved) the item is stamped and then issued. The library must support the facility for an item to be searched and for a daily update of records.

A.2 Contract Specification Document

```
Title: Library System
System Version: 1

Contracts:
```

```
- C1:
    Action: register member
    Actors:
        - Customers
    Preconditions:
        - Member's name must be provided
        - Member's address must be provided
        - Member's date-of-birth must be provided
    Action conditions:
        - The system must generate a member-id
    Postconditions:
        - The member-id must be unique

- C2:
    Action: register subject
    Actors:
        - Admins
    Action conditions:
        - The subject has a classification-mark

-C3:
    Action: register item
    Actors:
        - Admins
    Preconditions:
        - The item's bar-code must be unique
    Action conditions:
        - Items can be language-tapes or books
        - Language-tapes have title and level
        - Books have title and author

-C4:
    Action: borrow item
    Actors:
        - Members
    Preconditions:
        - Member-id must be valid
        - Number of loaned-items must be less than 8
    Action conditions:
        - Members can borrow up to 8 items
    Postconditions:
        - The item must be stamped

- C5:
    Action: reserve item
    Actors:
        - Members

- C6:
    Action: reserve item
    Actors:
        - Members

-C7:
    Action: extend loan
    Actors:
        - Members

- C8:
    Action: scan member-id
    Actors:
        - Admins
```

```
        Action conditions:
            - The member-id's bar-code can be scanned
            - The member-id can be typed

    - C9:
        Action: search item
        Actors:
            - Admins
            - Members

    - C10:
        Action: update records
        Actors:
            - Admins
```

Appendix B

SMT-LIB models

B.1 Inference example

SMTLib B.1: Inference Example for Library System

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;----------------------------------
; Datatypes
;----------------------------------
; Spec datatypes
(declare-datatypes ((Actor 0)) (((ACTOR (actor_name String)))))

(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (g_actors (Set Actor))
        ))))

(declare-datatypes ((Structural_S 0))
    (((STRUCT
        (entity String)
        (modal String)
        (have String)
        (property String)
        ))))

(declare-datatypes ((Type_S 0))
    (((TYPE_S
        (subtype String)
        (isa String)
        (type String)
        ))))



; UML datatypes
(declare-datatypes ( (Class     0) ) ( ((CLASS
    (cls_nme String) (cls_typ String) ))))
```

```
(declare-datatypes ( (Type      0) ) ( ((TYPE
    (typ_nme String) ))))

(declare-datatypes ( (Attribute 0) ) ( ((ATR
    (atr_cls String) (atr_nme String) (atr_typ String)
    (atr_vis String) (atr_sco String) ))))

(declare-datatypes ( (Param     0) ) ( ((PARAM
    (prm_nme String) (prm_typ String)) (nil) )))

(declare-datatypes ( (Operation 0) ) ( ((OPR
    (opr_cls String) (opr_nme String) (opr_typ String)
    (opr_vis String) (opr_sco String) (opr_prm Param)))))

(declare-datatypes ( (Relation 0) ) ( ((REL
    (rel_src String) (rel_des String) (rel_typ String)
    (rel_nme String) (rel_rol String) (rel_c_l String)
    (rel_c_u String) ))))

(declare-datatypes ( (Inheritance     0) ) ( ((INH
    (inh_sup String) (inh_sub String) ))))

(declare-datatypes ( (ClassDiagram     0) ) ( ((CD
    (dictionary (Set String)) (classes (Set Class))
    (types (Set Type)) (attributes (Set Attribute))
    (operations (Set Operation)) (relations (Set Relation))
    (inheritance (Set Inheritance)) ))))


;---------------------------------
; Sets
;---------------------------------
; CSD sets
(declare-fun clauses () (Set Clause))
(declare-fun actors () (Set Actor))
(declare-fun struct_sents () (Set Structural_S))
(declare-fun type_sents () (Set Type_S))
(assert (= clauses (as univset (Set Clause))))
(assert (= actors (as univset (Set Actor))))

; UML sets
(declare-fun classes    () (Set Class))
(declare-fun types      () (Set Type))
(declare-fun attributes () (Set Attribute))
(declare-fun operations () (Set Operation))
(declare-fun relations  () (Set Relation))
(declare-fun inheritances  () (Set Inheritance))


;---------------------------------
; Constructors and helpers
;---------------------------------
(define-fun instantiate-clause ((c Clause)) Bool
    (and
        (member c clauses)
        (forall
            ((a Actor))
            (=>
                (member a (g_actors c))
                (member a actors)))))
```

```
(define-fun get-param ((activity String)) Param
    (ite
        (str.contains activity " ")
        (PARAM (str.substr activity (str.indexof activity " " 0) (str.len activity)) "e")
        nil))

(define-fun get-activity ((activity String)) String
    (ite
        (str.contains activity " ")
        (str.substr activity 0 (str.indexof activity " " 0))
        activity))

(define-fun mk-operation ((a Actor) (activity String)) Operation
    (OPR (actor_name a) (get-activity activity) "e" "+" "instance" (get-param activity)))

(define-fun mk-relation ((o Operation)) Relation
    (REL (opr_cls o) (prm_nme (opr_prm o)) "Association" (opr_nme o) "e" "*" "*"))

(define-fun mk-attribute ((s Structural_S)) Attribute
    (ATR (entity s) (property s) "e" "e" "e")
)

(define-fun mk-inheritance ((t Type_S)) Inheritance
    (INH (type t) (subtype t))
)



;---------------------------------
; Inference rules
;---------------------------------

; For all actors in the set of actors, there is a class in the set of classes such that
↪  the name of the actor and the name of the class are the same.
(define-fun infer-classes () Bool
    (forall
        ((x Actor))
        (=>
            (member x actors)
            (exists
                ((y Class))
                (and
                    (member y classes)
                    (and
                        (=
                            (cls_nme y)
                            (actor_name x))
                        (=
                            (cls_typ y)
                            "classifier")))))))

; For all actors in the set of actors, there is a type in the set of types such that the
↪  name of the actor and the name of the class are the same.
(define-fun infer-types () Bool
    (forall
        ((x Actor))
        (=>
            (member x actors)
            (exists
                ((y Type))
                (and
                    (member y types)
```

```
                    (=
                        (actor_name x)
                        (typ_nme y)))))))

(define-fun infer-attribute ((s Structural_S)) Bool
    (member (mk-attribute s) attributes)
)

(define-fun infer-attributes () Bool
    (forall
        ((x Structural_S))
        (=>
            (member x struct_sents)
            (infer-attribute x)))
)

(define-fun infer-operation ((c Clause)) Bool
    (forall
        ((a Actor))
        (=>
            (and
                (member a actors)
                (member a (g_actors c))
            )
            (member (mk-operation a (activity c)) operations))))

(define-fun infer-operations () Bool
    (forall
       ((x Clause))
        (=>
            (member x clauses)
            (infer-operation x))))

(define-fun infer-relation ((o Operation)) Bool
    (member (mk-relation o) relations))

(define-fun infer-relations () Bool
    (forall
        ((x Operation))
        (=>
            (and
                (member x operations)
                (distinct
                    nil
                    (opr_prm x)))
            (infer-relation x))))

(define-fun infer-inheritance ((t Type_S)) Bool
    (member (mk-inheritance t) inheritances)
)

(define-fun infer-inheritances () Bool
    (forall
        ((x Type_S))
        (=>
            (member x type_sents)
            (infer-inheritance x)))
)


;--------------------------------
; Specification elements
```

```
;---------------------------------
(declare-const c1a (Set Actor))
(assert (member (ACTOR "manager") c1a))
(assert (member (ACTOR "user") c1a))
(declare-const c1 Clause)
(assert (= c1 (CLAUSE "registers users" c1a )))
(assert (instantiate-clause c1))

(declare-const c2a (Set Actor))
(assert (member (ACTOR "superadmin") c2a))
(assert (member (ACTOR "manager") c2a))
(assert (member (ACTOR "admin") c2a))
(declare-const c2 Clause)
(assert (= c2 (CLAUSE "delete users" c2a )))
(assert (instantiate-clause c2))


(declare-const c3a (Set Actor))
(assert (member (ACTOR "manager") c3a))
(assert (member (ACTOR "user") c3a))
(declare-const c3 Clause)
(assert (= c3 (CLAUSE "edit users" c3a )))
(assert (instantiate-clause c3))

(declare-const c4a (Set Actor))
(assert (member (ACTOR "user") c4a))
(declare-const c4 Clause)
(assert (= c4 (CLAUSE "login" c4a )))
(assert (instantiate-clause c4))

(assert (member (STRUCT "users" "must" "have" "name") struct_sents))
(assert (member (STRUCT "users" "can" "have" "address") struct_sents))
(assert (member (STRUCT "users" "must" "have" "age") struct_sents))

(assert (member (TYPE_S "admin" "is a" "user") type_sents))
(assert (member (TYPE_S "manager" "is an" "admin") type_sents))
(assert (member (TYPE_S "users" "are a" "user") type_sents))


;---------------------------------
; Infering model
;---------------------------------
(assert infer-operations)
(assert infer-classes)
(assert infer-types)
(assert infer-relations)
(assert infer-attributes)
(assert infer-inheritances)


;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (clauses))
(get-value (actors))
(get-value (struct_sents))
(get-value (type_sents))
(get-value (classes))
(get-value (types))
(get-value (operations))
(get-value (relations))
```

```
(get-value (attributes))
(get-value (inheritances))

(exit)
```

B.2 Soundness Model

SMTLib B.2: Soundness Model

```
; There is an actor thar derives the class
(define-fun check_class ((c Class)) Bool
    (exists
        ((a Actor))
        (and
            (member a actors)
            (is_syn
                (cls_nme c)
                (actor_name a)
            )
        )
    )
)

; All the classes come from an actor
(define-fun actors_rule () Bool
    (forall
        ((c Class))
        (=>
            (member c classes)
            (check_class c)
        )
    )
)


; There is
(define-fun check_operation ((o Operation)) Bool
    (exists
        ((c Clause))
        (and
            (member c clauses)
            (is_syn
                (opr_nme o)
                (activity c)
            )
        )
    )
)

; All the operations come from the specification
(define-fun operations_rule () Bool
    (forall
        ((o Operation))
        (=>
            (member o operations)
            (check_operation o)
        )
    )
```

```
)

(define-fun check_attribute ((a Attribute)) Bool
    (exists
        ((s Structural_S))
        (and
            (member s struct_sents)
            (is_syn
                (atr_nme a)
                (property s)
            )
        )
    )
)

(define-fun attributes_rule () Bool
    (forall
        ((a Attribute))
        (=>
            (member a attributes)
            (check_attribute a)
        )
    )
)

(define-fun check_inheritance ((h Inheritance)) Bool
    (exists
        ((s Type_S))
        (and
            (member s type_sents)
            (is_syn (inh_sup h) (type s))
            (is_syn (inh_sub h) (subtype s))
        )
    )
)

(define-fun inheritances_rule () Bool
    (forall
        ((h Inheritance))
        (=>
            (member h inheritances)
            (check_inheritance h)
        )
    )
)
```

B.3 Completeness Model

SMTLib B.3: Completeness Model

```
(define-fun check_actor ((a Actor)) Bool
    (exists
        ((c Class))
        (and
            (member c classes)
            (is_syn
                (actor_name a)
                (cls_nme c)
```

```
            )
        )
    )
)

(define-fun inverse_actors_rule () Bool
    (forall
        ((a Actor))
        (=>
            (member a actors)
            (check_actor a)
        )
    )
)

(define-fun check_activity ((a String)) Bool
    (exists
        ((o Operation))
        (and
            (member o operations)
            (is_syn
                a
                (opr_nme o)
            )
        )
    )
)

(define-fun inverese_operations_rule () Bool
    (forall
        ((c Clause))
        (=>
            (member c clauses)
            (check_activity (activity c))
        )
    )
)

(define-fun check_structures ((s Structural_S)) Bool
    (exists
        ((a Attribute))
        (and
            (member a attributes)
            (is_syn
                (property s)
                (atr_nme a)
            )
        )
    )
)

(define-fun inverese_attributes_rule () Bool
    (forall
        ((s Structural_S))
        (=>
            (member s struct_sents)
            (check_structures s)
        )
    )
)

(define-fun check_types ((s Type_S)) Bool
```

```
    (exists
        ((h Inheritance))
        (and
            (member h inheritances)
            (is_syn (inh_sub h) (subtype s))
            (is_syn (inh_sup h) (type s))
        )
    )
)

(define-fun inverese_inheritances_rule () Bool
    (forall
        ((s Type_S))
        (=>
            (member s type_sents)
            (check_types s)
        )
    )
)
```

B.4 Equivalence rules

SMTLib B.4: Equivalence Rules

```
;---------------------------------
; Inference rules
;---------------------------------

(define-fun check_dictionary ((s1 (Set String)) (s2 (Set String))) Bool
    (forall
        ((x String))
        (=>
            (member x s1)
            (exists
                ((y String))
                (and
                    (member y s2)
                    (is_syn x y)
                )
            )
        )
    )
)


(define-fun check_classes ((s1 (Set Class)) (s2 (Set Class))) Bool
    (forall
        ((x Class))
        (=>
            (member x s1)
            (exists
                ((y Class))
                (and
                    (member y s2)
                    (is_syn (cls_nme x) (cls_nme y))
                )
            )
        )
```

```
    )
)

(define-fun check_types ((s1 (Set Type)) (s2 (Set Type))) Bool
    (forall
        ((x Type))
        (=>
            (member x s1)
            (exists
                ((y Type))
                (and
                    (member y s2)
                    (is_syn (typ_nme x) (typ_nme y))
                )
            )
        )
    )
)

(define-fun check_attributes ((s1 (Set Attribute)) (s2 (Set Attribute))) Bool
    (forall
        ((x Attribute))
        (=>
            (member x s1)
            (exists
                ((y Attribute))
                (and
                    (member y s2)
                    (is_syn (atr_cls x) (atr_cls y))
                    (is_syn (atr_nme x) (atr_nme y))
                )
            )
        )
    )
)

(define-fun check_operations ((s1 (Set Operation)) (s2 (Set Operation))) Bool
    (forall
        ((x Operation))
        (=>
            (member x s1)
            (exists
                ((y Operation))
                (and
                    (member y s2)
                    (is_syn (opr_nme x) (opr_nme y))
                )
            )
        )
    )
)

(define-fun check_inheritances ((s1 (Set Inheritance)) (s2 (Set Inheritance))) Bool
    (forall
        ((x Inheritance))
        (=>
            (member x s1)
            (exists
                ((y Inheritance))
                (and
                    (member y s2)
                    (is_syn (inh_sup x) (inh_sup y))
```

```
                    (is_syn (inh_sub x) (inh_sub y))
                )
            )
        )
    )
)


;---------------------------------
; Class Diagram 1
;---------------------------------
;Dictionary
(declare-const dictionary1 (Set String))
(assert (member "customer" dictionary1))
(assert (member "client" dictionary1))
(assert (member "card" dictionary1))
(assert (= (card dictionary1) 3))
;Classes
(declare-const classes1 (Set Class))
(assert (member (CLASS "Customer" "classifier") classes1))
(assert (member (CLASS "Account" "classifier") classes1))
(assert (= (card classes1) 2))
;Types
(declare-const types1 (Set Type))
(assert (member (TYPE "Customer") types1))
(assert (member (TYPE "Client") types1))
(assert (member (TYPE "Card") types1))
(assert (= (card types1) 3))
;Attributes
(declare-const attributes1 (Set Attribute))
(assert (member (ATR "Customer" "name" "" "" "") attributes1))
(assert (member (ATR "Customer" "card" "" "" "") attributes1))
(assert (member (ATR "Card" "number" "" "" "") attributes1))
(assert (= (card attributes1) 3))
;Operations
(declare-const operations1 (Set Operation))
(assert (member (OPR "Admin" "insert" "" "" "" (PARAM "user" "")) operations1))
(assert (member (OPR "User" "delete" "" "" "" (PARAM "user" "")) operations1))
(assert (member (OPR "User" "edit" "" "" "" (PARAM "user" "")) operations1))
(assert (= (card operations1) 3))
;Relations
(declare-const relations1 (Set Relation))
(assert (member (REL "Customer" "Account" "association" "" "" "" "") relations1))
(assert (member (REL "Account" "Card" "association" "" "" "" "") relations1))
(assert (= (card relations1) 2))
;Inheritances
(declare-const inheritances1 (Set Inheritance))
(assert (member (INH "User" "Admin") inheritances1))
(assert (member (INH "DebitAccount" "Account") inheritances1))
(assert (member (INH "CreditAccount" "Account") inheritances1))
(assert (= (card inheritances1) 3))
;Class Diagram
(declare-const diagram1 ClassDiagram)
(assert (= diagram1
    (CD dictionary1 classes1 types1 attributes1 operations1 relations1 inheritances1)))

;---------------------------------
; Class Diagram 2
;---------------------------------
;Dictionary
(declare-const dictionary2 (Set String))
(assert (member "Clients" dictionary2))
```

```
(assert (member "clients" dictionary2))
(assert (member "debit card" dictionary2))
(assert (= (card dictionary2) 3))
;Classes
(declare-const classes2 (Set Class))
(assert (member (CLASS "Client" "classifier") classes2))
(assert (member (CLASS "Account" "classifier") classes2))
(assert (= (card classes2) 2))
;Types
(declare-const types2 (Set Type))
(assert (member (TYPE "Customer") types2))
(assert (member (TYPE "Card") types2))
(assert (= (card types2) 2))
;Attributes
(declare-const attributes2 (Set Attribute))
(assert (member (ATR "Client" "name" "" "" "") attributes2))
(assert (member (ATR "Client" "card" "" "" "") attributes2))
(assert (member (ATR "Card" "card number" "" "" "") attributes2))
(assert (= (card attributes2) 3))
;Operations
(declare-const operations2 (Set Operation))
(assert (member (OPR "User" "create" "" "" "" (PARAM "user" "")) operations2))
(assert (member (OPR "User" "delete" "" "" "" (PARAM "user" "")) operations2))
(assert (member (OPR "User" "edit" "" "" "" (PARAM "user" "")) operations2))
(assert (= (card operations2) 3))
;Relations
(declare-const relations2 (Set Relation))
(assert (member (REL "Customer" "Account" "association" "" "" "" "") relations2))
(assert (member (REL "Account" "Card" "association" "" "" "" "") relations2))
(assert (= (card relations2) 2))
;Inheritances
(declare-const inheritances2 (Set Inheritance))
(assert (member (INH "User" "Admin") inheritances2))
(assert (member (INH "DebitAccount" "Account") inheritances2))
(assert (member (INH "CreditAccount" "Account") inheritances2))
(assert (= (card inheritances2) 3))
;Class Diagram
(declare-const diagram2 ClassDiagram)
(assert (= diagram2
    (CD dictionary2 classes2 types2 attributes2 operations2 relations2 inheritances2)))


;---------------------------------
; Checks
;---------------------------------
(declare-const equiv_dicts Bool)
(declare-const equiv_classes Bool)
(declare-const equiv_types Bool)
(declare-const equiv_attributes Bool)
(declare-const equiv_operations Bool)
(declare-const equiv_inheritances Bool)
(assert (= equiv_dicts (check_dictionary dictionary1 dictionary2)))
(assert (= equiv_classes (check_classes classes1 classes2)))
(assert (= equiv_types (check_types types1 types2)))
(assert (= equiv_attributes (check_attributes attributes1 attributes2)))
(assert (= equiv_operations (check_operations operations1 operations2)))
(assert (= equiv_inheritances (check_inheritances inheritances1 inheritances2)))

(define-fun equiv_diagrams () Bool
    (and
        equiv_dicts
        equiv_classes
```

```
        equiv_types
        equiv_attributes
        equiv_operations
        equiv_inheritances
    )
)

(declare-const are_equivalent Bool)
(assert (= equiv_diagrams are_equivalent))

(check-sat)
(get-value (equiv_dicts))
(get-value (equiv_classes))
(get-value (equiv_types))
(get-value (equiv_attributes))
(get-value (equiv_operations))
(get-value (equiv_inheritances))
(get-value (are_equivalent))
(exit)
```

Appendix C

Evaluation

C.1 Model Inference

SMTLib C.1: Full SMT-LIB model to infer class model from ConSpec 7.9

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;----------------------------------
; Datatypes
;----------------------------------
; Spec datatypes
(declare-datatypes ((Actor 0)) (((ACTOR (actor_name String)))))

(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (g_actors (Set Actor))
        ))))

(declare-datatypes ((Structural_S 0))
    (((STRUCT
        (entity String)
        (modal String)
        (have String)
        (property String)
        ))))

(declare-datatypes ((Type_S 0))
    (((TYPE_S
        (subtype String)
        (isa String)
        (type String)
        ))))


; UML datatypes
(declare-datatypes ( (Class     0) ) ( ((CLASS
    (cls_nme String) (cls_typ String) ))))

(declare-datatypes ( (Type      0) ) ( ((TYPE
    (typ_nme String) ))))
```

```
(declare-datatypes ( (Attribute 0) ) ( ((ATR
    (atr_cls String) (atr_nme String) (atr_typ String)
    (atr_vis String) (atr_sco String) ))))

(declare-datatypes ( (Param     0) ) ( ((PARAM
    (prm_nme String) (prm_typ String)) (nil) )))

(declare-datatypes ( (Operation 0) ) ( ((OPR
    (opr_cls String) (opr_nme String) (opr_typ String)
    (opr_vis String) (opr_sco String) (opr_prm Param)))))

(declare-datatypes ( (Relation 0) ) ( ((REL
    (rel_src String) (rel_des String) (rel_typ String)
    (rel_nme String) (rel_rol String) (rel_c_l String)
    (rel_c_u String) ))))

(declare-datatypes ( (Inheritance     0) ) ( ((INH
    (inh_sup String) (inh_sub String) ))))

(declare-datatypes ( (ClassDiagram     0) ) ( ((CD
    (dictionary (Set String)) (classes (Set Class))
    (types (Set Type)) (attributes (Set Attribute))
    (operations (Set Operation)) (relations (Set Relation))
    (inheritance (Set Inheritance)) ))))


;---------------------------------
; Sets
;---------------------------------
; CSD sets
(declare-fun clauses () (Set Clause))
(declare-fun actors () (Set Actor))
(declare-fun struct_sents () (Set Structural_S))
(declare-fun type_sents () (Set Type_S))
(assert (= clauses (as univset (Set Clause))))
(assert (= actors (as univset (Set Actor))))

; UML sets
(declare-fun classes    () (Set Class))
(declare-fun types      () (Set Type))
(declare-fun attributes () (Set Attribute))
(declare-fun operations () (Set Operation))
(declare-fun relations  () (Set Relation))
(declare-fun inheritances  () (Set Inheritance))


;---------------------------------
; Constructors and helpers
;---------------------------------
(define-fun instantiate-clause ((c Clause)) Bool
    (and
        (member c clauses)
        (forall
            ((a Actor))
            (=>
                (member a (g_actors c))
                (member a actors)))))

(define-fun get-param ((activity String)) Param
    (ite
        (str.contains activity " ")
```

```
        (PARAM (str.substr activity (str.indexof activity " " 0) (str.len activity)) "e")
        nil))

(define-fun get-activity ((activity String)) String
    (ite
        (str.contains activity " ")
        (str.substr activity 0 (str.indexof activity " " 0))
        activity))

(define-fun mk-operation ((a Actor) (activity String)) Operation
    (OPR (actor_name a) (get-activity activity) "e" "+" "instance" (get-param activity)))

(define-fun mk-relation ((o Operation)) Relation
    (REL (opr_cls o) (prm_nme (opr_prm o)) "Association" (opr_nme o) "e" "*" "*"))

(define-fun mk-attribute ((s Structural_S)) Attribute
    (ATR (entity s) (property s) "e" "e" "e")
)

(define-fun mk-inheritance ((t Type_S)) Inheritance
    (INH (type t) (subtype t))
)



;---------------------------------
; Inference rules
;---------------------------------

; For all actors in the set of actors, there is a class in the set of classes such that
↪  the name of the actor and the name of the class are the same.
(define-fun infer-classes () Bool
    (forall
        ((x Actor))
        (=>
            (member x actors)
            (exists
                ((y Class))
                (and
                    (member y classes)
                    (and
                        (=
                            (cls_nme y)
                            (actor_name x))
                        (=
                            (cls_typ y)
                            "class")))))))

; For all actors in the set of actors, there is a type in the set of types such that the
↪  name of the actor and the name of the class are the same.
(define-fun infer-types () Bool
    (forall
        ((x Actor))
        (=>
            (member x actors)
            (exists
                ((y Type))
                (and
                    (member y types)
                    (=
                        (actor_name x)
                        (typ_nme y)))))))
```

```
(define-fun infer-attribute ((s Structural_S)) Bool
    (member (mk-attribute s) attributes)
)

(define-fun infer-attributes () Bool
    (forall
        ((x Structural_S))
        (=>
            (member x struct_sents)
            (infer-attribute x)))
)

(define-fun infer-operation ((c Clause)) Bool
    (forall
        ((a Actor))
        (=>
            (and
                (member a actors)
                (member a (g_actors c))
            )
            (member (mk-operation a (activity c)) operations))))

(define-fun infer-operations () Bool
    (forall
       ((x Clause))
        (=>
            (member x clauses)
            (infer-operation x))))

(define-fun infer-relation ((o Operation)) Bool
    (member (mk-relation o) relations))

(define-fun infer-relations () Bool
    (forall
        ((x Operation))
        (=>
            (and
                (member x operations)
                (distinct
                    nil
                    (opr_prm x)))
            (infer-relation x))))

(define-fun infer-inheritance ((t Type_S)) Bool
    (member (mk-inheritance t) inheritances)
)

(define-fun infer-inheritances () Bool
    (forall
        ((x Type_S))
        (=>
            (member x type_sents)
            (infer-inheritance x)))
)


;---------------------------------
; Specification elements
;---------------------------------

(declare-const C1a (Set Actor))
```

```
(assert (member (ACTOR "Customers") C1a))
(assert (member (ACTOR "Members") C1a))
(assert (member (ACTOR "Books") C1a))
(assert (member (ACTOR "Language-tapes") C1a))
(declare-const C1 Clause)
(assert (= C1 (CLAUSE "exists" C1a )))
(assert (instantiate-clause C1))

(declare-const C2a (Set Actor))
(assert (member (ACTOR "Library") C2a))
(declare-const C2 Clause)
(assert (= C2 (CLAUSE "loans loan-item" C2a )))
(assert (instantiate-clause C2))

(declare-const C3a (Set Actor))
(assert (member (ACTOR "Library") C3a))
(declare-const C3 Clause)
(assert (= C3 (CLAUSE "issues membership-card" C3a )))
(assert (instantiate-clause C3))



(assert (member (STRUCT "Items" "must" "have" "bar-code") struct_sents))
(assert (member (TYPE_S "Books" "are" "items") type_sents))
(assert (member (STRUCT "Books" "must" "have" "title") struct_sents))
(assert (member (STRUCT "Books" "must" "have" "author") struct_sents))
(assert (member (TYPE_S "Tapes" "are" "items") type_sents))
(assert (member (STRUCT "Tapes" "must" "have" "language") struct_sents))
(assert (member (STRUCT "tapes" "must" "have" "level") struct_sents))



;£fill: type sentences


;---------------------------------
; Infering model
;---------------------------------
(assert infer-operations)
(assert infer-classes)
(assert infer-types)
(assert infer-relations)
(assert infer-attributes)
(assert infer-inheritances)


;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (classes))
(get-value (types))
(get-value (operations))
(get-value (relations))
(get-value (attributes))
(get-value (inheritances))

(exit)
```

C.2 Model Validation

C.2.0.1 Invalid class model

JSON Model C.1: Full JSON of an invalid model with respect to ConSpec 7.9


```
{
    "classes": {
        "loan": {
            "name": "Loan",
            "class_type": "class",
            "attributes": {
                "loandate": {
                    "name": "loandate",
                    "type": "Date"
                },
                "duedate": {
                    "name": "duedate",
                    "type": "Date"
                }
            },
            "operations": {
                "check-overdue": {
                    "name": "CheckOverdue",
                    "parameters": {}
                }
            }
        },
        "copy": {
            "name": "Copy",
            "class_type": "class",
            "attributes": {
                "loandate": {
                    "name": "copyno",
                    "type": "Int"
                }
            },
            "operations": {
                "create": {
                    "name": "Create",
                    "parameters": {}
                }
            }
        },
        "reader": {
            "name": "Reader",
            "class_type": "class",
            "attributes": {
                "id": {
                    "name": "id",
                    "type": "Int"
                },
                "name": {
                    "name": "name",
                    "type": "String"
                },
                "address": {
                    "name": "address",
                    "type": "String"
                }
```

```
            },
            "operations": {
                "create": {
                    "name": "Create",
                    "parameters": {}
                },
                "modifyname": {
                    "name": "ModifyName",
                    "parameters": {}
                }
            }
        },
        "publication": {
            "name": "Publication",
            "class_type": "class",
            "attributes": {
                "title": {
                    "name": "title",
                    "type": "String"
                },
                "publisher": {
                    "name": "publisher",
                    "type": "String"
                }
            },
            "operations": {
                "create": {
                    "name": "Create",
                    "parameters": {}
                },
                "modifytitle": {
                    "name": "ModifyTitle",
                    "parameters": {}
                }
            }
        },
        "periodical": {
            "name": "Periodical",
            "class_type": "class",
            "attributes": {
                "volume": {
                    "name": "volume",
                    "type": "Int"
                },
                "editor": {
                    "name": "editor",
                    "type": "String"
                }
            },
            "operations": {
                "modifyeditor": {
                    "name": "ModifyEditor",
                    "parameters": {}
                }
            }
        },
        "book": {
            "name": "Book",
            "class_type": "class",
            "attributes": {
                "author": {
                    "name": "author",
```

```
                    "type": "String"
                }
            },
            "operations": {
                "modifyauthor": {
                    "name": "ModifyAuthor",
                    "parameters": {}
                }
            }
        }
    },
    "associations": {
        "publication-periodical-inheritance": {
            "source_class_name": "Publication",
            "destination_class_name": "Periodical",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        },
        "publication-book-inheritance": {
            "source_class_name": "Publication",
            "destination_class_name": "Book",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        }
    }
}
```

SMTLib C.2: Full SMT-LIB model to check an invalid class model (JSON Model C.1) against ConSpec 7.9

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;----------------------------------
; Datatypes
;----------------------------------
; Synonyms
(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                            (synonyms (Set String))
                                        ))))
; Specification
(declare-datatypes ((Actor 0)) (((ACTOR (actor_name String)))))

(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (c_actors (Set Actor))
        ))))

(declare-datatypes ((Sentence 0))
    (((SENTENCE
        (words (Set String))
        ))))
```

```
(declare-datatypes ((Structural_S 0))
    (((STRUCT
        (entity String)
        (modal String)
        (have String)
        (property String)
        ))))

(declare-datatypes ((Type_S 0))
    (((TYPE_S
        (subtype String)
        (isa String)
        (type String)
        ))))

; Class Diagram
(declare-datatypes ( (Class     0) ) ( ((CLASS
                                        (cls_nme String)
                                        (cls_typ String)
                                    ))))
(declare-datatypes ( (Type      0) ) ( ((TYPE
                                        (typ_nme String)
                                    ))))
(declare-datatypes ( (Attribute 0) ) ( ((ATR
                                        (atr_cls String)
                                        (atr_nme String)
                                        (atr_typ String)
                                        (atr_vis String)
                                        (atr_sco String)
                                    ))))
(declare-datatypes ( (Param     0) ) ( ((PARAM
                                        (prm_nme String)
                                        (prm_typ String)
                                    )
                                    (nil))))
(declare-datatypes ( (Operation 0) ) ( ((OPR
                                        (opr_cls String)
                                        (opr_nme String)
                                        (opr_typ String)
                                        (opr_vis String)
                                        (opr_sco String)
                                        (opr_prm Param)
                                    ))))
(declare-datatypes ( (Relation 0) ) ( ((REL
                                        (rel_src String)
                                        (rel_des String)
                                        (rel_typ String)
                                        (rel_nme String)
                                        (rel_rol String)
                                        (rel_c_l String)
                                        (rel_c_u String)
                                    ))))
(declare-datatypes ( (Inheritance     0) ) ( ((INH
                                        (inh_sup String)
                                        (inh_sub String)
                                    ))))


;---------------------------------
; Sets
;---------------------------------
; Synonyms
```

```
; Specification
(declare-const clauses (Set Clause))
(declare-const actors (Set Actor))
(declare-const sentences (Set Sentence))
(declare-const struct_sents (Set Structural_S))
(declare-const type_sents (Set Type_S))
(assert (= clauses (as univset (Set Clause))))
(assert (= actors (as univset (Set Actor))))
(assert (= sentences (as univset (Set Sentence))))
; Class Diagram
(declare-const classes (Set Class))
(declare-const types (Set Type))
(declare-const attributes (Set Attribute))
(declare-const operations (Set Operation))
(declare-const relations (Set Relation))
(declare-const inheritances (Set Inheritance))


;--------------------------------
; Synonyms
;--------------------------------
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Paper" "paper" "PAPER" "Papers" "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert
    "title"   "TITLE"   "Title"   "titles"   "statute_title"   "STATUTE_TITLE"
    ↪  "StatuteTitle"   "statuteTitle"   "statute-title"   "statute_titles"     "Statute
    ↪  Title"   "rubric"   "RUBRIC"   "Rubric"   "rubrics"
    (singleton "title"))) syns_dict))

(assert (member (mk-entry (insert
    "word-count"   "WORD-COUNT"   "Word-count"   "word_count"   "word-counts"
    "word_counts"   "Word Count"
    (singleton "word-count"))) syns_dict))

(assert (member (mk-entry (insert
    "student-indicator"   "STUDENT-INDICATOR"   "Student-indicator"
    "student_indicator"   "student-indicators"   "student_indicators"
    "Student Indicator"
    (singleton "student-indicator"))) syns_dict))

(assert (member (mk-entry (insert
    "authors"   "AUTHORS"   "Authors"   "author"   "writer"   "WRITER"
    "Writer"   "writers"   "AUTHOR"   "Author"
    (singleton "authors"))) syns_dict))

(assert (member (mk-entry (insert
    "referees"   "REFEREES"   "Referees"   "referee"   "REFEREE"   "Referee"
    "ref"   "REF"   "Ref"   "refs"
    (singleton "referees"))) syns_dict))

(assert (member (mk-entry (insert
    "Researcher"   "researcher"   "RESEARCHER"   "Researchers"   "researchers"
    "research_worker"   "RESEARCH_WORKER"   "ResearchWorker"   "researchWorker"
    "research-worker"   "research_workers"   "Research Worker"   "investigator"
    "INVESTIGATOR"   "Investigator"   "investigators"
    (singleton "Researcher"))) syns_dict))

(assert (member (mk-entry (insert
```

```
    "name"   "NAME"   "Name"   "names"
    (singleton "name"))) syns_dict))

(assert (= syns_dict (as univset (Set Entry))))

;---------------------------------
; Constructors and helpers
;---------------------------------

(define-fun is_syn ((w1 String) (w2 String)) Bool
    (or
        (= w1 w2)
        (exists
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))

(define-fun instantiate-clause ((c Clause)) Bool
    (and
        (member c clauses)
        (forall
            ((a Actor))
            (=>
                (member a (c_actors c))
                (member a actors)
            )
        )
    )
)




;---------------------------------
; Inference rules
;---------------------------------

; There is an actor thar derives the class
(define-fun check_class ((c Class)) Bool
    (exists
        ((a Actor))
        (and
            (member a actors)
            (is_syn
                (cls_nme c)
                (actor_name a)
            )
        )
    )
)

; All the classes come from an actor
(define-fun actors_rule () Bool
    (forall
        ((c Class))
        (=>
            (member c classes)
            (check_class c)
        )
```

```
    )
)


; There is
(define-fun check_operation ((o Operation)) Bool
    (exists
        ((c Clause))
        (and
            (member c clauses)
            (is_syn
                (opr_nme o)
                (activity c)
            )
        )
    )
)

; All the operations come from the specification
(define-fun operations_rule () Bool
    (forall
        ((o Operation))
        (=>
            (member o operations)
            (check_operation o)
        )
    )
)

(define-fun check_attribute ((a Attribute)) Bool
    (exists
        ((s Structural_S))
        (and
            (member s struct_sents)
            (is_syn
                (atr_nme a)
                (property s)
            )
        )
    )
)

(define-fun attributes_rule () Bool
    (forall
        ((a Attribute))
        (=>
            (member a attributes)
            (check_attribute a)
        )
    )
)

(define-fun check_inheritance ((h Inheritance)) Bool
    (exists
        ((s Type_S))
        (and
            (member s type_sents)
            (is_syn (inh_sup h) (type s))
            (is_syn (inh_sub h) (subtype s))
        )
    )
)
```

```
(define-fun inheritances_rule () Bool
    (forall
        ((h Inheritance))
        (=>
            (member h inheritances)
            (check_inheritance h)
        )
    )
)

(define-fun check_relation ((r Relation)) Bool
    (exists
        ((s Sentence))
        (and
            (member s sentences)
            (member (rel_src r) (words s))
            (member (rel_des r) (words s))
        )
    )
)

(define-fun relations_rule () Bool
    (forall
        ((r Relation))
        (=>
            (member r relations)
            (check_relation r)
        )
    )
)


;---------------------------------
; Specification
;---------------------------------
(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(assert (member (ACTOR "Members") C1a))
(assert (member (ACTOR "Books") C1a))
(assert (member (ACTOR "Language-tapes") C1a))
(declare-const C1 Clause)
(assert (= C1 (CLAUSE "exists" C1a )))
(assert (instantiate-clause C1))

(declare-const C2a (Set Actor))
(assert (member (ACTOR "Library") C2a))
(declare-const C2 Clause)
(assert (= C2 (CLAUSE "loans loan-item" C2a )))
(assert (instantiate-clause C2))

(declare-const C3a (Set Actor))
(assert (member (ACTOR "Library") C3a))
(declare-const C3 Clause)
(assert (= C3 (CLAUSE "issues membership-card" C3a )))
(assert (instantiate-clause C3))

(assert (= (card actors) 6))
(assert (= actors (as univset (Set Actor))))

(assert (= (card clauses) 3))
(assert (= clauses (as univset (Set Clause))))
```

```
(assert (member (STRUCT "Items" "must" "have" "bar-code") struct_sents))
(assert (member (TYPE_S "Books" "are" "items") type_sents))
(assert (member (STRUCT "Books" "must" "have" "title") struct_sents))
(assert (member (STRUCT "Books" "must" "have" "author") struct_sents))
(assert (member (TYPE_S "Tapes" "are" "items") type_sents))
(assert (member (STRUCT "Tapes" "must" "have" "language") struct_sents))
(assert (member (STRUCT "tapes" "must" "have" "level") struct_sents))


(assert (= (card struct_sents) 5))
(assert (= struct_sents (as univset (Set Structural_S))))

(assert (= (card type_sents) 2))
(assert (= type_sents (as univset (Set Type_S))))



; Shadow actors for inverse actors rule
(assert (= actors (as univset (Set Actor))))


;---------------------------------
; Class model
;---------------------------------

; Classes
(assert (member (CLASS "Loan" "classifier") classes))
(assert (member (CLASS "Copy" "classifier") classes))
(assert (member (CLASS "Reader" "classifier") classes))
(assert (member (CLASS "Publication" "classifier") classes))
(assert (member (CLASS "Periodical" "classifier") classes))
(assert (member (CLASS "Book" "classifier") classes))
(assert (= (card classes) 6))
(assert (= classes (as univset (Set Class))))

; Operations
(assert (member (OPR "Loan" "CheckOverdue" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Copy" "Create" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Reader" "Create" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Reader" "ModifyName" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Publication" "Create" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Publication" "ModifyTitle" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Periodical" "ModifyEditor" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Book" "ModifyAuthor" "" "" "" (PARAM "" "")) operations))
(assert (= (card operations) 8))

; Attributes
(assert (member (ATR "Loan" "loandate" "" "" "") attributes))
(assert (member (ATR "Loan" "duedate" "" "" "") attributes))
(assert (member (ATR "Copy" "copyno" "" "" "") attributes))
(assert (member (ATR "Reader" "id" "" "" "") attributes))
(assert (member (ATR "Reader" "name" "" "" "") attributes))
(assert (member (ATR "Reader" "address" "" "" "") attributes))
(assert (member (ATR "Publication" "title" "" "" "") attributes))
(assert (member (ATR "Publication" "publisher" "" "" "") attributes))
(assert (member (ATR "Periodical" "volume" "" "" "") attributes))
(assert (member (ATR "Periodical" "editor" "" "" "") attributes))
(assert (member (ATR "Book" "author" "" "" "") attributes))
(assert (= (card attributes) 11))
```

```
;Inheritances
(assert (member (INH "Publication" "Periodical") inheritances))
(assert (member (INH "Publication" "Book") inheritances))
(assert (= (card inheritances) 2))

;Relations
(assert (= (card relations) 0))

;---------------------------------
; Validating model
;---------------------------------
(declare-const classes_validation Bool)
(declare-const operations_validation Bool)
(declare-const attributes_validation Bool)
(declare-const inheritances_validation Bool)
(declare-const relations_validation Bool)
(assert (= classes_validation actors_rule))
(assert (= operations_validation operations_rule))
(assert (= attributes_validation attributes_rule))
(assert (= inheritances_validation inheritances_rule))


;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (classes_validation))
(get-value (operations_validation))
(get-value (attributes_validation))
(get-value (inheritances_validation))
(exit)
```

C.2.0.2 Sound class model

JSON Model C.2: Full JSON of a sound but incomplete model with respect to ConSpec 7.9

```
{
    "classes": {
        "customers": {
            "name": "Customers",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists"
                },
                "parameters": {}
            }
        },
        "members": {
            "name": "Members",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists"
                },
```

```
            "parameters": {}
        }
    },
    "books": {
        "name": "Books",
        "class_type": "class",
        "attributes": {
            "author": {
                "name": "author"
            }
        },
        "operations": {}
    },
    "language-tapes": {
        "name": "Language-tapes",
        "class_type": "class",
        "attributes": {},
        "operations": {
            "exists-epsilon": {
                "name": "exists"
            },
            "parameters": {}
        }
    },
    "library": {
        "name": "Library",
        "class_type": "class",
        "attributes": {},
        "operations": {
            "loans-epsilon-loan-item-epsilon": {
                "name": "loans"
            },
            "parameters": {
                "loan-item-epsilon": {
                    "name": "loan-item"
                }
            }
        }
    },
    "items": {
        "name": "Items",
        "class_type": "class",
        "attributes": {
            "bar-code": {
                "name": "bar-code"
            }
        },
        "operations": {}
    },
    "tapes": {
        "name": "Tapes",
        "class_type": "class",
        "attributes": {
            "language": {
                "name": "language"
            },
            "level": {
                "name": "level"
            }
        },
        "operations": {}
    }
```

```
    },
    "associations": {
        "items-books-inheritance": {
            "source_class_name": "items",
            "destination_class_name": "Books",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        },
        "items-tapes-inheritance": {
            "source_class_name": "items",
            "destination_class_name": "Tapes",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        }
    }
}
```

SMTLib C.3: Full SMT-LIB model to check a sound but incomplete class model (JSON Model C.2) against ConSpec 7.9

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;----------------------------------
; Datatypes
;----------------------------------
; Synonyms
(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                        (synonyms (Set String))
                                    ))))
; Specification
(declare-datatypes ((Actor 0)) (((ACTOR (actor_name String)))))

(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (c_actors (Set Actor))
        ))))

(declare-datatypes ((Sentence 0))
    (((SENTENCE
        (words (Set String))
        ))))

(declare-datatypes ((Structural_S 0))
    (((STRUCT
        (entity String)
        (modal String)
        (have String)
        (property String)
        ))))

(declare-datatypes ((Type_S 0))
    (((TYPE_S
```

```
        (subtype String)
        (isa String)
        (type String)
        ))))

; Class Diagram
(declare-datatypes ( (Class     0) ) ( ((CLASS
                                        (cls_nme String)
                                        (cls_typ String)
                                    ))))
(declare-datatypes ( (Type      0) ) ( ((TYPE
                                        (typ_nme String)
                                    ))))
(declare-datatypes ( (Attribute 0) ) ( ((ATR
                                        (atr_cls String)
                                        (atr_nme String)
                                        (atr_typ String)
                                        (atr_vis String)
                                        (atr_sco String)
                                    ))))
(declare-datatypes ( (Param     0) ) ( ((PARAM
                                        (prm_nme String)
                                        (prm_typ String)
                                    )
                                    (nil))))
(declare-datatypes ( (Operation 0) ) ( ((OPR
                                        (opr_cls String)
                                        (opr_nme String)
                                        (opr_typ String)
                                        (opr_vis String)
                                        (opr_sco String)
                                        (opr_prm Param)
                                    ))))
(declare-datatypes ( (Relation 0) ) ( ((REL
                                        (rel_src String)
                                        (rel_des String)
                                        (rel_typ String)
                                        (rel_nme String)
                                        (rel_rol String)
                                        (rel_c_l String)
                                        (rel_c_u String)
                                    ))))
(declare-datatypes ( (Inheritance     0) ) ( ((INH
                                        (inh_sup String)
                                        (inh_sub String)
                                    ))))


;---------------------------------
; Sets
;---------------------------------
; Synonyms
; Specification
(declare-const clauses (Set Clause))
(declare-const actors (Set Actor))
(declare-const sentences (Set Sentence))
(declare-const struct_sents (Set Structural_S))
(declare-const type_sents (Set Type_S))
(assert (= clauses (as univset (Set Clause))))
(assert (= actors (as univset (Set Actor))))
(assert (= sentences (as univset (Set Sentence))))
; Class Diagram
```

```
(declare-const classes (Set Class))
(declare-const types (Set Type))
(declare-const attributes (Set Attribute))
(declare-const operations (Set Operation))
(declare-const relations (Set Relation))
(declare-const inheritances (Set Inheritance))


;--------------------------------
; Synonyms
;--------------------------------
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Paper" "paper" "PAPER" "Papers" "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert
    "title"   "TITLE"   "Title"   "titles"   "statute_title"   "STATUTE_TITLE"
    ↪ "StatuteTitle"   "statuteTitle"   "statute-title"   "statute_titles"     "Statute
    ↪ Title"   "rubric"   "RUBRIC"   "Rubric"   "rubrics"
    (singleton "title"))) syns_dict))

(assert (member (mk-entry (insert
    "word-count"   "WORD-COUNT"   "Word-count"   "word_count"   "word-counts"
    "word_counts"   "Word Count"
    (singleton "word-count"))) syns_dict))

(assert (member (mk-entry (insert
    "student-indicator"   "STUDENT-INDICATOR"   "Student-indicator"
    "student_indicator"   "student-indicators"   "student_indicators"
    "Student Indicator"
    (singleton "student-indicator"))) syns_dict))

(assert (member (mk-entry (insert
    "authors"   "AUTHORS"   "Authors"   "author"   "writer"   "WRITER"
    "Writer"   "writers"   "AUTHOR"   "Author"
    (singleton "authors"))) syns_dict))

(assert (member (mk-entry (insert
    "referees"   "REFEREES"   "Referees"   "referee"   "REFEREE"   "Referee"
    "ref"   "REF"   "Ref"   "refs"
    (singleton "referees"))) syns_dict))

(assert (member (mk-entry (insert
    "Researcher"   "researcher"   "RESEARCHER"   "Researchers"   "researchers"
    "research_worker"   "RESEARCH_WORKER"   "ResearchWorker"   "researchWorker"
    "research-worker"   "research_workers"   "Research Worker"   "investigator"
    "INVESTIGATOR"   "Investigator"   "investigators"
    (singleton "Researcher"))) syns_dict))

(assert (member (mk-entry (insert
    "name"   "NAME"   "Name"   "names"
    (singleton "name"))) syns_dict))

(assert (= syns_dict (as univset (Set Entry))))

;--------------------------------
; Constructors and helpers
;--------------------------------

(define-fun is_syn ((w1 String) (w2 String)) Bool
```

```
    (or
        (= w1 w2)
        (exists
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))

(define-fun instantiate-clause ((c Clause)) Bool
    (and
        (member c clauses)
        (forall
            ((a Actor))
            (=>
                (member a (c_actors c))
                (member a actors)
            )
        )
    )
)




;---------------------------------
; Inference rules
;---------------------------------

; There is an actor thar derives the class
(define-fun check_class ((c Class)) Bool
    (exists
        ((a Actor))
        (and
            (member a actors)
            (is_syn
                (cls_nme c)
                (actor_name a)
            )
        )
    )
)

; All the classes come from an actor
(define-fun actors_rule () Bool
    (forall
        ((c Class))
        (=>
            (member c classes)
            (check_class c)
        )
    )
)


; There is
(define-fun check_operation ((o Operation)) Bool
    (exists
        ((c Clause))
        (and
            (member c clauses)
```

```
            (is_syn
                (opr_nme o)
                (activity c)
            )
        )
    )
)

; All the operations come from the specification
(define-fun operations_rule () Bool
    (forall
        ((o Operation))
        (=>
            (member o operations)
            (check_operation o)
        )
    )
)

(define-fun check_attribute ((a Attribute)) Bool
    (exists
        ((s Structural_S))
        (and
            (member s struct_sents)
            (is_syn
                (atr_nme a)
                (property s)
            )
        )
    )
)

(define-fun attributes_rule () Bool
    (forall
        ((a Attribute))
        (=>
            (member a attributes)
            (check_attribute a)
        )
    )
)

(define-fun check_inheritance ((h Inheritance)) Bool
    (exists
        ((s Type_S))
        (and
            (member s type_sents)
            (is_syn (inh_sup h) (type s))
            (is_syn (inh_sub h) (subtype s))
        )
    )
)

(define-fun inheritances_rule () Bool
    (forall
        ((h Inheritance))
        (=>
            (member h inheritances)
            (check_inheritance h)
        )
    )
)
```

```
(define-fun check_relation ((r Relation)) Bool
    (exists
        ((s Sentence))
        (and
            (member s sentences)
            (member (rel_src r) (words s))
            (member (rel_des r) (words s))
        )
    )
)

(define-fun relations_rule () Bool
    (forall
        ((r Relation))
        (=>
            (member r relations)
            (check_relation r)
        )
    )
)


;---------------------------------
; Specification
;---------------------------------
(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(assert (member (ACTOR "Members") C1a))
(assert (member (ACTOR "Books") C1a))
(assert (member (ACTOR "Language-tapes") C1a))
(declare-const C1 Clause)
(assert (= C1 (CLAUSE "exists" C1a )))
(assert (instantiate-clause C1))

(declare-const C2a (Set Actor))
(assert (member (ACTOR "Library") C2a))
(declare-const C2 Clause)
(assert (= C2 (CLAUSE "loans loan-item" C2a )))
(assert (instantiate-clause C2))

(declare-const C3a (Set Actor))
(assert (member (ACTOR "Library") C3a))
(declare-const C3 Clause)
(assert (= C3 (CLAUSE "issues membership-card" C3a )))
(assert (instantiate-clause C3))

(assert (= (card actors) 6))
(assert (= actors (as univset (Set Actor))))

(assert (= (card clauses) 3))
(assert (= clauses (as univset (Set Clause))))



(assert (member (STRUCT "Items" "must" "have" "bar-code") struct_sents))
(assert (member (TYPE_S "Books" "are" "items") type_sents))
(assert (member (STRUCT "Books" "must" "have" "title") struct_sents))
(assert (member (STRUCT "Books" "must" "have" "author") struct_sents))
(assert (member (TYPE_S "Tapes" "are" "items") type_sents))
(assert (member (STRUCT "Tapes" "must" "have" "language") struct_sents))
(assert (member (STRUCT "tapes" "must" "have" "level") struct_sents))
```

```
(assert (= (card struct_sents) 5))
(assert (= struct_sents (as univset (Set Structural_S))))

(assert (= (card type_sents) 2))
(assert (= type_sents (as univset (Set Type_S))))



; Shadow actors for inverse actors rule
(assert (= actors (as univset (Set Actor))))


;---------------------------------
; Class model
;---------------------------------

; Classes
(assert (member (CLASS "Customers" "classifier") classes))
(assert (member (CLASS "Members" "classifier") classes))
(assert (member (CLASS "Books" "classifier") classes))
(assert (member (CLASS "Language-tapes" "classifier") classes))
(assert (member (CLASS "Library" "classifier") classes))
(assert (member (CLASS "Items" "classifier") classes))
(assert (member (CLASS "Tapes" "classifier") classes))
(assert (= (card classes) 7))
(assert (= classes (as univset (Set Class))))

; Operations
(assert (member (OPR "Customers" "exists" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Members" "exists" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Language-tapes" "exists" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Library" "loans loan-item" "" "" "" (PARAM "loan-item" ""))
↪  operations))
(assert (= (card operations) 4))

; Attributes
(assert (member (ATR "Books" "author" "" "" "") attributes))
(assert (member (ATR "Items" "bar-code" "" "" "") attributes))
(assert (member (ATR "Tapes" "language" "" "" "") attributes))
(assert (member (ATR "Tapes" "level" "" "" "") attributes))
(assert (= (card attributes) 4))

;Inheritances
(assert (member (INH "items" "Books") inheritances))
(assert (member (INH "items" "Tapes") inheritances))
(assert (= (card inheritances) 2))

;Relations
(assert (= (card relations) 0))

;---------------------------------
; Validating model
;---------------------------------
(declare-const classes_validation Bool)
(declare-const operations_validation Bool)
(declare-const attributes_validation Bool)
(declare-const inheritances_validation Bool)
(declare-const relations_validation Bool)
(assert (= classes_validation actors_rule))
(assert (= operations_validation operations_rule))
```

```
(assert (= attributes_validation attributes_rule))
(assert (= inheritances_validation inheritances_rule))


;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (classes_validation))
(get-value (operations_validation))
(get-value (attributes_validation))
(get-value (inheritances_validation))
(exit)
```

C.2.0.3 Complete class model

JSON Model C.3: Full JSON of a complete but not sound model with respect to ConSpec 7.9

```
{
    "classes": {
        "randomclass": {
            "name": "RandomClass",
            "class_type": "class",
            "attributes": {
                "randomattribute": {
                    "name": "randomAttribute"
                }
            },
            "operations": {}
        },
        "customers": {
            "name": "Customers",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "members": {
            "name": "Members",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "books": {
            "name": "Books",
            "class_type": "class",
            "attributes": {
                "title": {
                    "name": "title"
```

```
                },
                "author": {
                    "name": "author"
                }
            },
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "language-tapes": {
            "name": "Language-tapes",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "library": {
            "name": "Library",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "loans-epsilon-loan-item-epsilon": {
                    "name": "loans",
                    "parameters": {
                        "loan-item-epsilon": {
                            "name": "loan-item"
                        }
                    }
                }
            }
        },
        "items": {
            "name": "Items",
            "class_type": "class",
            "attributes": {
                "bar-code": {
                    "name": "bar-code"
                }
            },
            "operations": {}
        },
        "tapes": {
            "name": "Tapes",
            "class_type": "class",
            "attributes": {
                "language": {
                    "name": "language"
                },
                "level": {
                    "name": "level"
                }
            },
            "operations": {}
        }
    },
```

```
    "associations": {
        "items-books-inheritance": {
            "source_class_name": "items",
            "destination_class_name": "Books",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        },
        "items-tapes-inheritance": {
            "source_class_name": "items",
            "destination_class_name": "Tapes",
            "type": "inheritance",
            "name": "",
            "role": "",
            "lower_cardinality": 1,
            "upper_cardinality": 1
        }
    }
}
```

SMTLib C.4: Full SMT-LIB model to check a complete but not sound class model (JSON Model C.3) against ConSpec 7.9

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;----------------------------------
; Datatypes
;----------------------------------

; Synonyms
(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                          (synonyms (Set String))
                                        ))))

; Specification
(declare-datatypes ((Actor 0)) (((ACTOR (actor_name String)))))

(declare-datatypes ((Clause 0))
    (((CLAUSE
        (activity String)
        (c_actors (Set Actor))
        ))))

(declare-datatypes ((Sentence 0))
    (((SENTENCE
        (words (Set String))
        ))))

(declare-datatypes ((Structural_S 0))
    (((STRUCT
        (entity String)
        (modal String)
        (have String)
        (property String)
        ))))

(declare-datatypes ((Type_S 0))
```

```
    (((TYPE_S
        (subtype String)
        (isa String)
        (type String)
        ))))

; Class Diagram
(declare-datatypes ( (Class     0) ) ( ((CLASS
                                        (cls_nme String)
                                        (cls_typ String)
                                    ))))
(declare-datatypes ( (Type      0) ) ( ((TYPE
                                        (typ_nme String)
                                    ))))
(declare-datatypes ( (Attribute 0) ) ( ((ATR
                                        (atr_cls String)
                                        (atr_nme String)
                                        (atr_typ String)
                                        (atr_vis String)
                                        (atr_sco String)
                                    ))))
(declare-datatypes ( (Param     0) ) ( ((PARAM
                                        (prm_nme String)
                                        (prm_typ String)
                                    )
                                    (nil))))
(declare-datatypes ( (Operation 0) ) ( ((OPR
                                        (opr_cls String)
                                        (opr_nme String)
                                        (opr_typ String)
                                        (opr_vis String)
                                        (opr_sco String)
                                        (opr_prm Param)
                                    ))))
(declare-datatypes ( (Relation 0) ) ( ((REL
                                        (rel_src String)
                                        (rel_des String)
                                        (rel_typ String)
                                        (rel_nme String)
                                        (rel_rol String)
                                        (rel_c_l String)
                                        (rel_c_u String)
                                    ))))
(declare-datatypes ( (Inheritance     0) ) ( ((INH
                                        (inh_sup String)
                                        (inh_sub String)
                                    ))))


;---------------------------------
; Sets
;---------------------------------

; Specification
(declare-const actors (Set Actor))
(declare-const clauses (Set Clause))
(declare-const sentences (Set Sentence))
(declare-const struct_sents (Set Structural_S))
(declare-const type_sents (Set Type_S))

; Class Diagram
(declare-const classes (Set Class))
```

```
(declare-const types (Set Type))
(declare-const attributes (Set Attribute))
(declare-const operations (Set Operation))
(declare-const relations (Set Relation))
(declare-const inheritances (Set Inheritance))


;--------------------------------
; Synonyms
;--------------------------------
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Paper" "paper" "PAPER" "Papers" "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert
    "title"   "TITLE"   "Title"   "titles"   "statute_title"   "STATUTE_TITLE"
    ↪ "StatuteTitle"   "statuteTitle"   "statute-title"   "statute_titles"     "Statute
    ↪ Title"   "rubric"   "RUBRIC"   "Rubric"   "rubrics"
    (singleton "title"))) syns_dict))

(assert (member (mk-entry (insert
    "word-count"   "WORD-COUNT"   "Word-count"   "word_count"   "word-counts"
    "word_counts"   "Word Count"
    (singleton "word-count"))) syns_dict))

(assert (member (mk-entry (insert
    "student-indicator"   "STUDENT-INDICATOR"   "Student-indicator"
    "student_indicator"   "student-indicators"   "student_indicators"
    "Student Indicator"
    (singleton "student-indicator"))) syns_dict))

(assert (member (mk-entry (insert
    "authors"   "AUTHORS"   "Authors"   "author"   "writer"   "WRITER"
    "Writer"   "writers"   "AUTHOR"   "Author"
    (singleton "authors"))) syns_dict))

(assert (member (mk-entry (insert
    "referees"   "REFEREES"   "Referees"   "referee"   "REFEREE"   "Referee"
    "ref"   "REF"   "Ref"   "refs"
    (singleton "referees"))) syns_dict))

(assert (member (mk-entry (insert
    "Researcher"   "researcher"   "RESEARCHER"   "Researchers"   "researchers"
    "research_worker"   "RESEARCH_WORKER"   "ResearchWorker"   "researchWorker"
    "research-worker"   "research_workers"   "Research Worker"   "investigator"
    "INVESTIGATOR"   "Investigator"   "investigators"
    (singleton "Researcher"))) syns_dict))

(assert (member (mk-entry (insert
    "name"   "NAME"   "Name"   "names"
    (singleton "name"))) syns_dict))


(assert (= syns_dict (as univset (Set Entry))))

;--------------------------------
; Constructors and helpers
;--------------------------------

(define-fun is_syn ((w1 String) (w2 String)) Bool
```

```
    (or
        (= w1 w2)
        (exists
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))


(define-fun instantiate-clause ((c Clause)) Bool
    (and
        (member c clauses)
        (forall
            ((a Actor))
            (=>
                (member a (c_actors c))
                (member a actors)
            )
        )
    )
)

;---------------------------------
; Inference rules
;---------------------------------

(define-fun check_actor ((a Actor)) Bool
    (exists
        ((c Class))
        (and
            (member c classes)
            (is_syn
                (actor_name a)
                (cls_nme c)
            )
        )
    )
)

(define-fun inverse_actors_rule () Bool
    (forall
        ((a Actor))
        (=>
            (member a actors)
            (check_actor a)
        )
    )
)

(define-fun check_activity ((a String)) Bool
    (exists
        ((o Operation))
        (and
            (member o operations)
            (is_syn
                a
                (opr_nme o)
            )
        )
    )
```

```
)

(define-fun inverese_operations_rule () Bool
    (forall
        ((c Clause))
        (=>
            (member c clauses)
            (check_activity (activity c))
        )
    )
)

(define-fun check_structures ((s Structural_S)) Bool
    (exists
        ((a Attribute))
        (and
            (member a attributes)
            (is_syn
                (property s)
                (atr_nme a)
            )
        )
    )
)

(define-fun inverese_attributes_rule () Bool
    (forall
        ((s Structural_S))
        (=>
            (member s struct_sents)
            (check_structures s)
        )
    )
)

(define-fun check_types ((s Type_S)) Bool
    (exists
        ((h Inheritance))
        (and
            (member h inheritances)
            (is_syn (inh_sub h) (subtype s))
            (is_syn (inh_sup h) (type s))
        )
    )
)

(define-fun inverese_inheritances_rule () Bool
    (forall
        ((s Type_S))
        (=>
            (member s type_sents)
            (check_types s)
        )
    )
)

(define-fun check_relation_sentences ((s Sentence)) Bool
    (exists
        ((r Relation))
        (and
            (member r relations)
            (member (rel_des r) (words s))
```

```
            (member (rel_src r) (words s))
        )
    )
)

(define-fun inverse_relations_rule () Bool
    (forall
        ((s Sentence))
        (=>
            (member s sentences)
            (check_relation_sentences s)
        )
    )
)

;--------------------------------
; Specification
;--------------------------------

(declare-const C1a (Set Actor))
(assert (member (ACTOR "Customers") C1a))
(assert (member (ACTOR "Members") C1a))
(assert (member (ACTOR "Books") C1a))
(assert (member (ACTOR "Language-tapes") C1a))
(declare-const C1 Clause)
(assert (= C1 (CLAUSE "exists" C1a )))
(assert (instantiate-clause C1))

(declare-const C2a (Set Actor))
(assert (member (ACTOR "Library") C2a))
(declare-const C2 Clause)
(assert (= C2 (CLAUSE "loans loan-item" C2a )))
(assert (instantiate-clause C2))

(declare-const C3a (Set Actor))
(assert (member (ACTOR "Library") C3a))
(declare-const C3 Clause)
(assert (= C3 (CLAUSE "issues membership-card" C3a )))
(assert (instantiate-clause C3))

(assert (= (card actors) 6))
(assert (= actors (as univset (Set Actor))))

(assert (= (card clauses) 3))
(assert (= clauses (as univset (Set Clause))))


(assert (member (STRUCT "Items" "must" "have" "bar-code") struct_sents))
(assert (member (TYPE_S "Books" "are" "items") type_sents))
(assert (member (STRUCT "Books" "must" "have" "title") struct_sents))
(assert (member (STRUCT "Books" "must" "have" "author") struct_sents))
(assert (member (TYPE_S "Tapes" "are" "items") type_sents))
(assert (member (STRUCT "Tapes" "must" "have" "language") struct_sents))
(assert (member (STRUCT "tapes" "must" "have" "level") struct_sents))

(assert (= (card struct_sents) 5))
(assert (= struct_sents (as univset (Set Structural_S))))

(assert (= (card type_sents) 2))
(assert (= type_sents (as univset (Set Type_S))))
```

```
;£fill: type sentences

;---------------------------------
; Class model
;---------------------------------

; Classes
(assert (member (CLASS "RandomClass" "classifier") classes))
(assert (member (CLASS "Customers" "classifier") classes))
(assert (member (CLASS "Members" "classifier") classes))
(assert (member (CLASS "Books" "classifier") classes))
(assert (member (CLASS "Language-tapes" "classifier") classes))
(assert (member (CLASS "Library" "classifier") classes))
(assert (member (CLASS "Items" "classifier") classes))
(assert (member (CLASS "Tapes" "classifier") classes))
(assert (= (card classes) 8))
(assert (= classes (as univset (Set Class))))

; Operations
(assert (member (OPR "Customers" "exists" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Members" "exists" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Books" "exists" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Language-tapes" "exists" "" "" "" (PARAM "" "")) operations))
(assert (member (OPR "Library" "loans loan-item" "" "" "" (PARAM "loan-item" ""))
↪  operations))
(assert (= (card operations) 5))
(assert (= operations (as univset (Set Operation))))

; Attributes
(assert (member (ATR "RandomClass" "randomAttribute" "" "" "") attributes))
(assert (member (ATR "Books" "title" "" "" "") attributes))
(assert (member (ATR "Books" "author" "" "" "") attributes))
(assert (member (ATR "Items" "bar-code" "" "" "") attributes))
(assert (member (ATR "Tapes" "language" "" "" "") attributes))
(assert (member (ATR "Tapes" "level" "" "" "") attributes))
(assert (= (card attributes) 6))
(assert (= attributes (as univset (Set Attribute))))

;Inheritances
(assert (member (INH "items" "Books") inheritances))
(assert (member (INH "items" "Tapes") inheritances))
(assert (= (card inheritances) 2))
(assert (= inheritances (as univset (Set Inheritance))))

;Relations
(assert (= (card relations) 0))
(assert (= relations (as univset (Set Relation))))


;---------------------------------
; Validating model
;---------------------------------

(declare-const inverse_classes_validation Bool)
(declare-const inverse_operations_validation Bool)
(declare-const inverse_attributes_validation Bool)
(declare-const inverse_inheritances_validation Bool)
(declare-const inverse_relations_validation Bool)
(assert (= inverse_classes_validation inverse_actors_rule))
(assert (= inverse_operations_validation inverese_operations_rule))
(assert (= inverse_attributes_validation inverese_attributes_rule))
(assert (= inverse_inheritances_validation inverese_inheritances_rule))
```

```
;---------------------------------
; Checks
;---------------------------------
(check-sat)
(get-value (inverse_classes_validation))
(get-value (inverse_operations_validation))
(get-value (inverse_attributes_validation))
(get-value (inverse_inheritances_validation))
(exit)
```

C.3 Model Comparison

C.3.0.1 Not equivalent class models

SMTLib C.5: Full SMT-LIB model to check not equivalent models

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;---------------------------------
; Datatypes and structures
;---------------------------------


(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                        (synonyms (Set String))
                                    ))))


; UML datatypes
(declare-datatypes ( (Class     0) ) ( ((CLASS
    (cls_nme String) (cls_typ String) ))))

(declare-datatypes ( (Type      0) ) ( ((TYPE
    (typ_nme String) ))))

(declare-datatypes ( (Attribute 0) ) ( ((ATR
    (atr_cls String) (atr_nme String) (atr_typ String)
    (atr_vis String) (atr_sco String) ))))

(declare-datatypes ( (Param     0) ) ( ((PARAM
    (prm_nme String) (prm_typ String)) (nil) )))

(declare-datatypes ( (Operation 0) ) ( ((OPR
    (opr_cls String) (opr_nme String) (opr_typ String)
    (opr_vis String) (opr_sco String) (opr_prm Param)))))

(declare-datatypes ( (Relation 0) ) ( ((REL
    (rel_src String) (rel_des String) (rel_typ String)
    (rel_nme String) (rel_rol String) (rel_c_l String)
    (rel_c_u String) ))))

(declare-datatypes ( (Inheritance     0) ) ( ((INH
    (inh_sup String) (inh_sub String) ))))
```

```
(declare-datatypes ( (ClassDiagram     0) ) ( ((CD
    (dictionary (Set String)) (classes (Set Class))
    (types (Set Type)) (attributes (Set Attribute))
    (operations (Set Operation)) (relations (Set Relation))
    (inheritance (Set Inheritance)) ))))



;---------------------------------
; Synonyms
;---------------------------------
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Paper" "paper" "PAPER" "Papers" "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert
    "title"   "TITLE"   "Title"   "titles"   "statute_title"   "STATUTE_TITLE"
    ↪  "StatuteTitle"   "statuteTitle"   "statute-title"   "statute_titles"     "Statute
    ↪  Title"   "rubric"   "RUBRIC"   "Rubric"   "rubrics"
    (singleton "title"))) syns_dict))

(assert (member (mk-entry (insert
    "word-count"   "WORD-COUNT"   "Word-count"   "word_count"   "word-counts"
    "word_counts"   "Word Count"
    (singleton "word-count"))) syns_dict))

(assert (member (mk-entry (insert
    "student-indicator"   "STUDENT-INDICATOR"   "Student-indicator"
    "student_indicator"   "student-indicators"   "student_indicators"
    "Student Indicator"
    (singleton "student-indicator"))) syns_dict))

(assert (member (mk-entry (insert
    "authors"   "AUTHORS"   "Authors"   "author"   "writer"   "WRITER"
    "Writer"   "writers"   "AUTHOR"   "Author"
    (singleton "authors"))) syns_dict))

(assert (member (mk-entry (insert
    "referees"   "REFEREES"   "Referees"   "referee"   "REFEREE"   "Referee"
    "ref"   "REF"   "Ref"   "refs"
    (singleton "referees"))) syns_dict))

(assert (member (mk-entry (insert
    "Researcher"   "researcher"   "RESEARCHER"   "Researchers"   "researchers"
    "research_worker"   "RESEARCH_WORKER"   "ResearchWorker"   "researchWorker"
    "research-worker"   "research_workers"   "Research Worker"   "investigator"
    "INVESTIGATOR"   "Investigator"   "investigators"
    (singleton "Researcher"))) syns_dict))

(assert (member (mk-entry (insert
    "name"   "NAME"   "Name"   "names"
    (singleton "name"))) syns_dict))


(assert (= syns_dict (as univset (Set Entry))))

;---------------------------------
; Constructors and helpers
;---------------------------------
```

```
(define-fun is_syn ((w1 String) (w2 String)) Bool
    (or
        (= w1 w2)
        (exists
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))

;---------------------------------
; Inference rules
;---------------------------------

(define-fun check_dictionary ((s1 (Set String)) (s2 (Set String))) Bool
    (forall
        ((x String))
        (=>
            (member x s1)
            (exists
                ((y String))
                (and
                    (member y s2)
                    (is_syn x y)
                )
            )
        )
    )
)


(define-fun check_classes ((s1 (Set Class)) (s2 (Set Class))) Bool
    (forall
        ((x Class))
        (=>
            (member x s1)
            (exists
                ((y Class))
                (and
                    (member y s2)
                    (is_syn (cls_nme x) (cls_nme y))
                )
            )
        )
    )
)

(define-fun check_types ((s1 (Set Type)) (s2 (Set Type))) Bool
    (forall
        ((x Type))
        (=>
            (member x s1)
            (exists
                ((y Type))
                (and
                    (member y s2)
                    (is_syn (typ_nme x) (typ_nme y))
                )
            )
        )
    )
```

```
)

(define-fun check_attributes ((s1 (Set Attribute)) (s2 (Set Attribute))) Bool
    (forall
        ((x Attribute))
        (=>
            (member x s1)
            (exists
                ((y Attribute))
                (and
                    (member y s2)
                    (is_syn (atr_cls x) (atr_cls y))
                    (is_syn (atr_nme x) (atr_nme y))
                )
            )
        )
    )
)

(define-fun check_operations ((s1 (Set Operation)) (s2 (Set Operation))) Bool
    (forall
        ((x Operation))
        (=>
            (member x s1)
            (exists
                ((y Operation))
                (and
                    (member y s2)
                    (is_syn (opr_nme x) (opr_nme y))
                )
            )
        )
    )
)

(define-fun check_relations ((s1 (Set Relation)) (s2 (Set Relation))) Bool
    (forall
        ((x Relation))
        (=>
            (member x s1)
            (exists
                ((y Relation))
                (and
                    (member y s2)
                    (is_syn (rel_src x) (rel_src y))
                    (is_syn (rel_des x) (rel_des y))
                )
            )
        )
    )
)


(define-fun check_inheritances ((s1 (Set Inheritance)) (s2 (Set Inheritance))) Bool
    (forall
        ((x Inheritance))
        (=>
            (member x s1)
            (exists
                ((y Inheritance))
                (and
                    (member y s2)
```

```
                    (is_syn (inh_sup x) (inh_sup y))
                    (is_syn (inh_sub x) (inh_sub y))
                )
            )
        )
    )
)


;---------------------------------
; Class Diagram 1
;---------------------------------
;Dictionary 1
(declare-const dictionary1 (Set String))
;£fill: dictionary 1

;Classes 1
(declare-const classes1 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes1))
(assert (member (CLASS "Researcher" "classifier") classes1))
(assert (= (card classes1) 2))

;Types 1
(declare-const types1 (Set Type))
;£fill: types 1

;Attributes 1
(declare-const attributes1 (Set Attribute))
(assert (member (ATR "Paper" "title" "" "" "") attributes1))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes1))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes1))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes1))(assert (member (ATR "Paper" "referees" "" "" "") attributes1))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes1))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes1))(assert (= (card attributes1)
↪  7))

;Operations 1
(declare-const operations1 (Set Operation))
(assert (= (card operations1) 0))

;Relations 1
(declare-const relations1 (Set Relation))
(assert (= (card relations1) 0))

;Inheritances 1
(declare-const inheritances1 (Set Inheritance))
(assert (= (card inheritances1) 0))

;Class Diagram 1
(declare-const diagram1 ClassDiagram)
(assert (= diagram1
    (CD dictionary1 classes1 types1 attributes1 operations1 relations1 inheritances1)))


;---------------------------------
; Class Diagram 2
;---------------------------------
;Dictionary 2
(declare-const dictionary2 (Set String))
;£fill: dictionary 2

;Classes 2
```

```
(declare-const classes2 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes2))
(assert (member (CLASS "Researcher" "classifier") classes2))
(assert (= (card classes2) 2))

;Types 2
(declare-const types2 (Set Type))
;£fill: types 2

;Attributes 2
(declare-const attributes2 (Set Attribute))
(assert (member (ATR "Paper" "title" "" "" "") attributes2))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes2))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes2))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes2))(assert (member (ATR "Paper" "referees" "" "" "") attributes2))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes2))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes2))(assert (= (card attributes2)
↪  7))

;Operations 2
(declare-const operations2 (Set Operation))
(assert (= (card operations2) 0))

;Relations 2
(declare-const relations2 (Set Relation))
(assert (= (card relations2) 0))

;Inheritances 2
(declare-const inheritances2 (Set Inheritance))
(assert (= (card inheritances2) 0))

;Class Diagram 2
(declare-const diagram2 ClassDiagram)
(assert (= diagram2
    (CD dictionary2 classes2 types2 attributes2 operations2 relations2 inheritances2)))


;-----------------------------------
; Checks
;-----------------------------------
(declare-const equiv_dicts Bool)
(declare-const equiv_classes Bool)
(declare-const equiv_types Bool)
(declare-const equiv_attributes Bool)
(declare-const equiv_operations Bool)
(declare-const equiv_relations Bool)
(declare-const equiv_inheritances Bool)
(assert (= equiv_classes (check_classes classes1 classes2)))
(assert (= equiv_attributes (check_attributes attributes1 attributes2)))
(assert (= equiv_operations (check_operations operations1 operations2)))
(assert (= equiv_inheritances (check_inheritances inheritances1 inheritances2)))

(define-fun equiv_diagrams () Bool
    (and
        equiv_classes
        equiv_attributes
        equiv_operations
        equiv_inheritances
    )
)

(declare-const are_equivalent Bool)
```

```
(assert (= equiv_diagrams are_equivalent))

(check-sat)
(get-value (equiv_classes))
(get-value (equiv_attributes))
(get-value (equiv_operations))
(get-value (equiv_inheritances))
(get-value (are_equivalent))

(exit)
```

C.3.0.2 Left equivalent class models

SMTLib C.6: Full SMT-LIB model to check left equivalent models

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;----------------------------------
; Datatypes and structures
;----------------------------------


(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                        (synonyms (Set String))
                                    ))))


; UML datatypes
(declare-datatypes ( (Class     0) ) ( ((CLASS
    (cls_nme String) (cls_typ String) ))))

(declare-datatypes ( (Type      0) ) ( ((TYPE
    (typ_nme String) ))))

(declare-datatypes ( (Attribute 0) ) ( ((ATR
    (atr_cls String) (atr_nme String) (atr_typ String)
    (atr_vis String) (atr_sco String) ))))

(declare-datatypes ( (Param     0) ) ( ((PARAM
    (prm_nme String) (prm_typ String)) (nil) )))

(declare-datatypes ( (Operation 0) ) ( ((OPR
    (opr_cls String) (opr_nme String) (opr_typ String)
    (opr_vis String) (opr_sco String) (opr_prm Param)))))

(declare-datatypes ( (Relation 0) ) ( ((REL
    (rel_src String) (rel_des String) (rel_typ String)
    (rel_nme String) (rel_rol String) (rel_c_l String)
    (rel_c_u String) ))))

(declare-datatypes ( (Inheritance     0) ) ( ((INH
    (inh_sup String) (inh_sub String) ))))

(declare-datatypes ( (ClassDiagram     0) ) ( ((CD
    (dictionary (Set String)) (classes (Set Class))
    (types (Set Type)) (attributes (Set Attribute))
    (operations (Set Operation)) (relations (Set Relation))
    (inheritance (Set Inheritance)) ))))
```

```
;--------------------------------
; Synonyms
;--------------------------------
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Paper" "paper" "PAPER" "Papers" "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert
    "title"   "TITLE"   "Title"   "titles"   "statute_title"   "STATUTE_TITLE"
    ↪  "StatuteTitle"   "statuteTitle"   "statute-title"   "statute_titles"     "Statute
    ↪  Title"   "rubric"   "RUBRIC"   "Rubric"   "rubrics"
    (singleton "title"))) syns_dict))

(assert (member (mk-entry (insert
    "word-count"   "WORD-COUNT"   "Word-count"   "word_count"   "word-counts"
    "word_counts"   "Word Count"
    (singleton "word-count"))) syns_dict))

(assert (member (mk-entry (insert
    "student-indicator"   "STUDENT-INDICATOR"   "Student-indicator"
    "student_indicator"   "student-indicators"   "student_indicators"
    "Student Indicator"
    (singleton "student-indicator"))) syns_dict))

(assert (member (mk-entry (insert
    "authors"   "AUTHORS"   "Authors"   "author"   "writer"   "WRITER"
    "Writer"   "writers"   "AUTHOR"   "Author"
    (singleton "authors"))) syns_dict))

(assert (member (mk-entry (insert
    "referees"   "REFEREES"   "Referees"   "referee"   "REFEREE"   "Referee"
    "ref"   "REF"   "Ref"   "refs"
    (singleton "referees"))) syns_dict))

(assert (member (mk-entry (insert
    "Researcher"   "researcher"   "RESEARCHER"   "Researchers"   "researchers"
    "research_worker"   "RESEARCH_WORKER"   "ResearchWorker"   "researchWorker"
    "research-worker"   "research_workers"   "Research Worker"   "investigator"
    "INVESTIGATOR"   "Investigator"   "investigators"
    (singleton "Researcher"))) syns_dict))

(assert (member (mk-entry (insert
    "name"   "NAME"   "Name"   "names"
    (singleton "name"))) syns_dict))


(assert (= syns_dict (as univset (Set Entry))))

;--------------------------------
; Constructors and helpers
;--------------------------------

(define-fun is_syn ((w1 String) (w2 String)) Bool
    (or
        (= w1 w2)
        (exists
            ((e Entry))
```

```
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))

;--------------------------------
; Inference rules
;--------------------------------

(define-fun check_dictionary ((s1 (Set String)) (s2 (Set String))) Bool
    (forall
        ((x String))
        (=>
            (member x s1)
            (exists
                ((y String))
                (and
                    (member y s2)
                    (is_syn x y)
                )
            )
        )
    )
)


(define-fun check_classes ((s1 (Set Class)) (s2 (Set Class))) Bool
    (forall
        ((x Class))
        (=>
            (member x s1)
            (exists
                ((y Class))
                (and
                    (member y s2)
                    (is_syn (cls_nme x) (cls_nme y))
                )
            )
        )
    )
)

(define-fun check_types ((s1 (Set Type)) (s2 (Set Type))) Bool
    (forall
        ((x Type))
        (=>
            (member x s1)
            (exists
                ((y Type))
                (and
                    (member y s2)
                    (is_syn (typ_nme x) (typ_nme y))
                )
            )
        )
    )
)

(define-fun check_attributes ((s1 (Set Attribute)) (s2 (Set Attribute))) Bool
    (forall
        ((x Attribute))
```

```
        (=>
            (member x s1)
            (exists
                ((y Attribute))
                (and
                    (member y s2)
                    (is_syn (atr_cls x) (atr_cls y))
                    (is_syn (atr_nme x) (atr_nme y))
                )
            )
        )
    )
)

(define-fun check_operations ((s1 (Set Operation)) (s2 (Set Operation))) Bool
    (forall
        ((x Operation))
        (=>
            (member x s1)
            (exists
                ((y Operation))
                (and
                    (member y s2)
                    (is_syn (opr_nme x) (opr_nme y))
                )
            )
        )
    )
)

(define-fun check_relations ((s1 (Set Relation)) (s2 (Set Relation))) Bool
    (forall
        ((x Relation))
        (=>
            (member x s1)
            (exists
                ((y Relation))
                (and
                    (member y s2)
                    (is_syn (rel_src x) (rel_src y))
                    (is_syn (rel_des x) (rel_des y))
                )
            )
        )
    )
)


(define-fun check_inheritances ((s1 (Set Inheritance)) (s2 (Set Inheritance))) Bool
    (forall
        ((x Inheritance))
        (=>
            (member x s1)
            (exists
                ((y Inheritance))
                (and
                    (member y s2)
                    (is_syn (inh_sup x) (inh_sup y))
                    (is_syn (inh_sub x) (inh_sub y))
                )
            )
        )
```

```
    )
)


;---------------------------------
; Class Diagram 1
;---------------------------------
;Dictionary 1
(declare-const dictionary1 (Set String))
;£fill: dictionary 1

;Classes 1
(declare-const classes1 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes1))
(assert (member (CLASS "Researcher" "classifier") classes1))
(assert (= (card classes1) 2))

;Types 1
(declare-const types1 (Set Type))
;£fill: types 1

;Attributes 1
(declare-const attributes1 (Set Attribute))
(assert (member (ATR "Paper" "title" "" "" "") attributes1))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes1))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes1))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes1))(assert (member (ATR "Paper" "referees" "" "" "") attributes1))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes1))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes1))(assert (= (card attributes1)
↪  7))

;Operations 1
(declare-const operations1 (Set Operation))
(assert (= (card operations1) 0))

;Relations 1
(declare-const relations1 (Set Relation))
(assert (= (card relations1) 0))

;Inheritances 1
(declare-const inheritances1 (Set Inheritance))
(assert (= (card inheritances1) 0))

;Class Diagram 1
(declare-const diagram1 ClassDiagram)
(assert (= diagram1
    (CD dictionary1 classes1 types1 attributes1 operations1 relations1 inheritances1)))


;---------------------------------
; Class Diagram 2
;---------------------------------
;Dictionary 2
(declare-const dictionary2 (Set String))
;£fill: dictionary 2

;Classes 2
(declare-const classes2 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes2))
(assert (member (CLASS "Researcher" "classifier") classes2))
(assert (= (card classes2) 2))
```

```
;Types 2
(declare-const types2 (Set Type))
;£fill: types 2

;Attributes 2
(declare-const attributes2 (Set Attribute))
(assert (member (ATR "Paper" "title" "" "" "") attributes2))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes2))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes2))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes2))(assert (member (ATR "Paper" "referees" "" "" "") attributes2))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes2))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes2))(assert (= (card attributes2)
↪  7))

;Operations 2
(declare-const operations2 (Set Operation))
(assert (= (card operations2) 0))

;Relations 2
(declare-const relations2 (Set Relation))
(assert (= (card relations2) 0))

;Inheritances 2
(declare-const inheritances2 (Set Inheritance))
(assert (= (card inheritances2) 0))

;Class Diagram 2
(declare-const diagram2 ClassDiagram)
(assert (= diagram2
    (CD dictionary2 classes2 types2 attributes2 operations2 relations2 inheritances2)))


;---------------------------------
; Checks
;---------------------------------
(declare-const equiv_dicts Bool)
(declare-const equiv_classes Bool)
(declare-const equiv_types Bool)
(declare-const equiv_attributes Bool)
(declare-const equiv_operations Bool)
(declare-const equiv_relations Bool)
(declare-const equiv_inheritances Bool)
(assert (= equiv_classes (check_classes classes1 classes2)))
(assert (= equiv_attributes (check_attributes attributes1 attributes2)))
(assert (= equiv_operations (check_operations operations1 operations2)))
(assert (= equiv_inheritances (check_inheritances inheritances1 inheritances2)))

(define-fun equiv_diagrams () Bool
    (and
        equiv_classes
        equiv_attributes
        equiv_operations
        equiv_inheritances
    )
)

(declare-const are_equivalent Bool)
(assert (= equiv_diagrams are_equivalent))

(check-sat)
(get-value (equiv_classes))
(get-value (equiv_attributes))
```

```
(get-value (equiv_operations))
(get-value (equiv_inheritances))
(get-value (are_equivalent))

(exit)
```

C.3.0.3 Right equivalent class models

SMTLib C.7: Full SMT-LIB model to check right equivalent models

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;----------------------------------
; Datatypes and structures
;----------------------------------


(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                        (synonyms (Set String))
                                    ))))


; UML datatypes
(declare-datatypes ( (Class     0) ) ( ((CLASS
    (cls_nme String) (cls_typ String) ))))

(declare-datatypes ( (Type      0) ) ( ((TYPE
    (typ_nme String) ))))

(declare-datatypes ( (Attribute 0) ) ( ((ATR
    (atr_cls String) (atr_nme String) (atr_typ String)
    (atr_vis String) (atr_sco String) ))))

(declare-datatypes ( (Param     0) ) ( ((PARAM
    (prm_nme String) (prm_typ String)) (nil) )))

(declare-datatypes ( (Operation 0) ) ( ((OPR
    (opr_cls String) (opr_nme String) (opr_typ String)
    (opr_vis String) (opr_sco String) (opr_prm Param)))))

(declare-datatypes ( (Relation 0) ) ( ((REL
    (rel_src String) (rel_des String) (rel_typ String)
    (rel_nme String) (rel_rol String) (rel_c_l String)
    (rel_c_u String) ))))

(declare-datatypes ( (Inheritance     0) ) ( ((INH
    (inh_sup String) (inh_sub String) ))))

(declare-datatypes ( (ClassDiagram     0) ) ( ((CD
    (dictionary (Set String)) (classes (Set Class))
    (types (Set Type)) (attributes (Set Attribute))
    (operations (Set Operation)) (relations (Set Relation))
    (inheritance (Set Inheritance)) ))))

;----------------------------------
; Synonyms
;----------------------------------
(declare-const syns_dict (Set Entry))
```

```
(assert (member (mk-entry (insert
    "Paper" "paper" "PAPER" "Papers" "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert
    "title"   "TITLE"   "Title"   "titles"   "statute_title"   "STATUTE_TITLE"
    ↪  "StatuteTitle"   "statuteTitle"   "statute-title"   "statute_titles"     "Statute
    ↪  Title"   "rubric"   "RUBRIC"   "Rubric"   "rubrics"
    (singleton "title"))) syns_dict))

(assert (member (mk-entry (insert
    "word-count"   "WORD-COUNT"   "Word-count"   "word_count"   "word-counts"
    "word_counts"   "Word Count"
    (singleton "word-count"))) syns_dict))

(assert (member (mk-entry (insert
    "student-indicator"   "STUDENT-INDICATOR"   "Student-indicator"
    "student_indicator"   "student-indicators"   "student_indicators"
    "Student Indicator"
    (singleton "student-indicator"))) syns_dict))

(assert (member (mk-entry (insert
    "authors"   "AUTHORS"   "Authors"   "author"   "writer"   "WRITER"
    "Writer"   "writers"   "AUTHOR"   "Author"
    (singleton "authors"))) syns_dict))

(assert (member (mk-entry (insert
    "referees"   "REFEREES"   "Referees"   "referee"   "REFEREE"   "Referee"
    "ref"   "REF"   "Ref"   "refs"
    (singleton "referees"))) syns_dict))

(assert (member (mk-entry (insert
    "Researcher"   "researcher"   "RESEARCHER"   "Researchers"   "researchers"
    "research_worker"   "RESEARCH_WORKER"   "ResearchWorker"   "researchWorker"
    "research-worker"   "research_workers"   "Research Worker"   "investigator"
    "INVESTIGATOR"   "Investigator"   "investigators"
    (singleton "Researcher"))) syns_dict))

(assert (member (mk-entry (insert
    "name"   "NAME"   "Name"   "names"
    (singleton "name"))) syns_dict))


(assert (= syns_dict (as univset (Set Entry))))

;----------------------------------
; Constructors and helpers
;----------------------------------

(define-fun is_syn ((w1 String) (w2 String)) Bool
    (or
        (= w1 w2)
        (exists
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))

;----------------------------------
```

```
; Inference rules
;--------------------------------

(define-fun check_dictionary ((s1 (Set String)) (s2 (Set String))) Bool
    (forall
        ((x String))
        (=>
            (member x s1)
            (exists
                ((y String))
                (and
                    (member y s2)
                    (is_syn x y)
                )
            )
        )
    )
)


(define-fun check_classes ((s1 (Set Class)) (s2 (Set Class))) Bool
    (forall
        ((x Class))
        (=>
            (member x s1)
            (exists
                ((y Class))
                (and
                    (member y s2)
                    (is_syn (cls_nme x) (cls_nme y))
                )
            )
        )
    )
)

(define-fun check_types ((s1 (Set Type)) (s2 (Set Type))) Bool
    (forall
        ((x Type))
        (=>
            (member x s1)
            (exists
                ((y Type))
                (and
                    (member y s2)
                    (is_syn (typ_nme x) (typ_nme y))
                )
            )
        )
    )
)

(define-fun check_attributes ((s1 (Set Attribute)) (s2 (Set Attribute))) Bool
    (forall
        ((x Attribute))
        (=>
            (member x s1)
            (exists
                ((y Attribute))
                (and
                    (member y s2)
                    (is_syn (atr_cls x) (atr_cls y))
```

```
                    (is_syn (atr_nme x) (atr_nme y))
                )
            )
        )
    )
)

(define-fun check_operations ((s1 (Set Operation)) (s2 (Set Operation))) Bool
    (forall
        ((x Operation))
        (=>
            (member x s1)
            (exists
                ((y Operation))
                (and
                    (member y s2)
                    (is_syn (opr_nme x) (opr_nme y))
                )
            )
        )
    )
)

(define-fun check_relations ((s1 (Set Relation)) (s2 (Set Relation))) Bool
    (forall
        ((x Relation))
        (=>
            (member x s1)
            (exists
                ((y Relation))
                (and
                    (member y s2)
                    (is_syn (rel_src x) (rel_src y))
                    (is_syn (rel_des x) (rel_des y))
                )
            )
        )
    )
)


(define-fun check_inheritances ((s1 (Set Inheritance)) (s2 (Set Inheritance))) Bool
    (forall
        ((x Inheritance))
        (=>
            (member x s1)
            (exists
                ((y Inheritance))
                (and
                    (member y s2)
                    (is_syn (inh_sup x) (inh_sup y))
                    (is_syn (inh_sub x) (inh_sub y))
                )
            )
        )
    )
)


;---------------------------------
; Class Diagram 1
;---------------------------------
```

```
;Dictionary 1
(declare-const dictionary1 (Set String))
;£fill: dictionary 1

;Classes 1
(declare-const classes1 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes1))
(assert (member (CLASS "Researcher" "classifier") classes1))
(assert (= (card classes1) 2))

;Types 1
(declare-const types1 (Set Type))
;£fill: types 1

;Attributes 1
(declare-const attributes1 (Set Attribute))
(assert (member (ATR "Paper" "title" "" "" "") attributes1))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes1))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes1))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes1))(assert (member (ATR "Paper" "referees" "" "" "") attributes1))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes1))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes1))(assert (= (card attributes1)
↪  7))

;Operations 1
(declare-const operations1 (Set Operation))
(assert (= (card operations1) 0))

;Relations 1
(declare-const relations1 (Set Relation))
(assert (= (card relations1) 0))

;Inheritances 1
(declare-const inheritances1 (Set Inheritance))
(assert (= (card inheritances1) 0))

;Class Diagram 1
(declare-const diagram1 ClassDiagram)
(assert (= diagram1
    (CD dictionary1 classes1 types1 attributes1 operations1 relations1 inheritances1)))


;---------------------------------
; Class Diagram 2
;---------------------------------
;Dictionary 2
(declare-const dictionary2 (Set String))
;£fill: dictionary 2

;Classes 2
(declare-const classes2 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes2))
(assert (member (CLASS "Researcher" "classifier") classes2))
(assert (= (card classes2) 2))

;Types 2
(declare-const types2 (Set Type))
;£fill: types 2

;Attributes 2
(declare-const attributes2 (Set Attribute))
```

```
(assert (member (ATR "Paper" "title" "" "" "") attributes2))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes2))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes2))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes2))(assert (member (ATR "Paper" "referees" "" "" "") attributes2))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes2))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes2))(assert (= (card attributes2)
↪  7))

;Operations 2
(declare-const operations2 (Set Operation))
(assert (= (card operations2) 0))

;Relations 2
(declare-const relations2 (Set Relation))
(assert (= (card relations2) 0))

;Inheritances 2
(declare-const inheritances2 (Set Inheritance))
(assert (= (card inheritances2) 0))

;Class Diagram 2
(declare-const diagram2 ClassDiagram)
(assert (= diagram2
    (CD dictionary2 classes2 types2 attributes2 operations2 relations2 inheritances2)))


;---------------------------------
; Checks
;---------------------------------
(declare-const equiv_dicts Bool)
(declare-const equiv_classes Bool)
(declare-const equiv_types Bool)
(declare-const equiv_attributes Bool)
(declare-const equiv_operations Bool)
(declare-const equiv_relations Bool)
(declare-const equiv_inheritances Bool)
(assert (= equiv_classes (check_classes classes1 classes2)))
(assert (= equiv_attributes (check_attributes attributes1 attributes2)))
(assert (= equiv_operations (check_operations operations1 operations2)))
(assert (= equiv_inheritances (check_inheritances inheritances1 inheritances2)))

(define-fun equiv_diagrams () Bool
    (and
        equiv_classes
        equiv_attributes
        equiv_operations
        equiv_inheritances
    )
)

(declare-const are_equivalent Bool)
(assert (= equiv_diagrams are_equivalent))

(check-sat)
(get-value (equiv_classes))
(get-value (equiv_attributes))
(get-value (equiv_operations))
(get-value (equiv_inheritances))
(get-value (are_equivalent))

(exit)
```

C.3.0.4 Equivalent class models

SMTLib C.8: Full SMT-LIB model to check equivalent models

```
(set-option :produce-models true)
(set-logic ALL_SUPPORTED)

;---------------------------------
; Datatypes and structures
;---------------------------------


(declare-datatypes ( (Entry     0) ) ( ((mk-entry
                                        (synonyms (Set String))
                                    ))))


; UML datatypes
(declare-datatypes ( (Class     0) ) ( ((CLASS
    (cls_nme String) (cls_typ String) ))))

(declare-datatypes ( (Type      0) ) ( ((TYPE
    (typ_nme String) ))))

(declare-datatypes ( (Attribute 0) ) ( ((ATR
    (atr_cls String) (atr_nme String) (atr_typ String)
    (atr_vis String) (atr_sco String) ))))

(declare-datatypes ( (Param     0) ) ( ((PARAM
    (prm_nme String) (prm_typ String)) (nil) )))

(declare-datatypes ( (Operation 0) ) ( ((OPR
    (opr_cls String) (opr_nme String) (opr_typ String)
    (opr_vis String) (opr_sco String) (opr_prm Param)))))

(declare-datatypes ( (Relation 0) ) ( ((REL
    (rel_src String) (rel_des String) (rel_typ String)
    (rel_nme String) (rel_rol String) (rel_c_l String)
    (rel_c_u String) ))))

(declare-datatypes ( (Inheritance     0) ) ( ((INH
    (inh_sup String) (inh_sub String) ))))

(declare-datatypes ( (ClassDiagram     0) ) ( ((CD
    (dictionary (Set String)) (classes (Set Class))
    (types (Set Type)) (attributes (Set Attribute))
    (operations (Set Operation)) (relations (Set Relation))
    (inheritance (Set Inheritance)) ))))



;---------------------------------
; Synonyms
;---------------------------------
(declare-const syns_dict (Set Entry))

(assert (member (mk-entry (insert
    "Paper" "paper" "PAPER" "Papers" "papers"
    (singleton "Paper"))) syns_dict))

(assert (member (mk-entry (insert
```

```
    "title"   "TITLE"   "Title"   "titles"   "statute_title"   "STATUTE_TITLE"
    ↪ "StatuteTitle"   "statuteTitle"   "statute-title"   "statute_titles"     "Statute
    ↪ Title"   "rubric"   "RUBRIC"   "Rubric"   "rubrics"
    (singleton "title"))) syns_dict))

(assert (member (mk-entry (insert
    "word-count"   "WORD-COUNT"   "Word-count"   "word_count"   "word-counts"
    "word_counts"   "Word Count"
    (singleton "word-count"))) syns_dict))

(assert (member (mk-entry (insert
    "student-indicator"   "STUDENT-INDICATOR"   "Student-indicator"
    "student_indicator"   "student-indicators"   "student_indicators"
    "Student Indicator"
    (singleton "student-indicator"))) syns_dict))

(assert (member (mk-entry (insert
    "authors"   "AUTHORS"   "Authors"   "author"   "writer"   "WRITER"
    "Writer"   "writers"   "AUTHOR"   "Author"
    (singleton "authors"))) syns_dict))

(assert (member (mk-entry (insert
    "referees"   "REFEREES"   "Referees"   "referee"   "REFEREE"   "Referee"
    "ref"   "REF"   "Ref"   "refs"
    (singleton "referees"))) syns_dict))

(assert (member (mk-entry (insert
    "Researcher"   "researcher"   "RESEARCHER"   "Researchers"   "researchers"
    "research_worker"   "RESEARCH_WORKER"   "ResearchWorker"   "researchWorker"
    "research-worker"   "research_workers"   "Research Worker"   "investigator"
    "INVESTIGATOR"   "Investigator"   "investigators"
    (singleton "Researcher"))) syns_dict))

(assert (member (mk-entry (insert
    "name"   "NAME"   "Name"   "names"
    (singleton "name"))) syns_dict))


(assert (= syns_dict (as univset (Set Entry))))


;---------------------------------
; Constructors and helpers
;---------------------------------

(define-fun is_syn ((w1 String) (w2 String)) Bool
    (or
        (= w1 w2)
        (exists
            ((e Entry))
            (and
                (member e syns_dict)
                (and
                    (member w1 (synonyms e))
                    (member w2 (synonyms e)))))))

;---------------------------------
; Inference rules
;---------------------------------

(define-fun check_dictionary ((s1 (Set String)) (s2 (Set String))) Bool
    (forall
```

```
        ((x String))
        (=>
            (member x s1)
            (exists
                ((y String))
                (and
                    (member y s2)
                    (is_syn x y)
                )
            )
        )
    )
)


(define-fun check_classes ((s1 (Set Class)) (s2 (Set Class))) Bool
    (forall
        ((x Class))
        (=>
            (member x s1)
            (exists
                ((y Class))
                (and
                    (member y s2)
                    (is_syn (cls_nme x) (cls_nme y))
                )
            )
        )
    )
)

(define-fun check_types ((s1 (Set Type)) (s2 (Set Type))) Bool
    (forall
        ((x Type))
        (=>
            (member x s1)
            (exists
                ((y Type))
                (and
                    (member y s2)
                    (is_syn (typ_nme x) (typ_nme y))
                )
            )
        )
    )
)

(define-fun check_attributes ((s1 (Set Attribute)) (s2 (Set Attribute))) Bool
    (forall
        ((x Attribute))
        (=>
            (member x s1)
            (exists
                ((y Attribute))
                (and
                    (member y s2)
                    (is_syn (atr_cls x) (atr_cls y))
                    (is_syn (atr_nme x) (atr_nme y))
                )
            )
        )
    )
```

```
)

(define-fun check_operations ((s1 (Set Operation)) (s2 (Set Operation))) Bool
    (forall
        ((x Operation))
        (=>
            (member x s1)
            (exists
                ((y Operation))
                (and
                    (member y s2)
                    (is_syn (opr_nme x) (opr_nme y))
                )
            )
        )
    )
)

(define-fun check_relations ((s1 (Set Relation)) (s2 (Set Relation))) Bool
    (forall
        ((x Relation))
        (=>
            (member x s1)
            (exists
                ((y Relation))
                (and
                    (member y s2)
                    (is_syn (rel_src x) (rel_src y))
                    (is_syn (rel_des x) (rel_des y))
                )
            )
        )
    )
)


(define-fun check_inheritances ((s1 (Set Inheritance)) (s2 (Set Inheritance))) Bool
    (forall
        ((x Inheritance))
        (=>
            (member x s1)
            (exists
                ((y Inheritance))
                (and
                    (member y s2)
                    (is_syn (inh_sup x) (inh_sup y))
                    (is_syn (inh_sub x) (inh_sub y))
                )
            )
        )
    )
)


;---------------------------------
; Class Diagram 1
;---------------------------------
;Dictionary 1
(declare-const dictionary1 (Set String))
;£fill: dictionary 1

;Classes 1
```

```
(declare-const classes1 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes1))
(assert (member (CLASS "Researcher" "classifier") classes1))
(assert (= (card classes1) 2))

;Types 1
(declare-const types1 (Set Type))
;£fill: types 1

;Attributes 1
(declare-const attributes1 (Set Attribute))
(assert (member (ATR "Paper" "title" "" "" "") attributes1))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes1))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes1))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes1))(assert (member (ATR "Paper" "referees" "" "" "") attributes1))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes1))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes1))(assert (= (card attributes1)
↪  7))

;Operations 1
(declare-const operations1 (Set Operation))
(assert (= (card operations1) 0))

;Relations 1
(declare-const relations1 (Set Relation))
(assert (= (card relations1) 0))

;Inheritances 1
(declare-const inheritances1 (Set Inheritance))
(assert (= (card inheritances1) 0))

;Class Diagram 1
(declare-const diagram1 ClassDiagram)
(assert (= diagram1
    (CD dictionary1 classes1 types1 attributes1 operations1 relations1 inheritances1)))


;---------------------------------
; Class Diagram 2
;---------------------------------
;Dictionary 2
(declare-const dictionary2 (Set String))
;£fill: dictionary 2

;Classes 2
(declare-const classes2 (Set Class))
(assert (member (CLASS "Paper" "classifier") classes2))
(assert (member (CLASS "Researcher" "classifier") classes2))
(assert (= (card classes2) 2))

;Types 2
(declare-const types2 (Set Type))
;£fill: types 2

;Attributes 2
(declare-const attributes2 (Set Attribute))
```

```
(assert (member (ATR "Paper" "title" "" "" "") attributes2))(assert (member (ATR "Paper"
↪  "word-count" "" "" "") attributes2))(assert (member (ATR "Paper" "student-indicator"
↪  "" "" "") attributes2))(assert (member (ATR "Paper" "authors" "" "" "")
↪  attributes2))(assert (member (ATR "Paper" "referees" "" "" "") attributes2))(assert
↪  (member (ATR "Researcher" "name" "" "" "") attributes2))(assert (member (ATR
↪  "Researcher" "student-indicator" "" "" "") attributes2))(assert (= (card attributes2)
↪  7))

;Operations 2
(declare-const operations2 (Set Operation))
(assert (= (card operations2) 0))

;Relations 2
(declare-const relations2 (Set Relation))
(assert (= (card relations2) 0))

;Inheritances 2
(declare-const inheritances2 (Set Inheritance))
(assert (= (card inheritances2) 0))

;Class Diagram 2
(declare-const diagram2 ClassDiagram)
(assert (= diagram2
    (CD dictionary2 classes2 types2 attributes2 operations2 relations2 inheritances2)))


;---------------------------------
; Checks
;---------------------------------
(declare-const equiv_dicts Bool)
(declare-const equiv_classes Bool)
(declare-const equiv_types Bool)
(declare-const equiv_attributes Bool)
(declare-const equiv_operations Bool)
(declare-const equiv_relations Bool)
(declare-const equiv_inheritances Bool)
(assert (= equiv_classes (check_classes classes1 classes2)))
(assert (= equiv_attributes (check_attributes attributes1 attributes2)))
(assert (= equiv_operations (check_operations operations1 operations2)))
(assert (= equiv_inheritances (check_inheritances inheritances1 inheritances2)))

(define-fun equiv_diagrams () Bool
    (and
        equiv_classes
        equiv_attributes
        equiv_operations
        equiv_inheritances
    )
)

(declare-const are_equivalent Bool)
(assert (= equiv_diagrams are_equivalent))

(check-sat)
(get-value (equiv_classes))
(get-value (equiv_attributes))
(get-value (equiv_operations))
(get-value (equiv_inheritances))
(get-value (are_equivalent))

(exit)
```

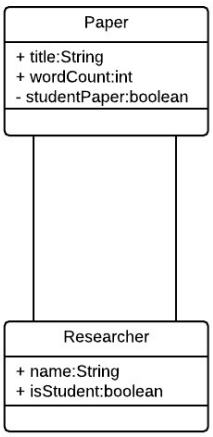


Figure C.1: Class diagram containing only attributes generated by us

C.4 Class Diagrams for Model Extraction

C.4.1 Diagram generated by us containing only attributes

JSON Model C.4: Extracted JSON class model from a class diagram generated by us containing only attributes


```
{
    "classes": {
        "class1556741427092573": {
            "name": "Class1556741427092573",
            "class_type": "class",
            "attributes": {
                "wordcount": {
                    "name": "wordCount"
                },
                "studentpaper": {
                    "name": "studentPaper"
                }
            },
            "operations": {}
        },
        "class15567414275395752": {
            "name": "Class15567414275395752",
            "class_type": "class",
            "attributes": {
                "isstudent": {
                    "name": "isStudent"
                }
            },
            "operations": {}
        }
    },
    "associations": {}
}
```

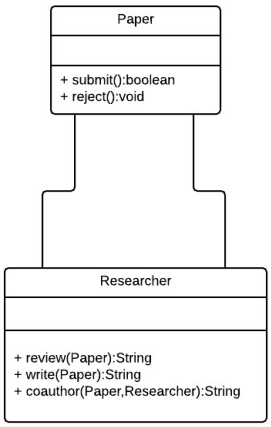


Figure C.2: Class diagram containing only operations generated by us

C.4.2 Diagram generated by us containing only operations

JSON Model C.5: Extracted JSON class model from a class diagram generated by us containing only operations


```
{
    "classes": {
        "class1556741534929416": {
            "name": "Class1556741534929416",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "reject-epsilon": {
                    "name": "reject",
                    "parameters": {}
                }
            }
        },
        "class15567415355371758": {
            "name": "Class15567415355371758",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "writepaper-epsilon": {
                    "name": "writePaper",
                    "parameters": {}
                }
            }
        }
    },
    "associations": {}
}
```

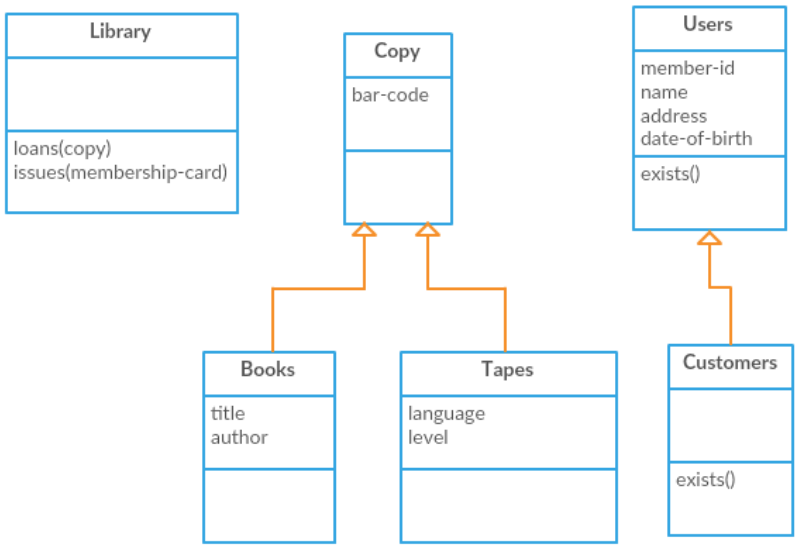


Figure C.3: Coloured class diagram generated by us

C.4.3 Coloured Diagram generated by us

JSON Model C.6: Extracted JSON class model from a coloured class diagram generated by us

```
{
    "classes": {
        "users": {
            "name": "Users",
            "class_type": "class",
            "attributes": {
                "name": {
                    "name": "name"
                },
                "address": {
                    "name": "address"
                }
            },
            "operations": {
                "exists-epsilon": {
                    "name": "exists",
                    "parameters": {}
                }
            }
        },
        "library": {
            "name": "Library",
            "class_type": "class",
            "attributes": {},
            "operations": {
                "loanscopy-epsilon": {
                    "name": "loanscopy",
```

```
                    "parameters": {}
                }
            }
        },
        "class1556741604235644": {
            "name": "Class1556741604235644",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "books": {
            "name": "Books",
            "class_type": "class",
            "attributes": {
                "title": {
                    "name": "title"
                },
                "author": {
                    "name": "author"
                }
            },
            "operations": {}
        },
        "class15567416047291602": {
            "name": "Class15567416047291602",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "tapes": {
            "name": "Tapes",
            "class_type": "class",
            "attributes": {
                "language": {
                    "name": "language"
                },
                "level": {
                    "name": "level"
                }
            },
            "operations": {}
        }
    },
    "associations": {}
}
```


C.4.4 Existing diagram containing only attributes

JSON Model C.7: Extracted JSON class model from an existing class diagram containing only attributes


```
{
    "classes": {
        "class1556714237249346": {
            "name": "Class1556714237249346",
            "class_type": "class",
            "attributes": {
                "maximum": {
```

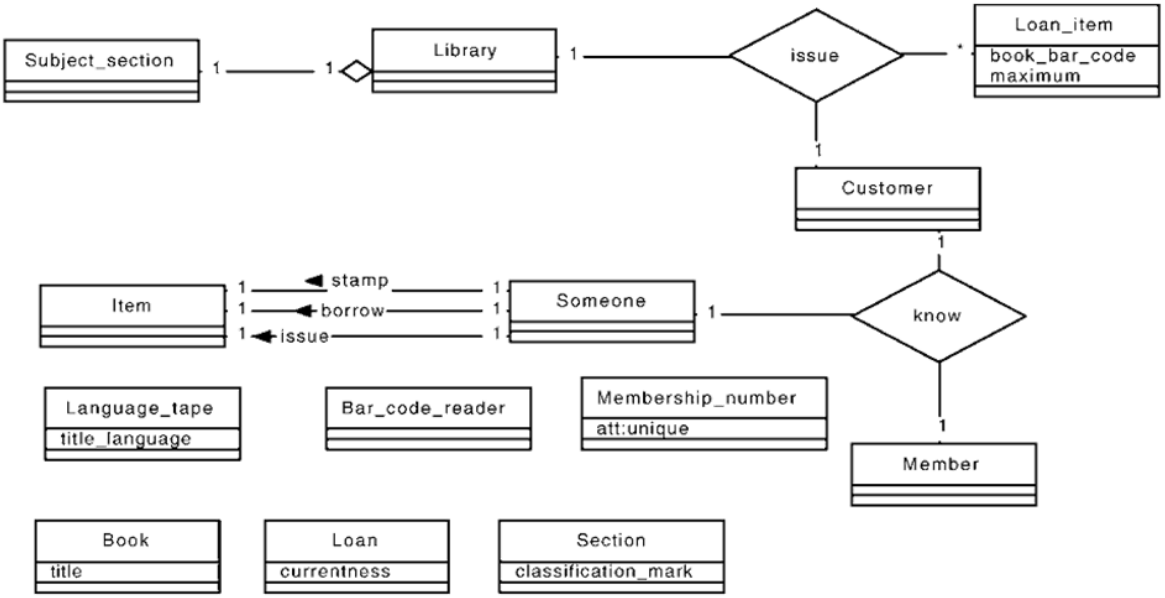


Figure C.4: Existing class diagram containing only attributes


```
                "name": "maximum"
            }
        },
        "operations": {}
    },
    "library": {
        "name": "Library",
        "class_type": "class",
        "attributes": {},
        "operations": {}
    },
    "class1556714237883433": {
        "name": "Class1556714237883433",
        "class_type": "class",
        "attributes": {},
        "operations": {}
    },
    "class15567142381023322": {
        "name": "Class15567142381023322",
        "class_type": "class",
        "attributes": {},
        "operations": {}
    },
    "customer": {
        "name": "Customer",
        "class_type": "class",
        "attributes": {
            "7": {
                "name": "7"
            }
        },
        "operations": {}
    },
```

```
        "someone": {
            "name": "Someone",
            "class_type": "class",
            "attributes": {
                "membersh": {
                    "name": "Membersh"
                },
                "attiunique": {
                    "name": "attiunique"
                }
            },
            "operations": {}
        },
        "class1556714239051326": {
            "name": "Class1556714239051326",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "class1556714239662323": {
            "name": "Class1556714239662323",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "class1556714240075325": {
            "name": "Class1556714240075325",
            "class_type": "class",
            "attributes": {
                "nique": {
                    "name": "nique"
                }
            },
            "operations": {}
        },
        "class1556714240442323": {
            "name": "Class1556714240442323",
            "class_type": "class",
            "attributes": {
                "titlelanguage": {
                    "name": "Titlelanguage"
                }
            },
            "operations": {}
        },
        "class1556714240746324": {
            "name": "Class1556714240746324",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "class1556714241006322": {
            "name": "Class1556714241006322",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "book": {
            "name": "Book",
            "attributes": {
                "title": {
                    "name": "Title"
```

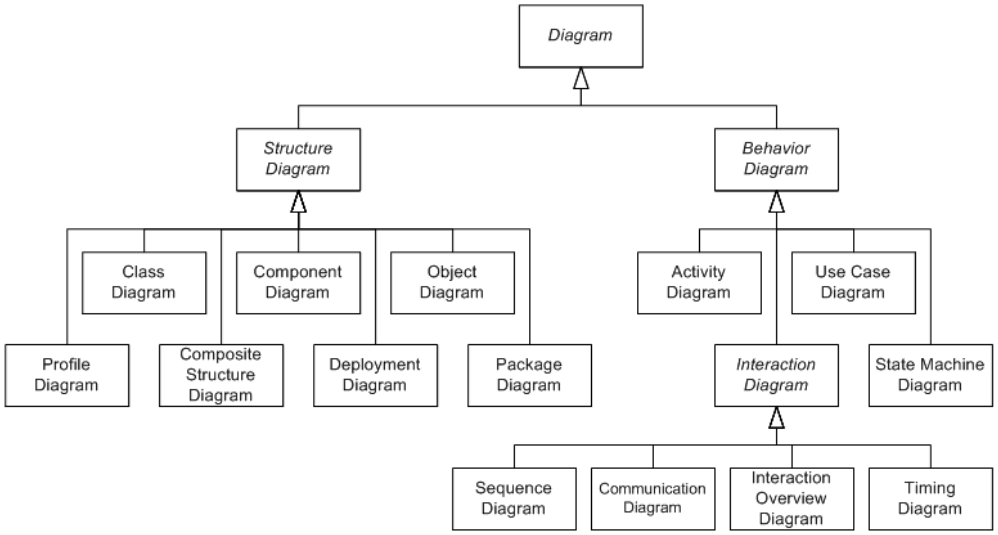


Figure C.5: Existing class diagram containing only classes

```
                }
            },
            "operations": {}
        },
        "loan": {
            "name": "Loan",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "section": {
            "name": "Section",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        }
    },
    "associations": {}
}
```

C.4.5 Existing diagram containing only classes

JSON Model C.8: Extracted JSON class model from an existing class diagram containing only attributes

```
{
    "classes": {},
    "associations": {}
}
```

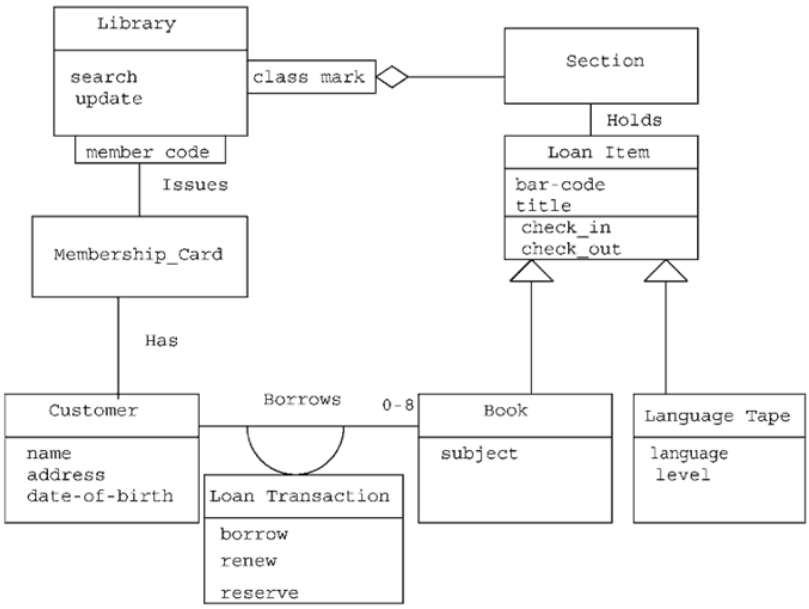


Figure C.6: Existing complete diagram with attributes and operations

C.4.6 Existing complete diagram containing attributes, and operations

JSON Model C.9: Extracted JSON class model from an existing complete class diagram with attributes and operations


```
{
    "classes": {
        "class1556712268236669": {
            "name": "Class1556712268236669",
            "class_type": "class",
            "attributes": {
                "update": {
                    "name": "update"
                }
            },
            "operations": {}
        },
        "section": {
            "name": "Section",
            "class_type": "class",
            "attributes": {
                "holds": {
                    "name": "Holds"
                },
                "loanitem": {
                    "name": "LoanItem"
                },
                "title": {
                    "name": "title"
                }
```

```
            },
            "operations": {}
        },
        "issues": {
            "name": "Issues",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "languagetape": {
            "name": "LanguageTape",
            "class_type": "class",
            "attributes": {
                "language": {
                    "name": "language"
                },
                "level": {
                    "name": "level"
                }
            },
            "operations": {}
        },
        "class1556712270651666": {
            "name": "Class1556712270651666",
            "class_type": "class",
            "attributes": {
                "subject": {
                    "name": "subject"
                },
                "tion": {
                    "name": "tion"
                }
            },
            "operations": {}
        },
        "class1556712271316668": {
            "name": "Class1556712271316668",
            "class_type": "class",
            "attributes": {
                "customer": {
                    "name": "customer"
                },
                "name": {
                    "name": "name"
                },
                "address": {
                    "name": "address"
                }
            },
            "operations": {}
        },
        "bostows": {
            "name": "BOSTOWS",
            "class_type": "class",
            "attributes": {
                "borrow": {
                    "name": "borrow"
                },
                "renew": {
                    "name": "renew"
                },
                "reserve": {
```

```
                    "name": "reserve"
                }
            },
            "operations": {}
        }
    },
    "associations": {}
}
```

C.4.7 Existing complex diagram

JSON Model C.10: Extracted JSON class model from an existing class diagram with a complex set of connections

```
{
    "classes": {
        "class1556716751375274": {
            "name": "Class1556716751375274",
            "class_type": "class",
            "attributes": {
                "ee": {
                    "name": "ee"
                }
            },
            "operations": {}
        },
        "class15567167518818901": {
            "name": "Class15567167518818901",
            "class_type": "class",
            "attributes": {
                "what": {
                    "name": "what"
                },
                "life": {
                    "name": "life"
                }
            },
            "operations": {}
        },
        "class1556716752278167": {
            "name": "Class1556716752278167",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        },
        "each": {
            "name": "Each",
            "class_type": "class",
            "attributes": {
                "custor": {
                    "name": "custor"
                }
            },
            "operations": {}
        },
        "class15567167534231628": {
            "name": "Class15567167534231628",
            "class_type": "class",
```

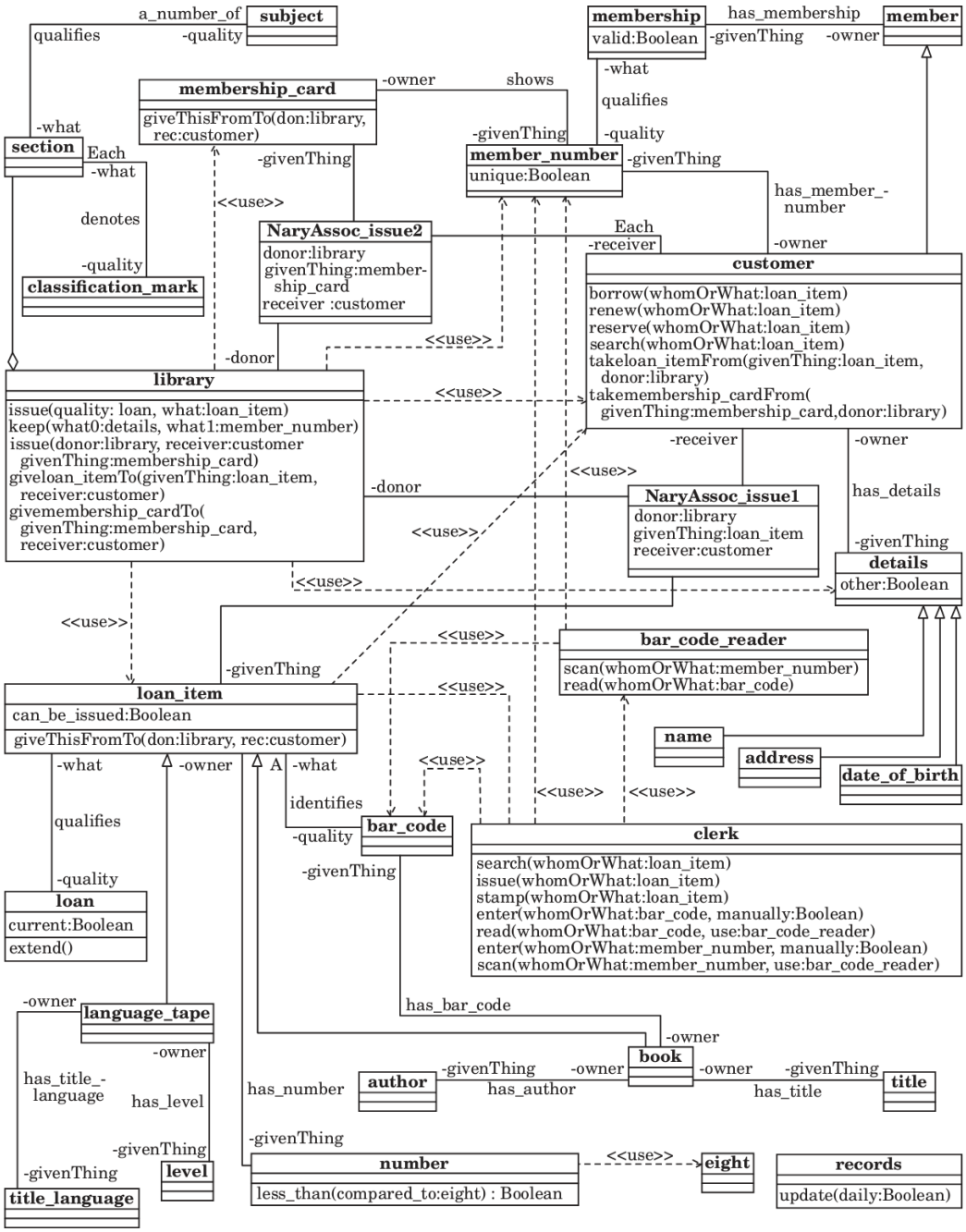


Figure C.7: Existing complex class diagram

```
            "attributes": {
                "eee": {
                    "name": "eee"
                },
                "1iototfac": {
                    "name": "1iototFac"
                }
            },
            "operations": {}
        },
        "class1556716755416943": {
            "name": "Class1556716755416943",
            "class_type": "class",
            "attributes": {
                "number": {
                    "name": "number"
                },
                "wher": {
                    "name": "wher"
                },
                "ner": {
                    "name": "ner"
                }
            },
            "operations": {}
        },
        "class1556716756123987": {
            "name": "Class1556716756123987",
            "class_type": "class",
            "attributes": {
                "donor": {
                    "name": "donor"
                },
                "receiver": {
                    "name": "receiver"
                }
            },
            "operations": {}
        },
        "class1556716757365777": {
            "name": "Class1556716757365777",
            "class_type": "class",
            "attributes": {
                "a": {
                    "name": "a"
                },
                "librar": {
                    "name": "librar"
                },
                "teayalqualtyv": {
                    "name": "Teayalqualtyv"
                }
            },
            "operations": {}
        },
        "class1556716758488799": {
            "name": "Class1556716758488799",
            "class_type": "class",
            "attributes": {
                "rary": {
                    "name": "rary"
                },
```

```
            "eceiver": {
                "name": "eceiver"
            },
            "nthine": {
                "name": "nThine"
            }
        },
        "operations": {}
    },
    "at": {
        "name": "At",
        "class_type": "class",
        "attributes": {
            "roydonor": {
                "name": "roydonor"
            },
            "oonvsiventhins": {
                "name": "oonvsivenThins"
            }
        },
        "operations": {}
    },
    "class1556716761924104": {
        "name": "Class1556716761924104",
        "class_type": "class",
        "attributes": {
            "y": {
                "name": "y"
            },
            "loanitemeo": {
                "name": "loanitemeo"
            }
        },
        "operations": {}
    },
    "class1556716763607081": {
        "name": "Class1556716763607081",
        "class_type": "class",
        "attributes": {},
        "operations": {}
    },
    "class15567167653221372": {
        "name": "Class15567167653221372",
        "class_type": "class",
        "attributes": {
            "vy1": {
                "name": "vy1"
            },
            "et": {
                "name": "et"
            },
            "a": {
                "name": "a"
            }
        },
        "operations": {}
    },
    "class1556716765926251": {
        "name": "Class1556716765926251",
        "class_type": "class",
        "attributes": {
            "details": {
```

```
                "name": "details"
            },
            "ler": {
                "name": "ler"
            }
        },
        "operations": {}
    },
    "class1556716766995095": {
        "name": "Class1556716766995095",
        "class_type": "class",
        "attributes": {
            "|receiver": {
                "name": "|receiver"
            },
            "1": {
                "name": "1"
            }
        },
        "operations": {}
    },
    "class15567167681133708": {
        "name": "Class15567167681133708",
        "class_type": "class",
        "attributes": {
            ",": {
                "name": ","
            },
            "ini": {
                "name": "inI"
            }
        },
        "operations": {}
    },
    "class15567167687911091": {
        "name": "Class15567167687911091",
        "class_type": "class",
        "attributes": {
            "qualifies": {
                "name": "qualifies"
            }
        },
        "operations": {}
    },
    "i": {
        "name": "I",
        "class_type": "class",
        "attributes": {
            "clerk": {
                "name": "clerk"
            }
        },
        "operations": {}
    },
    "class1556716771411355": {
        "name": "Class1556716771411355",
        "class_type": "class",
        "attributes": {},
        "operations": {}
    },
    "class1556716772780141": {
        "name": "Class1556716772780141",
```

```
            "class_type": "class",
            "attributes": {
                "yshott": {
                    "name": "yshott"
                }
            },
            "operations": {}
        },
        "class1556716773079196": {
            "name": "Class1556716773079196",
            "class_type": "class",
            "attributes": {
                "quality": {
                    "name": "quality"
                },
                "giventh": {
                    "name": "givenTh"
                }
            },
            "operations": {}
        },
        "class1556716773577837": {
            "name": "Class1556716773577837",
            "class_type": "class",
            "attributes": {
                "owner": {
                    "name": "owner"
                }
            },
            "operations": {
                "extend-epsilon": {
                    "name": "extend",
                    "parameters": {}
                }
            }
        },
        "class1556716774501925": {
            "name": "Class1556716774501925",
            "class_type": "class",
            "attributes": {
                "giventhing": {
                    "name": "givenThing"
                }
            },
            "operations": {}
        },
        "class1556716775492167": {
            "name": "Class1556716775492167",
            "class_type": "class",
            "attributes": {
                "owner": {
                    "name": "owner"
                },
                ".": {
                    "name": "."
                }
            },
            "operations": {}
        },
        "eoee": {
            "name": "Eoee",
            "class_type": "class",
```

```
            "attributes": {
                "|": {
                    "name": "|"
                },
                "number": {
                    "name": "number"
                }
            },
            "operations": {}
        },
        "class1556716777753823": {
            "name": "Class1556716777753823",
            "class_type": "class",
            "attributes": {
                "ywner": {
                    "name": "ywner"
                },
                "level": {
                    "name": "level"
                }
            },
            "operations": {}
        },
        "class1556716778787598": {
            "name": "Class1556716778787598",
            "class_type": "class",
            "attributes": {
                "giventh": {
                    "name": "givenTh"
                },
                "giventhingps": {
                    "name": "givenThingps"
                }
            },
            "operations": {}
        },
        "class1556716779265207": {
            "name": "Class1556716779265207",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        }
    },
    "associations": {}
}
```


C.4.8 Existing diagram drawn by hand

JSON Model C.11: Extracted JSON class model from an existing class diagram drawn by hand


```
{
    "classes": {
        "xn": {
            "name": "XN",
            "class_type": "class",
            "attributes": {},
            "operations": {}
```

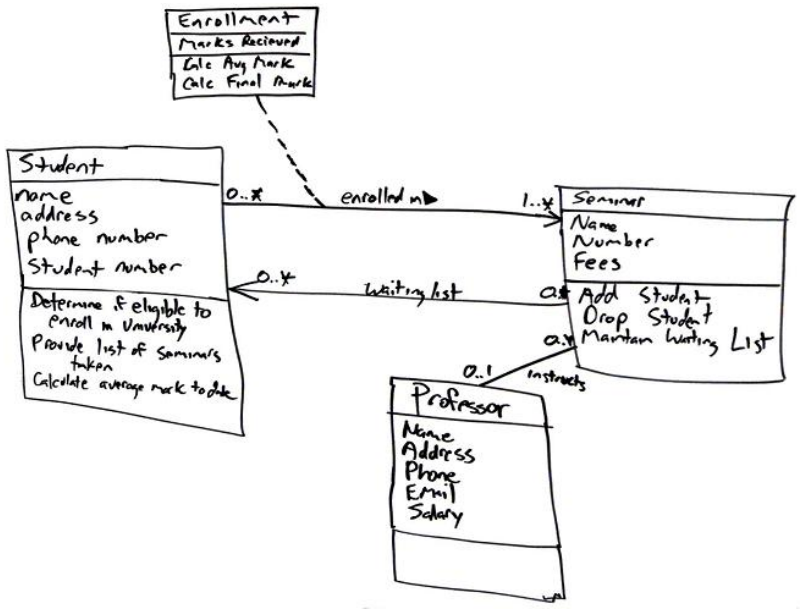


Figure C.8: Existing class diagram drawn by hand


```
        },
        "name": {
            "name": "Name",
            "class_type": "class",
            "attributes": {
                "mvebec": {
                    "name": "Mvebec"
                },
                "fees": {
                    "name": "Fees"
                },
                "asted": {
                    "name": "aSted"
                },
                "beepset": {
                    "name": "BeepSet"
                },
                "aymartenheslis": {
                    "name": "ayMartenhesLis"
                }
            },
            "operations": {}
        },
        "class1556741952920076": {
            "name": "Class1556741952920076",
            "class_type": "class",
            "attributes": {
                "fdisk": {
                    "name": "fdisk"
                },
                "<": {
                    "name": "<"
                }
```

```
            },
            "operations": {}
        },
        "class1556741953701626": {
            "name": "Class1556741953701626",
            "class_type": "class",
            "attributes": {},
            "operations": {}
        }
    },
    "associations": {}
}
```

List of Figures

List of Tables

List of Grammars

List of Texts

List of ConSpec Specifications

List of JSON Class Models

List of Predicates

List of SMTLib models

List of Equations

List of Acronyms

BNF Backus–Naur form. 1, 54, 58

CNL Controlled Natural Language. 1, 45, 54–57, 61, 64, 65

DbC Design By Contract. 1

FOL First Order Logic. 1, 37, 38, 46

NLP Natural Language Processing. 1, 55

POS Parts of Speech. 1, 55, 56

RE Requirements Engineering. 1, 24, 26, 44, 51

REP Requirements Engineering Process. 1, 51, 60

SE Software Engineering. 1, 22, 51

UML Unified Model Language. 1

Bibliography

[1] Object Management Group (OMG). *Requirements Interchange Format (ReqIF) Specification, Version 1.2.* `http://www.omg.org/spec/ReqIF/1.2`. 2016.

[2] Mariem Abdouli, Wahiba Ben Abdessalem Karaa, and Henda Ben Ghezala. "Survey of works that transform requirements into UML diagrams". In: *2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA)*. IEEE. 2016, pp. 117–123.

[3] Parosh Aziz Abdulla et al. "A survey of regular model checking". In: *International Conference on Concurrency Theory*. Springer. 2004, pp. 35–48.

[4] Habtamu Abie, Reijo M Savola, and Ilesh Dattani. "Robust, secure, self-adaptive and resilient messaging middleware for business critical systems". In: *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. IEEE. 2009, pp. 153–160.

[5] Pekka Abrahamsson et al. "Agile software development methods: Review and analysis". In: *arXiv preprint arXiv:1709.08439* (2017).

[6] Pekka Abrahamsson et al. "New directions on agile methods: a comparative analysis". In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE. 2003, pp. 244–254.

[7] Jean-Raymond Abrial. "Steam-boiler control specification problem". In: *Formal Methods for Industrial Applications*. Springer, 1996, pp. 500–509.

[8] Jean-Raymond Abrial, Egon Börger, Hans Langmaack, et al. *Formal methods for industrial applications: Specifying and programming the steam boiler control*. Vol. 9. Springer Science & Business Media, 1996.

[9] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. "Software inspections: an effective verification process". In: *IEEE software* 6.3 (1989), pp. 31–36.

[10] W Richards Adrion, Martha A Branstad, and John C Cherniavsky. "Validation, verification, and testing of computer software". In: *ACM Computing Surveys (CSUR)* 14.2 (1982), pp. 159–192.

[11] Marcus Alanen and Ivan Porres. *Model interchange using OMG standards*. IEEE, 2005.

[12] Emin Aleskerov, Bernd Freisleben, and Bharat Rao. "Cardwatch: A neural network based database mining system for credit card fraud detection". In: *Proceedings of the IEEE/IAFE 1997 computational intelligence for financial engineering (CIFEr)*. IEEE. 1997, pp. 220–226.

[13] Mack Alford. *Software Requirements Engineering Methodology*. Wiley Online Library, 1979.

[14] Agile Alliance. *Manifesto for Agile Software Development*. 2001. URL: `https://agilemanifesto.org/`.

[15] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[16] Michaël Armand et al. "A modular integration of SAT/SMT solvers to Coq through proof witnesses". In: *International Conference on Certified Programs and Proofs*. Springer. 2011, pp. 135–150.

[17] Deborah J Armstrong. "The quarks of object-oriented development". In: *Communications of the ACM* 49.2 (2006), pp. 123–128.

[18] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[19] Colin Atkinson and Thomas Kuhne. "Model-driven development: a metamodeling foundation". In: *IEEE software* 20.5 (2003), pp. 36–41.

[20] Eric Atwell. "The University of Pennsylvania (Penn) Treebank Tagset". In: *University of Pennsylvania* (1990), p. 48.

[21] Franz Baader and Ulrike Sattler. "An overview of tableau algorithms for description logics". In: *Studia Logica* 69.1 (2001), pp. 5–40.

[22] Franz Baader et al. *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[23] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[24] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[25] Stacey Bailey and Detmar Meurers. "Diagnosing meaning errors in short answers to reading comprehension questions". In: *Proceedings of the Third Workshop on Innovative Use of NLP for Building Educational Applications*. Association for Computational Linguistics. 2008, pp. 107–115.

[26] Imran Sarwar Bajwa and M Abbas Choudhary. "From natural language software specifications to UML class models". In: *International Conference on Enterprise Information Systems*. Springer. 2011, pp. 224–237.

[27] Imran Sarwar Bajwa, Mark G Lee, and Behzad Bordbar. "SBVR Business Rules Generation from Natural Language Specification." In: *AAAI spring symposium: AI for business agility*. 2011, pp. 2–8.

[28] Imran Sarwar Bajwa and Muhammad Anwar Shahzada. "Automated Generation of OCL Constraints: NL based Approach vs Pattern Based Approach". In: *Mehran University Research Journal of Engineering and Technology* 36.2 (2017), pp. 243–254.

[29] Mira Balaban, Azzam Maraee, and Arnon Sturm. "Management of correctness problems in UML class diagrams towards a pattern-based approach". In: *International Journal of Information System Modeling and Design (IJISMD)* 1.4 (2010), pp. 24–47.

[30] Rajiv D Banker et al. "Software complexity and maintenance costs". In: *Communications of the ACM* 36.11 (1993), pp. 81–95.

[31] Horace B Barlow. "Unsupervised learning". In: *Neural computation* 1.3 (1989), pp. 295–311.

[32] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. "The Spec# programming system: An overview". In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer. 2004, pp. 49–69.

[33] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The satisfiability modulo theories library (SMT-LIB)(2010)". In: *SMT-LIB. org* 156 (2016).

[34] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The smt-lib standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*. Vol. 13. 2010, p. 14.

[35] Clark Barrett et al. "Cvc4". In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 171–177.

[36] Victor R Basili and David M Weiss. "Evaluation of a software requirements document by analysis of change data". In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pp. 314–323.

[37] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, 2007.

[38] Alex E Bell. "Death by UML fever". In: *Queue* 2.1 (2004), p. 72.

[39] Thomas E Bell and Thomas A Thayer. "Software requirements: Are they really a problem?" In: *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press. 1976, pp. 61–68.

[40] Herbert D Benington. "Production of large computer programs". In: *Annals of the History of Computing* 5.4 (1983), pp. 350–361.

[41] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. "Reasoning on UML class diagrams". In: *Artificial intelligence* 168.1-2 (2005), pp. 70–118.

[42] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. "Reasoning on UML class diagrams using description logic based systems". In: *Proc. of the KI'2001 Workshop on Applications of Description Logics*. Vol. 44. 2001.

[43] Jan A. Bergstra and JV Tucker. "Some natural structures which fail to possess a sound and decidable Hoare-like logic for their while-programs". In: *Theoretical Computer Science: the journal of the EATCS* 17.3 (1982), pp. 303–315.

[44] Steven Bird and Edward Loper. "NLTK: the natural language toolkit". In: *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics. 2004, p. 31.

[45] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[46] Christopher M Bishop et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[47] Barry W. Boehm. "Verifying and validating software requirements and design specifications". In: *IEEE software* 1.1 (1984), p. 75.

[48] Wendy Boggs and Michael Boggs. *Mastering UML with Rational Rose*. Sybex San Francisco, CA, 1999.

[49] Grady Booch. "UML in action". In: *Communications of the ACM* 42.10 (1999), pp. 26–28.

[50] Alex Borgida. "On the relative expressiveness of description logics and predicate logics". In: *Artificial intelligence* 82.1-2 (1996), pp. 353–367.

[51] Jürgen Börstler. "User-centered requirements engineering in record-an overview". In: *the Nordic Workshop on Programming Environment Research, Proceedings NWPER*. Vol. 96. 1996, pp. 149–156.

[52] Jonathan P Bowen and Michael G Hinchey. "Seven more myths of formal methods". In: *IEEE software* 12.4 (1995), pp. 34–41.

[53] Jonathan Bowen and Victoria Stavridou. "Safety-critical systems, formal methods and standards". In: *Software Engineering Journal* 8.4 (1993), pp. 189–209.

[54] Karin K Breitman and Julio Cesar Sampaio do Prado Leite. "Ontology as a requirements engineering product". In: *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*. IEEE. 2003, pp. 309–319.

[55] Davide Bresolin et al. "Decidable and undecidable fragments of Halpern and Shoham's interval temporal logic: towards a complete classification". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2008, pp. 590–604.

[56] Stephen Brown, Tom Lysaght, and Deshi Ye. *Software Testing: Principles and Practice*. China Machine Press, 2012.

[57] Gregory D. Buzzard and Trevor N. Mudge. "Object-based computing and the ADA programming language". In: *Computer* 3 (1985), pp. 11–19.

[58] TIOBE software BV. *TIOBE Index*. 2018. URL: `https://www.tiobe.com/tiobe-index/` (visited on 03/17/2018).

[59] Jordi Cabot, Robert Claris, Daniel Riera, et al. "Verification of UML/OCL class diagrams using constraint programming". In: *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*. IEEE. 2008, pp. 73–80.

[60] Jordi Cabot, Robert Clarisó, and Daniel Riera. "UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming". In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM. 2007, pp. 547–548.

[61] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[62] Robert E Callan. *Building Object-Oriented Systems: An introduction from concepts to implementation in C++*. Computational Mechanics, 1994.

[63] Diego Calvanese et al. "Reasoning in Expressive Description Logics." In: *Handbook of automated reasoning* 2 (2001), pp. 1581–1634.

[64] H Frank Cervone. "Understanding agile project management methods using Scrum". In: *OCLC Systems & Services: International digital library perspectives* 27.1 (2011), pp. 18–22.

[65] Patrice Chalin et al. "Beyond assertions: Advanced specification and verification with JML and ESC/Java2". In: *International Symposium on Formal Methods for Components and Objects*. Springer. 2005, pp. 342–363.

[66] Jayeeta Chanda et al. "Traceability of requirements and consistency verification of UML use case, activity and Class diagram: A Formal approach". In: *Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on*. IEEE. 2009, pp. 1–4.

[67] Peter Pin-Shan Chen. "The entity-relationship model—toward a unified view of data". In: *ACM Transactions on Database Systems (TODS)* 1.1 (1976), pp. 9–36.

[68] Weiwei Cheng and Eyke Hüllermeier. "Combining instance-based learning and logistic regression for multilabel classification". In: *Machine Learning* 76.2-3 (2009), pp. 211–225.

[69] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[70] Sam Chung and Yun-Sik Lee. "Reverse software engineering with uml for web site maintenance". In: *Proceedings of the First International Conference on Web Information Systems Engineering*. Vol. 2. IEEE. 2000, pp. 157–161.

[71] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[72] Edmund M Clarke, Orna Grumberg, and David E Long. "Model checking and abstraction". In: *ACM transactions on Programming Languages and Systems (TOPLAS)* 16.5 (1994), pp. 1512–1542.

[73] Edmund M Clarke and Jeannette M Wing. "Formal methods: State of the art and future directions". In: *ACM Computing Surveys (CSUR)* 28.4 (1996), pp. 626–643.

[74] Edmund M Clarke et al. *Handbook of model checking*. Springer, 2018.

[75] Manuel Clavel and Marina Egea. "ITP/OCL: A rewriting-based validation tool for UML+ OCL static class diagrams". In: *International Conference on Algebraic Methodology and Software Technology*. Springer. 2006, pp. 368–373.

[76] Adam Coates, Andrew Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 215–223.

[77] Alistair Cockburn. "Structuring use cases with goals". In: *Journal of object-oriented programming* 10.5 (1997), pp. 56–62.

[78] Morris R Cohen. "The basis of contract". In: *Harvard Law Review* 46.4 (1933), pp. 553–592.

[79] David R Cok. "OpenJML: JML for Java 7 by extending OpenJDK". In: *NASA Formal Methods Symposium*. Springer. 2011, pp. 472–479.

[80] David R Cok and Joseph R Kiniry. "Esc/java2: Uniting esc/java and jml". In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer. 2004, pp. 108–128.

[81] Peter Alan Coldicott et al. *Automation of software application engineering using machine learning and reasoning*. US Patent 8,607,190. Dec. 2013.

[82] Ronan Collobert and Jason Weston. "A unified architecture for natural language processing: Deep neural networks with multitask learning". In: *Proceedings of the 25th international conference on Machine learning*. ACM. 2008, pp. 160–167.

[83] Stephen A Cook. "Soundness and completeness of an axiom system for program verification". In: *SIAM Journal on Computing* 7.1 (1978), pp. 70–90.

[84] Kendra Cooper and Mabo Ito. "1.6. 2 Formalizing a Structured Natural Language Requirements Specification Notation". In: *INCOSE International Symposium*. Vol. 12. 1. Wiley Online Library. 2002, pp. 1025–1032.

[85] Dan Craigen, Susan Gerhart, and Ted Ralston. "An international survey of industrial applications of formal methods". In: *Z User Workshop, London 1992*. Springer. 1993, pp. 1–5.

[86] John N Crossley. *What is mathematical logic?* Courier Corporation, 1990.

[87] Raúl Cruz-Barbosa and Alfredo Vellido. "Semi-supervised analysis of human brain tumours from partially labeled MRS information, using manifold learning models". In: *International journal of neural systems* 21.01 (2011), pp. 17–29.

[88] Wendy L Currie and Philip Seltsikas. "Delivering business critical information systems though application service providers: the need for a market segmentation strategy". In: *International Journal of Innovation Management* 5.03 (2001), pp. 323–349.

[89] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. "A survey of automated techniques for formal software verification". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (2008), pp. 1165–1178.

[90] Aspassia Daskalopulu and Marek Sergot. "The representation of legal contracts". In: *AI & SOCIETY* 11.1-2 (1997), pp. 6–17.

[91] Thomas H Davenport et al. *Mission critical: Realizing the promise of enterprise systems*. Harvard Business Press, 2000.

[92] Charles Day. "Python power". In: *Computing in Science & Engineering* 16.1 (2014), p. 88.

[93] William HE Day and Herbert Edelsbrunner. "Efficient algorithms for agglomerative hierarchical clustering methods". In: *Journal of classification* 1.1 (1984), pp. 7–24.

[94] Andrea De Lucia et al. "An experimental comparison of ER and UML class diagrams for data modelling". In: *Empirical Software Engineering* 15.5 (2010), pp. 455–492.

[95] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.

[96] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[97] Stéphane Demri, Deepak D'Souza, and Régis Gascon. "A decidable temporal logic of repeating values". In: *International Symposium on Logical Foundations of Computer Science*. Springer. 2007, pp. 180–194.

[98] Diego Dermeval et al. "A systematic review on the use of ontologies in requirements engineering". In: *Software Engineering (SBES), 2014 Brazilian Symposium on*. IEEE. 2014, pp. 1–10.

[99] Kurt W Derr. *Applying OMT: A Practical step-by-step guide to using the Object Modeling Technique*. Cambridge University Press, 1995.

[100] David Detlefs, Greg Nelson, and James B Saxe. "Simplify: a theorem prover for program checking". In: *Journal of the ACM (JACM)* 52.3 (2005), pp. 365–473.

[101] Edsger W Dijkstra. "Cooperating sequential processes". In: *The origin of concurrent programming*. Springer, 1968, pp. 65–138.

[102] Edsger Wybe Dijkstra. *Notes on structured programming*. 1970.

[103] Jun Ding et al. "Convolutional neural network with data augmentation for SAR target recognition". In: *IEEE Geoscience and remote sensing letters* 13.3 (2016), pp. 364–368.

[104] Natalia Dragan, Michael L Collard, and Jonathan I Maletic. "Automatic identification of class stereotypes". In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–10.

[105] R Geoff Dromey. "Cornering the chimera [software quality]". In: *IEEE Software* 13.1 (1996), pp. 33–43.

[106] Michael Dummett. "Wittgenstein's philosophy of mathematics". In: *The Philosophical Review* 68.3 (1959), pp. 324–348.

[107] Bruno Dutertre and Leonardo De Moura. "The yices smt solver". In: *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2.2 (2006), pp. 1–2.

[108] Matthew B Dwyer, George S Avrunin, and James C Corbett. "Property specification patterns for finite-state verification". In: *Proceedings of the second workshop on Formal methods in software practice*. ACM. 1998, pp. 7–15.

[109] Sean R Eddy. "Hidden markov models". In: *Current opinion in structural biology* 6.3 (1996), pp. 361–365.

[110] PAUL N Edwards. "From baggage to the PC, minus the hype". In: *IEEE Spectrum* 34.2 (1997), pp. 10–12.

[111] Golnaz Elahi, Eric Yu, and Nicola Zannone. "A modeling ontology for integrating vulnerabilities into security requirements conceptual foundations". In: *International Conference on Conceptual Modeling*. Springer. 2009, pp. 99–114.

[112] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Elsevier, 2001.

[113] Michael Fagan. "Design and code inspections to reduce errors in program development". In: *Software pioneers*. Springer, 2002, pp. 575–607.

[114] Michael E Fagan. "Advances in software inspections". In: *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 335–360.

[115] Kirill Fakhroutdinov. *UML Diagrams. Diagrams characteristics*. Accessed: 2018-12-29. 2012.

[116] Tom Fawcett and Foster Provost. "Adaptive fraud detection". In: *Data mining and knowledge discovery* 1.3 (1997), pp. 291–316.

[117] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[118] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[119] Robert L Flood and Michael C Jackson. *Critical systems thinking*. Springer, 1991.

[120] Kevin Forsberg and Harold Mooz. "The relationship of system engineering to the project cycle". In: *INCOSE International Symposium*. Vol. 1. 1. Wiley Online Library. 1991, pp. 57–65.

[121] Martin Fowler, Cris Kobryn, and Kendall Scott. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[122] Emilio Frazzoli, Munther A Dahleh, and Eric Feron. "Real-time motion planning for agile autonomous vehicles". In: *Journal of guidance, control, and dynamics* 25.1 (2002), pp. 116–129.

[123] Norbert E Fuchs, Uta Schwertel, and Rolf Schwitter. "Attempto Controlled English—not just another logic specification language". In: *International Workshop on Logic Programming Synthesis and Transformation*. Springer. 1998, pp. 1–20.

[124] Adam Funk et al. "Clone: Controlled language for ontology editing". In: *The Semantic Web*. Springer, 2007, pp. 142–155.

[125] Martin Furer, Oded Goldreich, and Yishay Mansour. "On completeness and soundness in interactive proof systems". In: (1989).

[126] Robert P Futrelle et al. "Extraction, layout analysis and classification of diagrams in PDF documents". In: *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.* IEEE. 2003, pp. 1007–1013.

[127] Elena García-Barriocanal, Miguel-Angel Sicilia, and Salvador Sánchez-Alonso. "Usability evaluation of ontology editors". In: *Knowledge Organization* 32.1 (2005), pp. 1–9.

[128] Tom Gelhausen. "Modellextraktion aus natürlichen Sprachen : eine Methode zur systematischen Erstellung von Domänenmodellen". German. PhD thesis. Karlsruhe Institute of Technology, 2010. 303 pp. ISBN: 978-3-86644-547-5. DOI: 10.5445/KSP/1000019366.

[129] Gerhard Gentzen. "Investigations into logical deduction". In: *American philosophical quarterly* 1.4 (1964), pp. 288–306.

[130] Susan Gerhart, Dan Craigen, and Ted Ralston. "Experience with formal methods in critical systems". In: *IEEE Software* 11.1 (1994), pp. 21–28.

[131] Robert L Glass. "The Standish report: does it really describe a software crisis?" In: *Communications of the ACM* 49.8 (2006), pp. 15–16.

[132] Kurt Gödel. *On formally undecidable propositions of Principia Mathematica and related systems*. Courier Corporation, 1992.

[133] Amrit L. Goel. "Software reliability models: Assumptions, limitations, and applicability". In: *IEEE Transactions on software engineering* 12 (1985), pp. 1411–1423.

[134] Martin Gogolla, Fabian Büttner, and Mark Richters. "USE: A UML-based specification environment for validating UML and OCL". In: *Science of Computer Programming* 69.1-3 (2007), pp. 27–34.

[135] Martin Gogolla and Mark Richters. "Equivalence rules for UML class diagrams". In: *The Unified Modeling Language, UML* 98 (1998), pp. 87–96.

[136] Martin Gogolla and Mark Richters. "Expressing UML class diagrams properties with OCL". In: *Object Modeling with the OCL*. Springer, 2002, pp. 85–114.

[137] Joseph A Goguen and Charlotte Linde. "Techniques for requirements elicitation". In: *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*. IEEE. 1993, pp. 152–164.

[138] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[139] Anthony Hall. "Seven myths of formal methods". In: *IEEE software* 7.5 (1990), pp. 11–19.

[140] Catalina Hallett, Donia Scott, and Richard Power. "Composing questions through conceptual authoring". In: *Computational linguistics* 33.1 (2007), pp. 105–133.

[141] Charles L Hamblin. "Translation to and from Polish Notation". In: *The Computer Journal* 5.3 (1962), pp. 210–213.

[142] Harmain M Harmain and R Gaizauskas. "CM-Builder: an automated NL-based CASE tool". In: *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*. IEEE. 2000, pp. 45–53.

[143] John A Hartigan and Manchek A Wong. "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108.

[144] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[145] Simon S Haykin et al. *Neural networks and learning machines*. Vol. 3. Pearson education Upper Saddle River, 2009.

[146] Kathryn L. Heninger. "Specifying software requirements for complex systems: New techniques and their application". In: *IEEE Transactions on Software Engineering* 1 (1980), pp. 2–13.

[147] Hatem Herchi and Wahiba Ben Abdessalem. "From user requirements to UML class diagram". In: *arXiv preprint arXiv:1211.0713* (2012).

[148] Ann M Hickey and Alan M Davis. "An ontological approach to requirements elicitation technique selection". In: *Ontologies*. Springer, 2007, pp. 403–431.

[149] Michael G Hinchey and Jonathan Peter Bowen. *Applications of formal methods*. Vol. 1. Prentice Hall New Jersey, 1995.

[150] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[151] C. A. R. Hoare. *Chapter II: Notes on data structuring*. Academic Press Ltd., 1972.

[152] Gerard J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295.

[153] Ian Horrocks and Ulrike Sattler. "Ontology reasoning in the SHOQ (D) description logic". In: *IJCAI*. Vol. 1. 3. 2001, pp. 199–204.

[154] SL Howells, RJ Maxwell, and JR Griffiths. "Classification of tumour 1H NMR spectra by pattern recognition". In: *NMR in Biomedicine* 5.2 (1992), pp. 59–64.

[155] Guang-Bin Huang, Qin-Yu Zhu, Chee-Kheong Siew, et al. "Extreme learning machine: a new learning scheme of feedforward neural networks". In: *Neural networks* 2 (2004), pp. 985–990.

[156] Ullrich Hustadt, Renate A Schmidt, and Lilia Georgieva. "A survey of decidable first-order fragments and description logics". In: *Journal of Relational Methods in Computer Science* 1.251-276 (2004), p. 3.

[157] The Standish Group International. *The CHAOS Report (1994)*. Tech. rep. The Standish Group, 1995.

[158] Nathan Isaacs. "The standardizing of contracts". In: *The Yale Law Journal* 27.1 (1917), pp. 34–48.

[159] IEC ISO. *IEEE. 29148: 2011-Systems and software engineering-Requirements engineering*. Tech. rep. IEEE, 2011.

[160] Tommi Jaakkola and David Haussler. "Exploiting generative models in discriminative classifiers". In: *Advances in neural information processing systems*. 1999, pp. 487–493.

[161] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[162] Ivar Jacobson, Grady Booch, and Jim Rumbaugh. "The Objectory Software Development Process". In: *ISBN: 0-201-57169-2, Addison Wesley* (1997).

[163] Anil K Jain. "Data clustering: 50 years beyond K-means". In: *Pattern recognition letters* 31.8 (2010), pp. 651–666.

[164] Aman Jatain and Deepti Gaur. "Reverse Engineering of Object Oriented System using Hierarchical Clustering". In: *INTERNATIONAL ARAB JOURNAL OF INFORMATION TECHNOLOGY* 15.5 (2018), pp. 857–865.

[165] Magne Jørgensen and Kjetil Moløkken-Østvold. "How large are software cost overruns? A review of the 1994 CHAOS report". In: *Information and Software Technology* 48.4 (2006), pp. 297–301.

[166] Dan Jurafsky and James H Martin. *Speech and language processing*. Vol. 3. Pearson London: 2014.

[167] Ivan Jureta, John Mylopoulos, and Stephane Faulkner. "Revisiting the core ontology and problem in requirements engineering". In: *International Requirements Engineering, 2008. RE'08. 16th IEEE*. IEEE. 2008, pp. 71–80.

[168] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[169] Vasanthi Kaliappan and Norhayati Mohd Ali. "Improving Consistency of UML Diagrams and Its Implementation Using Reverse Engineering Approach". In: *Bulletin of Electrical Engineering and Informatics* 7.4 (2018), pp. 665–672.

[170] Peter Kaye. *The new private international law of contract of the European Community: implementation of the EEC's Contractual Obligations Convention in England and Wales under the Contracts (Applicable Law) Act 1990*. Dartmouth, 1993.

[171] Soon-Kyeong Kim and Carrington David. "Formalizing the UML class diagram using Object-Z". In: *International Conference on the Unified Modeling Language*. Springer. 1999, pp. 83–98.

[172] Durk P Kingma et al. "Semi-supervised learning with deep generative models". In: *Advances in neural information processing systems*. 2014, pp. 3581–3589.

[173] Joseph R Kiniry and David R Cok. "{ESC/Java2}: Uniting {ESC/Java} and {JML}: Progress and issues in building and using {ESC/Java2} and a report on a case study involving the use of {ESC/Java2} to verify portions of an {Internet} voting tally system". In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Ed. by Gilles BartheLilian BurdyMarieke HuismanJean-Louis LanetTraian Muntean. 2005.

[174] Nahum Kiryati, Yuval Eldar, and Alfred M Bruckstein. "A probabilistic Hough transform". In: *Pattern recognition* 24.4 (1991), pp. 303–316.

[175] Barbara Kitchenham and Shari Lawrence Pfleeger. "Software quality: the elusive target [special issues section]". In: *IEEE software* 13.1 (1996), pp. 12–21.

[176] Thomas Kleymann. "Hoare logic and VDM: Machine-checked soundness and completeness proofs". In: (1998).

[177] John C Knight. "Safety critical systems: challenges and directions". In: *Proceedings of the 24th international conference on software engineering*. ACM. 2002, pp. 547–550.

[178] Donald E Knuth. "Backus normal form vs. backus naur form". In: *Communications of the ACM* 7.12 (1964), pp. 735–736.

[179] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. "Execution monitoring of security-critical programs in distributed systems: A specification-based approach". In: *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*. IEEE. 1997, pp. 175–187.

[180] Teuvo Kohonen. "The self-organizing map". In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480.

[181] Ralf Kollmann et al. "A study on the current state of the art in tool-supported UML-based static reverse engineering". In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 22–32.

[182] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[183] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[184] R Krzysztof and D Kozen. "Limits for automatic verification of finite-state concurrent systems". In: *Information Processing Letters* 22 (1986), pp. 307–309.

[185] Tobias Kuhn. "A survey and classification of controlled natural languages". In: *Computational Linguistics* 40.1 (2014), pp. 121–170.

[186] Deeptimahanti Deva Kumar and Ratna Sanyal. "Static UML model generator from analysis of requirements (SUGAR)". In: *2008 Advanced Software Engineering and Its Applications*. IEEE. 2008, pp. 77–84.

[187] Vipin Kumar. "Algorithms for constraint-satisfaction problems: A survey". In: *AI magazine* 13.1 (1992), p. 32.

[188] Zijad Kurtanović and Walid Maalej. "Automatically Classifying Functional and Non-functional Requirements Using Supervised Machine Learning". In: *Requirements Engineering Conference (RE), 2017 IEEE 25th International*. IEEE. 2017, pp. 490–495.

[189] Marta Kwiatkowska, Gethin Norman, and David Parker. "PRISM: Probabilistic symbolic model checker". In: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2002, pp. 200–204.

[190] Leslie Lamport. "Proving the correctness of multiprocess programs". In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[191] Phillip A Laplante and Colin J Neill. "The demise of the waterfall model is imminent". In: *Queue* 1.10 (2004), p. 10.

[192] James R Larus et al. "Righting software". In: *IEEE software* 21.3 (2004), pp. 92–100.

[193] Gary T Leavens, Albert L Baker, and Clyde Ruby. "JML: A notation for detailed design". In: *behavioral specifications of Businesses and Systems*. Springer, 1999, pp. 175–188.

[194] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), p. 436.

[195] Seok Won Lee and Robin A Gandhi. "Ontology-based active requirements engineering framework". In: *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*. IEEE. 2005, 8–pp.

[196] Andy Liaw, Matthew Wiener, et al. "Classification and regression by randomForest". In: *R news* 2.3 (2002), pp. 18–22.

[197] Richard J Lipton. "A necessary and sufficient condition for the existence of Hoare logics". In: *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE. 1977, pp. 1–6.

[198] Xavier Llora and Josep M Garrell. "Evolution of decision trees". In: *Forth Catalan Conference on Artificial Intelligence (CCIA'2001)*. 2001, pp. 115–122.

[199] Edward Loper and Steven Bird. "NLTK: the natural language toolkit". In: *arXiv preprint cs/0205028* (2002).

[200] Pericles Loucopoulos and Vassilios Karakostas. *System requirements engineering*. McGraw-Hill, Inc., 1995.

[201] Jan Luts et al. "A combined MRI and MRSI based multiclass system for brain tumour recognition using LS-SVMs with class probabilities and feature selection". In: *Artificial intelligence in medicine* 40.2 (2007), pp. 87–102.

[202] James MacQueen et al. "Some methods for classification and analysis of multivariate observations". In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[203] Sonal Mahajan and William GJ Halfond. "WebSee: A tool for debugging HTML presentation failures". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–8.

[204] Michael S Mahoney. "Finding a history for software engineering". In: *IEEE Annals of the History of Computing* 26.1 (2004), pp. 8–19.

[205] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[206] Christopher D Manning. "Part-of-speech tagging from 97% to 100%: is it time for some linguistics?" In: *International conference on intelligent text processing and computational linguistics*. Springer. 2011, pp. 171–189.

[207] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a large annotated corpus of English: The Penn Treebank". In: *Computational linguistics* 19.2 (1993), pp. 313–330.

[208] Diana Marosin, Marc Van Zee, and Sepideh Ghanavati. "Formalizing and modeling enterprise architecture (EA) principles with goal-oriented requirements language (GRL)". In: *International Conference on Advanced Information Systems Engineering*. Springer. 2016, pp. 205–220.

[209] Philippe Martin. "Knowledge representation in CGLF, CGIF, KIF, frame-CG and formalized-english". In: *International Conference on Conceptual Structures*. Springer. 2002, pp. 77–91.

[210] Cathy Maugis, Gilles Celeux, and Marie-Laure Martin-Magniette. "Variable selection for clustering with Gaussian mixture models". In: *Biometrics* 65.3 (2009), pp. 701–709.

[211] Catherine Meadows. "Applying formal methods to the analysis of a key management protocol". In: *Journal of Computer Security* 1.1 (1992), pp. 5–35.

[212] Juan Jose Mendoza Santana. "Construct by Contract: An Approach for Developing Reliable Software". MA thesis. National University of Ireland, Maynooth, 2013.

[213] Juan Jose Mendoza Santana. "Tototl: A tool for automated formalization and verification of NL specifications". MA thesis. University of St Andrews, 2014.

[214] Juan Jose Mendoza Santana and Juliana Küster Filipe Bowles. "A logic-based approach to software development". In: *COCONAT 2015*. Conference on Computing Natural Reasoning. 2015.

[215] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142.

[216] Bertrand Meyer. "Dependable software". In: *Dependable Systems: Software, Computing, Networks*. Springer, 2006, pp. 1–33.

[217] Bertrand Meyer. *Design by contract*. Prentice Hall, 2002.

[218] Bertrand Meyer. "Eiffel: A language and environment for software engineering". In: *Journal of Systems and Software* 8.3 (1988), pp. 199–246.

[219] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall New York, 1988.

[220] Farid Meziane and Sunil Vadera. "Artificial intelligence in software engineering: Current developments and future prospects". In: *Artificial intelligence applications for improved software engineering development: New prospects*. IGI Global, 2010, pp. 278–299.

[221] Huaikou Miao, Ling Liu, and Li Li. "Formalizing UML models with Object-Z". In: *International Conference on Formal Engineering Methods*. Springer. 2002, pp. 523–534.

[222] George Miller. *WordNet: An electronic lexical database*. MIT press, 1998.

[223] Guy W Mineau, Bernard Moulin, and John F Sowa. "Conceptual Graphs for Knowledge Representation First International Conference on Conceptual Structures, ICCS'93 Quebec City, Canada, August 4–7, 1993 Proceedings". In: *Conference proceedings ICCS-ConceptStruct*. Springer. 1993, p. 12.

[224] Enrique A Miranda et al. "Using reverse engineering techniques to infer a system use case model". In: *Journal of Software: Evolution and Process* 31.2 (2019), e2121.

[225] Aditi Mithal and Ponnurangam Kumaraguru. "Optical character recognition tool". In: (2017).

[226] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[227] John D Musa. "A theory of software reliability and its application". In: *IEEE transactions on software engineering* 3 (1975), pp. 312–327.

[228] John D. Musa and A. Frank Ackerman. "Quantifying software validation: when to stop testing?" In: *IEEE Software* 6.3 (1989), pp. 19–27.

[229] Glenford J Myers et al. *The art of software testing*. Vol. 2. Wiley Online Library, 2004.

[230] Joseph Myers. "Apple v. microsoft: Virtual identity in the gui wars". In: *Richmond Journal of Law & Technology* 1.1 (1995), p. 5.

[231] Sastry Nanduri and Spencer Rugaber. "Requirements validation via automated natural language parsing". In: *Journal of Management Information Systems* 12.3 (1995), pp. 9–19.

[232] Peter Naur. "Software engineering". In: *Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. 1969.

[233] Jiquan Ngiam et al. "Multimodal deep learning". In: *Proceedings of the 28th international conference on machine learning (ICML-11)*. 2011, pp. 689–696.

[234] Oscar Nierstrasz. "A Survey of Object-Oriented Concepts". In: *Object-oriented Concepts, Databases, and Applications*. Ed. by Won Kim and F. H. Lochovsky. New York, NY, USA: ACM, 1989, pp. 3–21.

[235] Bashar Nuseibeh and Steve Easterbrook. "Requirements engineering: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 35–46.

[236] OMG. *Semantics of Business Vocabulary and Business Rules (SBVR)*. Tech. rep. OMG, 2008.

[237] OMG. *Semantics of Business Vocabulary and Business Rules (SBVR), V1.4. Annex E - Overview of the Approach*. Tech. rep. OMG, 2016.

[238] Ashwin Panchapakesan, Rami Abielmona, and Emil Petriu. "A python-based design-by-contract evolutionary algorithm framework with augmented diagnostic capabilities". In: *Evolutionary Computation (CEC), 2013 IEEE Congress on*. IEEE. 2013, pp. 2517–2524.

[239] Shashank Pandit et al. "Netprobe: a fast and scalable system for fraud detection in online auction networks". In: *Proceedings of the 16th international conference on World Wide Web*. ACM. 2007, pp. 201–210.

[240] Kui-Hong Park, Yong-Jae Kim, and Jong-Hwan Kim. "Modular Q-learning based multi-agent cooperation for robot soccer". In: *Robotics and Autonomous Systems* 35.2 (2001), pp. 109–122.

[241] Tom Pender, Eugene McSheffrey, and Lou Varveris. *UML bible*. Vol. 1. Wiley New York, 2003.

[242] Paulo F Pires et al. "Integrating ontologies, model driven, and CNL in a multi-viewed approach for requirements engineering". In: *Requirements Engineering* 16.2 (2011), pp. 133–160.

[243] Amir Pnueli. "Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends". In: *Current trends in Concurrency*. Springer, 1986, pp. 510–584.

[244] Klaus Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated, 2010.

[245] Klaus Pohl et al. "Applying AI techniques to requirements engineering: The NATURE prototype". In: *Proceedings ICSE-Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence*. 1994.

[246] Hendrik Post et al. "Linking functional requirements and software verification". In: *2009 17th IEEE International Requirements Engineering Conference*. IEEE. 2009, pp. 295–302.

[247] Richard Power. "OWL Simplified English: a finite-state language for ontology editing". In: *International Workshop on Controlled Natural Language*. Springer. 2012, pp. 44–60.

[248] Aswin Pranam. "Software Development Methodologies". In: *Product Management Essentials*. Springer, 2018, pp. 65–74.

[249] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[250] Zhihua Qu. *Cooperative control of dynamical systems: applications to autonomous vehicles*. Springer Science & Business Media, 2009.

[251] Truong Ho-Quang et al. "Automatic classification of uml class diagrams from images". In: *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 1. IEEE. 2014, pp. 399–406.

[252] Anna Queralt and Ernest Teniente. "Reasoning on UML class diagrams with OCL constraints". In: *International Conference on Conceptual Modeling*. Springer. 2006, pp. 497–512.

[253] Ernst Rabel. "The Statute of Frauds and Comparative Legal History". In: *LQ Rev.* 63 (1947), p. 174.

[254] Lawrence R Rabiner and Biing-Hwang Juang. "An introduction to hidden Markov models". In: *ieee assp magazine* 3.1 (1986), pp. 4–16.

[255] Steven R Rakitin. *Software verification and validation for practitioners and managers*. Artech House, Inc., 2001.

[256] Ravi Ramamoorthi and James Arvo. "Creating generative models from range images". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. Citeseer. 1999, pp. 195–204.

[257] M Ranzato et al. "On deep generative models with applications to recognition". In: (2011).

[258] Manny Rayner et al. "OMG unified modeling language specification". In: *Version 1.3,© 1999 Object Management Group, Inc.* Citeseer. 2005.

[259] Jeremy Reimer. "A History of the GUI". In: *Ars Technica* 5 (2005), pp. 1–17.

[260] Mark Reynolds et al. "A decidable temporal logic of parallelism". In: *Notre Dame Journal of Formal Logic* 38.3 (1997), pp. 419–436.

[261] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Bell Laboratories, 1975.

[262] José Matías Rivero et al. "From mockups to user interface models: an extensible model driven approach". In: *International Conference on Web Engineering*. Springer. 2010, pp. 13–24.

[263] Guido Rossum. "Python reference manual". In: (1995).

[264] Raymond J Rubey, Joseph A Dana, and Peter W Biche. "Quantitative aspects of software validation". In: *IEEE Transactions on Software Engineering* 2 (1975), pp. 150–155.

[265] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[266] Nayan B Ruparelia. "Software development lifecycle models". In: *ACM SIGSOFT Software Engineering Notes* 35.3 (2010), pp. 8–13.

[267] Mrinmaya Sachan et al. "Learning answer-entailing structures for machine comprehension". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Vol. 1. 2015, pp. 239–249.

[268] Hossein Saiedian. "An invitation to formal methods". In: *Computer* 29.4 (1996), pp. 16–17.

[269] Jean E Sammet. "The early history of COBOL". In: *History of programming languages I*. ACM. 1978, pp. 199–243.

[270] Douglas C Schmidt. "Model-driven engineering". In: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), p. 25.

[271] Bran Selic. "The pragmatics of model-driven development". In: *IEEE software* 20.5 (2003), pp. 19–25.

[272] Claude Elwood Shannon. "A mathematical theory of communication". In: *Bell system technical journal* 27.3 (1948), pp. 379–423.

[273] Shreta Sharma and SK Pandey. "Integrating AI techniques in requirements phase: a literature review". In: (2014).

[274] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking safety properties using induction and a SAT-solver". In: *International conference on formal methods in computer-aided design*. Springer. 2000, pp. 127–144.

[275] Zvi Shiller and Y-R Gwo. "Dynamic motion planning of autonomous vehicles". In: *IEEE Transactions on Robotics and Automation* 7.2 (1991), pp. 241–249.

[276] Keng Siau and Lihyunn Lee. "Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML". In: *Requirements engineering* 9.4 (2004), pp. 229–237.

[277] Katja Siegemund et al. "Towards ontology-driven requirements engineering". In: *Workshop semantic web enabled software engineering at 10th international semantic web conference (ISWC), Bonn*. 2011.

[278] Mathias Soeken et al. "Verifying UML/OCL models using Boolean satisfiability". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2010, pp. 1341–1344.

[279] Richard Soley et al. "Model driven architecture". In: *OMG white paper* 308.308 (2000), p. 5.

[280] Ian Sommerville. *Software Engineering*. 6th. Addison-Wesley, 2001.

[281] Ian Sommerville. *Software Engineering*. 10th. Pearson Education, 2015.

[282] Volker Sperschneider and Grigoris Antoniou. *Logic; A Foundation for Computer Science (International Computer Science Series)*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[283] Richard S Stephens. "Probabilistic approach to the Hough transform". In: *Image and vision computing* 9.1 (1991), pp. 66–71.

[284] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[285] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[286] CMMI Product Team. "Capability maturity model® integration (CMMI SM), version 1.1". In: *CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1. 1)* (2002).

[287] Toby J Teorey, Dongqing Yang, and James P Fry. "A logical design methodology for relational databases using the extended entity-relationship model". In: *ACM Computing Surveys (CSUR)* 18.2 (1986), pp. 197–222.

[288] Richard H Thayer, Sidney C Bailin, and M Dorfman. *Software requirements engineerings*. IEEE Computer Society Press, 1997.

[289] Walter F. Tichy, Mathias Landhäußer, and Sven J. Körner. "Nlrp-bench: A benchmark for natural language requirements processing". In: *Software-engineering and management 2015*. Ed. by Uwe Aßmann et al. Bonn: Gesellschaft für Informatik e.V., 2015, pp. 159–164.

[290] Walter F Tichy and Sven J Koerner. "Text to software: developing tools to close the gaps in software engineering". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. 2010, pp. 379–384.

[291] Kristina Toutanova and Christopher D Manning. "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger". In: *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13*. Association for Computational Linguistics. 2000, pp. 63–70.

[292] Kristina Toutanova et al. "Feature-rich part-of-speech tagging with a cyclic dependency network". In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*. Association for Computational Linguistics. 2003, pp. 173–180.

[293] David A Van Dyk and Xiao-Li Meng. "The art of data augmentation". In: *Journal of Computational and Graphical Statistics* 10.1 (2001), pp. 1–50.

[294] Moshe Y Vardi. "An automata-theoretic approach to linear temporal logic". In: *Logics for concurrency*. Springer, 1996, pp. 238–266.

[295] Moshe Y Vardi. "Why is modal logic so robustly decidable?" In: *Descriptive Complexity and Finite Models*. Ed. by N. Immerman and P. Kolaitis. 1997.

[296] Jos B Warmer and Anneke G Kleppe. *The Object Constraint Language: Precise modeling with UML*. {Addison-Wesley Professional}, 1998.

[297] Jos Warmer and Anneke Kleppe. "The object constraint language second edition: Getting your models ready for MDA". In: *Canada: Person Education, Inc* (2003).

[298] Sholom M Weiss, Ioannis Kapouleas, and JW Shavlik. "An empirical comparison of pattern recognition, neural nets and machine learning classification methods". In: *Readings in machine learning* (1990), pp. 177–183.

[299] Sandra Williams, Richard Power, and Allan Third. "How easy is it to learn a controlled natural language for building a knowledge base?" In: *International Workshop on Controlled Natural Language*. Springer. 2014, pp. 20–32.

[300] Guido Wimmel and Jan Jürjens. "Specification-based test generation for security-critical systems using mutations". In: *International Conference on Formal Engineering Methods*. Springer. 2002, pp. 471–482.

[301] Niklaus Wirth. "The programming language Pascal". In: *Acta informatica* 1.1 (1971), pp. 35–63.

[302] Ian H Witten et al. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[303] Janet Woodcock. *Software engineering mathematics*. CRC Press, 2014.

[304] Jim Woodcock et al. "Formal methods: Practice and experience". In: *ACM computing surveys (CSUR)* 41.4 (2009), p. 19.

[305] Yonghui Wu et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". In: *arXiv preprint arXiv:1609.08144* (2016).

[306] Wenpeng Yin, Sebastian Ebert, and Hinrich Schütze. "Attention-based convolutional neural network for machine comprehension". In: *arXiv preprint arXiv:1602.04341* (2016).

[307] Eric Yu and John Mylopoulos. "Why goal-oriented requirements engineering". In: *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*. Vol. 15. 1998, pp. 15–22.

[308] Alan L Yuille et al. "Determining generative models of objects under varying illumination: Shape and albedo from multiple images using SVD and integrability". In: *International Journal of Computer Vision* 35.3 (1999), pp. 203–222.

[309] Du Zhang and Jeffrey JP Tsai. "Machine learning and software engineering". In: *Software Quality Journal* 11.2 (2003), pp. 87–119.

[310] Min-Ling Zhang and Zhi-Hua Zhou. "ML-KNN: A lazy learning approach to multi-label learning". In: *Pattern recognition* 40.7 (2007), pp. 2038–2048.

[311] Will Y Zou et al. "Bilingual word embeddings for phrase-based machine translation". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 2013, pp. 1393–1398.

[312] Didar Zowghi and Chad Coulin. "Requirements elicitation: A survey of techniques, approaches, and tools". In: *Engineering and managing software requirements*. Springer, 2005, pp. 19–46.