Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Formal reasoning over class models using TOMM

Juan Jose Mendoza Santana[a] Juliana Küster Filipe Bowles[a]

a. School of Computer Science, University of St Andrews, UK

Abstract Class diagrams are widely used in modelling and system design. They capture the relation between the requirements specification (problem domain) and system components (solution domain). However, constant changes to requirements and manual modelling may result in invalid software models, and potentially invalid software solutions. We propose an automated approach at the meta-model level to reason about the validity of diagrams and/or their associated requirements. This paper introduces the foundations of the formal framework TOMM, and illustrates how it can be used for validation of class diagram based models, and potentially extended for model generation and comparison.

Keywords Framework; Class diagrams; Requirements; Formalisation; Validation

1 Introduction

Models are used to design, document, communicate, verify, test, and guide the implementation of software systems, specially in the context of Model-Driven Development (MDD) [Sch06]. Key to these developments has been the Unified Modelling Language (UML) [17b], which provides a series of diagrams that capture different perspectives of software systems. In particular, class diagrams describe the relation between the problem domain (requirements) and the solution domain (software systems), which is the reason why they are widely used and actively researched [BC11; C+08; BMS10; GBR07; MLL02; MLL02; CE06].

Formal verification can be used to determine the *correctness* of a class diagram through properties such as satisfiability, i.e., a model is satisfiable if has at least one correct non-empty instantiation [C+08]. While existing work frequently treats class diagrams as isolated artefacts essentially focusing on internal properties, little has been done to reason about the relation between diagrams and requirements. As a consequence of this, it is possible to have a correct class diagram that does not model correctly the requirements it is supposed to, i.e., a correct but invalid diagram.

We address this problem here by proposing a new framework which integrates several kinds of formal reasoning, including model verification and validation. Our framework embraces the relation amongst artefacts, such as requirements specifications and diagrams, all of them expressed in a common language in order to provide a comprehensive context for formal reasoning. The problem of diagram validation is also tackled by proposing an axiomatic system that allows us to establish the validity of a class diagram with respect to its corresponding requirements specification.

This paper is structured as follows. In Section 2 we discuss the existing work on model verification and highlight the contributions we make in this paper. An overview of the components of our framework is included in Section 3. In Section 4, we describe the notation used to capture requirements specifications and their corresponding formalisation. We discuss class diagram formalisation in Section 5, and model validity in Section 6. The evaluation of our requirements specifications and validation calculus is given in Section 7. Finally, in Section 8 we summarise the research presented here and outline further ongoing developments.

2 Related Work and Contributions

According to Balaban [BMS10], model *correctness* is denoted by a non-empty finite reality. His work includes both the analysis of *consistency* (checking for non-emptiness) and *finite satisfiability* (checking for termination). In a different approach, Cabot [CCR07] deals with strong and weak satisfiability, as well as redundant constraints, using Constraint Logic Programming [JL87].

In addition, there have been several attempts to include UML's constraint language OCL in order to verify UML models, including the work of Gogolla [GBR07], Soeken [Soe+10], Miao [MLL02], and Clavel [CE06]. To this end, they propose different types of formal encoding for class diagrams to enable the use of SAT/SMT solvers [DB11], or to perform model checking [Lam77; CGR11].

In spite of all the work done regarding model verification, little has been explicitly done regarding model validation. While it is clear that the former refers to well-defined correctness, the proper definition of model validation is still unclear. To address this limitation we propose one new definition in Section 6.

In addition to model verification, there is another topic that is relevant for our work. In order to provide a context for model validation, it is necessary to discuss requirements specifications. The CHAOS report [Int95] and the work of Jorgensen [JM06] and Fuchs [FSS98] make it evident that the success of a software project is closely related to the quality of its requirements. Similarly, the construction of valid models depends on the specification of precise requirements.

We distinguish three categories of requirements specifications by surveying the literature [KS98; Pre05; Som15]. *Graphical specifications* include prototypes and diagrams, *textual specifications* correspond to written documents, and *formal specifications* use mathematical models. These categories are characterised by the type of notations used, mainly graphical notations, natural or formal languages.

Class diagrams are sometimes shown as graphical specifications, however, they are not meant to represent explicitly the user needs which are more commonly expressed as textual specifications. Neither of these two formats can be used to formally reason about requirements, hence the need to count on formal specifications as well.

The elements to draw class diagrams are unique and clearly defined in the UML language specification [17b]. In contrast, there are numerous alternatives for textual and formal specifications, which we discuss next.

Textual specifications can be written in structured documents, as the ones proposed by the IEEE in the standard ISO/IEC/IEEE 29148 [ISO11]. This proposal differentiates the structure of *stakeholder*, *system*, and *software* requirements specification documents in order to provide a complete set of perspectives for the requirements. In our work (Section 4.1) we have been inspired by this standard to propose our document structure.

In addition to the structure of a document, we have proposed a Controlled Natural Language (CNL) to be used to limit textual specifications. CNLs impose constraints over Natural Language, in order to enforce the usage of well-defined vocabulary and grammatical rules that reduce ambiguities and enable computability, while maintaining the familiarity of natural language. An example of CNL is the *SBVR Structured English* [17a], also proposed by the OMG, and characterised by its use of visual elements, such as colouring and underlining, to identify components within a sentence. CNLs have demonstrated how computational engines can be used to predict, parse, check and correct requirements, as well as enable a more nature interaction with software models. We present a CNL targeted to describe concise requirements specifications within structured documents (Section 4.1).

In order to perform formal validation of class diagrams and requirements specifications, it is necessary for both of them to be written in compatible formal notations. Existing work describes several alternatives to formalise Class Diagrams [Bre+97; SF97; Cal+02; OD08]. They make use of mathematical constructs such as ontologies and topological functioning model, and languages including Z and description logics. Trying to extend any of these formalisms towards requirements specifications constitutes a challenge that can be addressed by considering a notation that natively includes elements of specifications and diagrams alike. Here we use predicate logic for this purpose.

A logic system is integrated by a formal language with clear semantics and syntax, and an inference engine that can be used to reason about statements within the system [Fit12; SA91; Cro90]. Predicate logic extends the elements of propositional logic to include quantifiers and predicates [Woo14]. In Section 4 and Section 5 we discuss the predicates that integrate our validation framework.

The significant existing work on requirements engineering [GMB94; FKV91; ZLL02; HJR02; PD06; BMM19; Ble+18], CNLs [17a; FSS98; WS09; Cla+05], and class diagram extraction [IA10; Cha+09; HA12; Ben+16], formalisation and reasoning [Szl06; CCR07; Cal+02; MGB05; MC01], makes us wonder about the need for yet another effort. However, the novelty of our approach resides on its holistic design, which aims to put all these efforts together in one seamless framework that will be: a) usable for non-experts, b) applicable for non-critical systems, c) extensible to support other software models, d) an interface to ease usage of formal methods, and e) capable of supporting different reasoning tasks (namely validation, verification, comparison, refinement, etc.). Though these properties are yet to be developed and proved, the foundations presented in this paper clearly are an initial step in that direction.

3 TOMM

Thinking of Models and More (TOMM) is a framework that seeks to integrate formal reasoning over software models and specifications in a seamless way. In other words, it aims to enable different tasks, such as validation, verification, generation and comparison over models and specifications captured in a common formal representation. This is achieved through the interaction of several research areas, such as natural language processing, formal methods, and machine learning in the context of model-driven development.

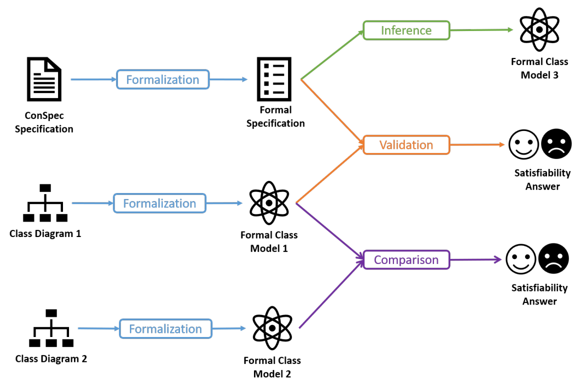

Figure 1 – TOMM conceptual design

The vision for TOMM is to make it possible to handle requirements specifications and class diagrams in a flexible and integrated way. To achieve this, both requirements specifications and class diagrams are captured at the meta-models level, and formalised in predicate logic. The formalisation enables us to do inference, validation and comparison of models. Figure 1 shows the different artefacts within our framework and how they interact with each other.

In addition to Class Diagrams, the inputs for TOMM include ConSpec Specifications as a representation of requirements. These are written using a language and a document structure designed to ease formalisation and interaction with models. And they are properly introduced in the following section.

4 Requirements Formalisation

Software requirements are typically expressed informally, using textual or graphical languages, as discussed in Section 2. The IEEE has provided a guidance for complete and well-structured requirement documents in the ISO/IEC/IEEE 29148 standard [ISO11]. However, this guidance, though intuitive, is not optimal for formal reasoning, and does not hold any explicit relationship with class diagrams. We address these limitations in Section 4.1, proposing an original CNL oriented to capture functional requirements specifications and a structured document inspired by legal contracts and the IEEE standard.

4.1 SpeCNL and ConSpec

SpeCNL is proposed here as a Controlled Natural Language to capture functional requirements in a semi-formal and intuitive manner through well-defined *parts of speech*, *concepts*, and *sentences*.

ConSpec is a document structure which provides a context in which SpeCNL requirements can be expressed, and mapped to elements of class diagrams. This document is based on legal contracts, in which the obligations and responsibilities of the parts are defined in clearly stated clauses. A ConSpec document is composed by a *title*, the current *version* of the document, and a set of *clauses* that describe the functional requirements.

Every clause in a ConSpec document is identified by a unique *clause number*, and must include one *activity* and the *actors* that perform such activity. In addition, every clause can contain *pre-conditions*, *activity conditions*, and *post-conditions* to indicate the constraints to be satisfied before, during, and after the execution of the activity. *Consequences* in case these constraints are violated can be included, as well as *dependencies* with other clauses.

The library system introduced by Callan [Cal94] has been used repeatedly in related work [HG00; KD99; BC11], and we make use of it in Text 1 to demonstrate our document structure for functional requirements, together with our CNL.

Text 1 – Segment of Callan's requirements for a library system[Cal94]

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)*. An item can be borrowed, reserved or renewed to extend a current loan.

In order to manually translate these requirements into a ConSpec specification, it is first necessary to identify all potential activities, which are coloured in red, potential entities are coloured in **blue** and potential attributes are coloured in *grey*. Using these identified elements, we manually generate the equivalent SpeCNL elements, and locate them in their corresponding field within the ConSpec specification.

The first step to generate ConSpec specifications is to is to simplify the existing requirements into sentences closer to SpeCNL. Figure 2 shows the process to manually simplify the sentence *"An item can be borrowed"* into the sentence *"Customers borrow loan-items"*. Some interpretations and assumptions were required here; for instance, we assume that *"Customers"* are the subject of the sentence, and that *"item"* and *"loan-items"* are equivalent words.

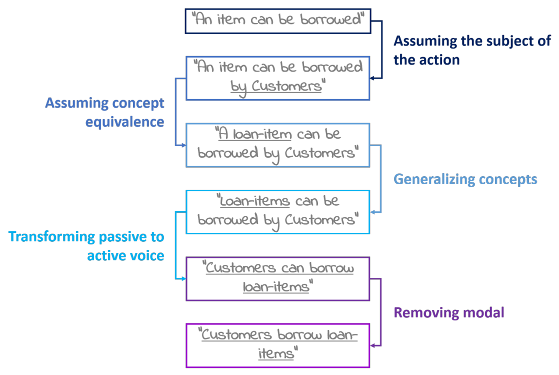
"An item can be borrowed"
Assuming the subject of the action
"An item can be borrowed by Customers"
Assuming concept equivalence
"A loan-item can be borrowed by Customers"
Generalizing concepts
"Loan-items can be borrowed by Customers"
Transforming passive to active voice
"Customers can borrow loan-items"
Removing modal
"Customers borrow loan-items"

Figure 2 – Example of manual simplification of sentences

Though this process requires some cognitive effort, the tasks proposed are intuitive, and they can help to detect potential problems within the requirements.

The simplified sentence *"Customers borrow loan-items"* is then used to generate a ConSpec clause, using the grammar rules of SpeCNL included in Tables 4, 5, 6 and 7. The parsing trees shown in Figure 3 illustrate how these rules are used with to parse this simplified sentences into the elements of a clause.

Following these steps, simplification and parsing, the original requirements are translated into a ConSpec specification which is shown in ConSpec 1. With this example we also show how ConSpec encourages the stakeholders (users, sponsors, engineers, etc.) to be clear and precise about what the expected functionality is.

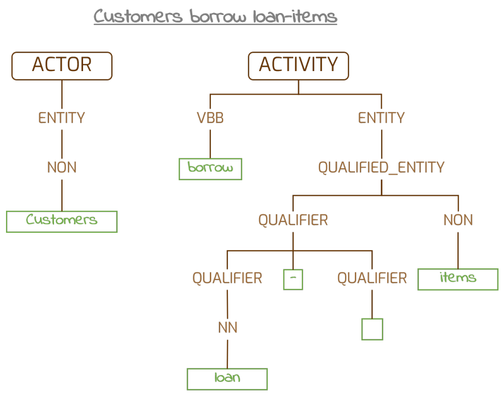

Figure 3 – Parsing trees for sample activity and actors

Reading ConSpec documents like this one is an intuitive activity for the stakeholders, no different from reading any other text in English. Writing ConSpec documents, however, requires knowledge about the rules of SpeCNL and the structure of ConSpec (Tables 4, 5, 6 and 7). This can be eased with the help of tools to edit, predict, collect and validate specifications, which is part of our work currently in progress.

ConSpec 1 – Segment of the manual translation of Text 1 into ConSpec specification

```
Title: Callan Library System
Version: 1.0
Clauses:
      - C1:
            Action: reserve loan-item
            Actors:
                  - Customers
            Preconditions:
                  - Customers are members
                  - Members have member-number
                  - Members have name
                  - Members have address
                  - Members have date-of-birth
                  - Books are loan-items
                  - Books must have title
                  - Books must have author
                  - Language-tapes are loan-items
                  - Language-tapes must have language
                  [...]
      - C2:
            Action: borrow loan-item
            Actors:
                  - Customers
            Preconditions:
                  - Loan-item must be available
            Dependencies:
                  - C1
      - C3:
            Action: renew loan-item
            Actors:
                  - Customers
            Preconditions:
                  - Loan-item must be borrowed
      [...]
```

4.2 Formalisation

SpeCNL and ConSpec were designed with the intention to facilitate formalisation of requirements and to establish a relation with class diagrams. For instance, the substitution rule $ACTOR \rightarrow ENTITY$ can be easily transformed into predicate logic as $\mathrm{Actor}(x) \wedge \mathrm{Entity}(x)$. In order to formalise a contract, it is necessary to parse each of its elements using the corresponding substitution rules. The sequence of substitution rules applied is known as the parsing tree [Joh98] of the element.

From Figure 3 we observe that ConSpec documents include only the root and the leaves of the parsing trees. The former corresponds to the clause element being parsed and the later to its value. In this example, the clause elements are *activity* and *actors*, with values *"borrow loan-items"* and *"Customers"* respectively. We use these parsing trees in order to formalise this specification.

For each non-terminal in the transformation rules described in Appendix B and Appendix C, there is a corresponding predicate that takes as argument the terminal used in the corresponding application of rules (leaves). In the case of the parsing trees in Figure 3, the following predicates can be manually extracted.

Example 1 – Predicates for parsing trees shown in Figure 3

$$\begin{aligned}&\mathrm{ACTOR}(Customers) \wedge \mathrm{ENTITY}(Customers) \wedge \mathrm{NON}(Customers) \wedge \\ &\mathrm{ACTIVITY}(borrow, \textit{loan-items}) \wedge \mathrm{VBB}(borrow) \wedge \mathrm{ENTITY}(\textit{loan-items}) \wedge \\ &\mathrm{QUALIFIED\_ENTITY}(\textit{loan-items}) \wedge \mathrm{QUALIFIER}(loan) \wedge \mathrm{NN}(loan) \wedge \ldots\end{aligned}$$

In this way, all the elements of a ConSpec document can be formalised as first-order predicates. However, not all of the predicates generated are required to establish a relation with class models; in Section 6 we discuss relevant predicates for this relation. In the next Section (5) we discuss the notation and the process to formalise class diagrams using first-order predicates.

5 Class diagrams formalisation

In this section we present the elements required to formalise class diagrams into predicates. In this way, diagrams and specifications will be expressed in a common language, making it possible to reason over them.

Class diagrams are part of the Unified Modelling Language standard, and they have clearly defined structures, semantics, and notation. Table 3 in Appendix A contains the pre-defined meta-data from UML and Table 1 describes the predicates required to formalise class diagrams.

Table 1 – Structure of ConSpec document

Class Diagram Component	**Predicate**
Class: indicates that the variable c is the name of a classifier of type $t \in C_U$	$\mathrm{CLS}(c, t)$
Attribute: expressed that class c has an attribute a of type $t \in T$, with visibility $v \in V_U$ and scope $s \in S_U$	$\mathrm{ATR}(c, a, t, v, s)$
Operation: 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 $t \in T \cup void$	$\mathrm{OPR}(c, o, t, v, s, P)$

Table 1 – Structure of ConSpec document

Class Diagram Component	**Predicate**
Type: indicates that t is a type	TYPE(t)
Relation: 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	REL$(s, d, t, n, r, \#_l, \#_u)$
Inheritance: indicates that class g generalises s, or conversely, that s is a specialisation of g	INH(g, s)

In order to show how a diagram can be formalized using these predicates, we use the diagram corresponding to the library example from Section 4. Figure 4 shows the original diagram presented by Callan [Cal94] and Example 2 shows the most significant predicates corresponding to formalisation of this diagram. It is observed that the formalisation of class diagrams is a straight forward mapping from elements of the class model to predicates. In the next sections, the relation between class diagram and requirements will be established using these formalisations.

Example 2 – Predicates for Callan's Class Diagram

$$\begin{array}{l}
\mathrm{CLS}(Book) \wedge \mathrm{TYPE}(Book) \wedge \\
\mathrm{CLS}(LoanItem) \wedge \mathrm{TYPE}(LoanItem) \\
\mathrm{CLS}(LoanTransaction) \wedge \mathrm{TYPE}(LoanTransaction) \\
\mathrm{ATR}(LoanItem, title, \epsilon, \epsilon, instance) \wedge CLS(LoanItem) \wedge \mathrm{TYPE}(\epsilon) \\
\mathrm{ATR}(LoanItem, barcode, String, \epsilon, instance) \\
\mathrm{ATR}(Customer, name, String, \epsilon, instance) \wedge \mathrm{TYPE}(String) \\
\mathrm{ATR}(Book, subject, \epsilon, Public, instance) \\
\mathrm{OPR}(LoanTransaction, borrow, \epsilon, \epsilon, instance, \{\}) \\
\mathrm{OPR}(LoanItem, check\_in, void, Public, classifier, \{(barcode, String)\}) \\
\mathrm{REL}(Library, Membership_Card, Relation, Issues, membercode, \epsilon, \epsilon) \\
\mathrm{REL}(Library, subsection, Aggregation, \epsilon, classmark, \epsilon, \epsilon) \\
\mathrm{REL}(Customer, Book, ClassedRelation, borrows, \epsilon, 0, 8) \\
\mathrm{INH}(LoanItem, Book) \\
\mathrm{INH}(LoanItem, LanguageTape)
\end{array}$$

6 Class diagrams validation

According to Sommerville [Som15], requirements validation is "the process of checking that requirements define the system that the customer really wants". Similarly, through software validation "the software is checked to ensure that it is what the customer requires". There is an evident relation between requirements and software validation, however, these definitions pay no attention to *model validation*, which we informally define as "the process of checking that the models reflect the needs that the customer has expressed in the requirements". More formally, model validity in TOMM is defined as follows.

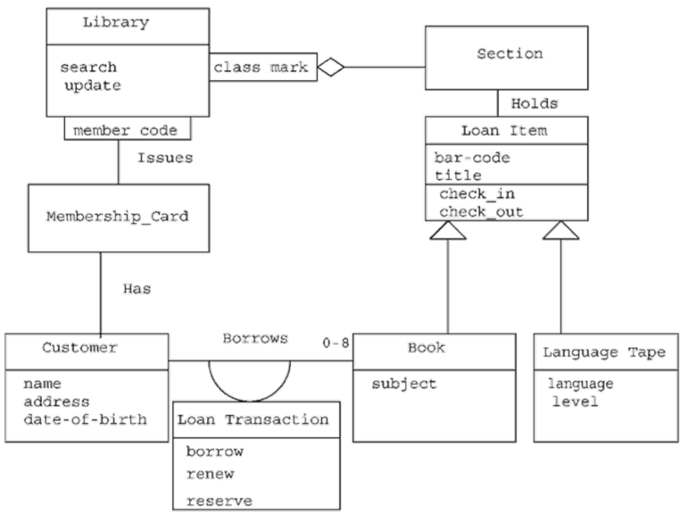

Figure 4 – Library Class Diagram

Definition 1 *A model is **valid** if it is sound and complete.*

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

Where μ is any given model.

Definition 2 *A model is **sound** if all of its elements are derived from the specifications. It is formally expressed as:*

$$Sound(\mu) \implies \forall \phi \in M, \exists \psi_1, \psi_2, ...\psi_n \in S_v \mid \psi_1 \wedge \psi_2 \wedge ...\psi_n \models_R \phi$$

In here ϕ is a predicate from the set M of predicates describing the model, $\psi_1, \psi_2, ...\psi_n$ are predicates from the set S_v of relevant predicates describing the specification, and the symbol $\models_R$ indicates that the predicate ϕ is inferred from the application of some rule R over the predicates $\psi_1, \psi_2, ...\psi_n$

In this way, *soundness* establishes the relation from the predicates of the model μ to the predicates of the specification σ.

Definition 3 *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:*

$$Complete(\mu) \implies \forall \psi \in S_v, \exists \phi_1, \phi_2, ...\phi_n \in M \mid \psi \models_R \phi_1 \wedge \phi_2 \wedge ...\phi_n$$

Inversely to soundness, completeness *establishes the relation from the predicates of the specification σ to the predicates of the model μ.*

Notice that in both definitions above, we refer to the set S_v of relevant predicates from the specification. This is because, as stated in Section 4, not all predicates generated from the parsing tree provide information that contributes to the verification process. For example, *conditional sentences* express constraints which are not part of class diagrams, unless OCL is used. The relevant predicates in S_v are those used in the inference rules presented next.

Definition 4 *The subset S_v of* ***relevant specification predicates*** *for class diagram validity is defined as:*

$$S_v = \{\text{ACTOR}, \text{ACTIVITY}, \text{VBB}, \text{QUALIFIED\_ENTITY}, \\ \text{STRUCTURAL\_SENTENCE}, \text{TYPE\_SENTENCE}\}$$

We have previously referred to the application of some inference rule $\models_R$ to indicate that there is a relation between predicates in M and S_v, this relation is identified by a series of validity axioms that permit to establish completeness and soundness.

The validity axioms about to be described, together with the set of predicates representing a ConSpec specification and a class diagrams, constitute our *validation calculus.* This calculus depends on the definition of *semantic equivalences* between words, represented with the symbol $\approx$, to explicitly state that two words have the same meaning in the context of the problem. For example, in banking system, the words *"customer"* and *"client"* may both refer to the account holders; whereas in marketing systems, the one may refer to the company hiring the system, and the other one to visitors of marketing campaigns. In this way, disambiguation is handled, and valid elements can be related even if their names vary from one representation to another. This is formally defined as follows.

Definition 5 ***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 problem and solution domains. This relation is:*

- *symmetric: $w_1 \approx w_1$*
- *reflexive: if $w_1 \approx w_2$ then $w_2 \approx w_1$*
- *transitive: if $w_1 \approx w_2$ and $w_2 \approx w_3$ then $w_1 \approx w_3$*

The axioms to be defined allow to establish both properties mentioned above: *soundness* and *completeness.* Each soundness axiom corresponds to one element of the class diagram, and it has an inverse axiom to reason about the corresponding part in the specification. Here the soundness axioms are shown together with their inverse (completeness) axioms.

Class and Actor Axioms

The *class axiom* states that there is a relation between the class predicates in M and the actor predicates in S_v.

$$\frac{\forall \gamma_x \in M \exists \alpha_a \in S_v \mid a \approx x}{\top}$$

Where γ_x is any predicate of the form CLASS$(x, *)$ in M and γ_y is any predicate of the form ACTOR(a) and $x \approx a$ is the semantic equivalence between x and a. Note that in this case, the value of the classifier type in the class predicate is indicated as ignored using the symbol $*$, this is because it is assumed to reflect a design choice and not an element of the problem domain, hence it is irrelevant for the axiom. The same assumption and notation are applicable for the rest of the rules.

The inverse of the class axiom is the *actor axiom*, which using the same notation is expressed as:

$$\frac{\forall \alpha_a \in S_v \exists \gamma_x \in M \mid x \approx a}{\top}$$

Operation and Activity Axioms

The *operation axiom* establishes that any operation in M is derived from an activity in S_v

$$\frac{\forall \omega_x \in M \exists \delta_a \in S_v \mid a \approx x}{\top}$$

In this case, the formula ω_x represents any predicate $\mathrm{OPR}(*, x, *, *, *, *)$ in M, and δ_a represents predicates $\mathrm{ACTIVITY}(a, *)$ in S_v.

Inversely, the *activity axiom* is defined as:

$$\frac{\forall \delta_a \in S_v \exists \omega_x \in M \mid a \approx x}{\top}$$

Attribute and Structural Axioms

The *attribute axiom* establishes the relationship between structural sentences and class attributes.

$$\frac{\forall \rho_{x_y} \in M \exists \sigma_{a_b} \in S_v \mid a \approx x \land b \approx y}{\top}$$

In a structural sentence two distinctive elements are required: the entity and the attribute of the entity. They are represented in the formula σ_{a_b}. for the predicate $\mathrm{STRUCTURAL\_SENTENCE}(a, *, b)$. The attribute predicate $\mathrm{ATR}(x, y, *, *, *)$ is captured in the ρ formulas with parameters x, y.

Its corresponding completeness axiom is defined in terms of *actors* as follows:

$$\frac{\forall \sigma_{a_b} \in S_v \exists \rho_{x_y} \in M \mid a \approx x \land b \approx y}{\top}$$

Inheritance and Type Axioms

This axiom establishes the relationship between inheritance predicates and type sentences.

$$\frac{\forall \eta_{x_y} \in M \exists \tau_{a_b} \in S_v \mid a \approx x \land b \approx y}{\top}$$

In this axiom the formula τ represents the $\mathrm{TYPE\_SENTENCE(b, *, a)}$ predicate, and the formula η represents the $\mathrm{INH}(x, y)$ predicate. Variables x and y in eta represent a superclass and a subclass respectively, while b is a subtype and a is a type, as seen in the substitution rules of Table 6.

The completeness axiom related to this is the linked to *type* sentences in the following way:

$$\frac{\forall \tau_{a_b} \in S_v \exists \eta_{x_y} \in M \mid a \approx x \land b \approx y}{\top}$$

With these foundations, we establish some of the relations from the components of a class diagram to the ConSpec specification. Axioms for predicates $\mathrm{TYPE}(t)$ and $\mathrm{REL}(s, d, t, n, r, \#_l, \#_u)$ are not included in this first version of TOMM, but we will de defined on a next version. By checking the current set of axioms against the predicates of the model and the specification respectively, it is possible to check whether a class diagram is complete, sound, and ultimately valid.

7 Evaluation

7.1 Requirements Specification

In order to evaluate our specification format, a series of publicly available requirements were collected and translated into ConSpec and SpeCNL. These requirements were taken from the NLRP-Bench [TLK15] system, which contains specification documents and class diagram available in publications, academic works and some shared by institutions and companies. Requirements were sampled based on their *category*, *length* and *domain* to cover most of the cases. The list of chosen requirements for the evaluation is shown in Table 2.

Table 2 – Sample requirements

Requirements	**Category**	**Length**	**Domain**
Ships	Academic	49 words	Transportation
Trains	Academic	78 words	Transportation
ATM Simulation	Academic	750 words	Banking
ACME Library	Academic	16 pages	Business
Library	Publications	217 words	Business
Steam-Boiler	Publications	10 pages	Hardware
Laws of Chess	Publications	18 pages	Gaming
Whois Protocol	Real System	100 words	Communication protocol
Light Control System	Real System	13 pages	Hardware

Each clause in a ConSpec document is associated with a unique activity; however, the lack of activities in the original "Ships" requirements made it difficult to generate any such clause. Hence, we had to introduce a *neutral action* in which we can list the elements and structure of the original requirements. This neutral action allowed us to generate a clause with activity "exist" and actors "Ships", in which all the other conditions could be expressed.

For brevity, we do not discuss the individual evaluation of all the requirements; instead, we condense our findings here. *Comparison sentences* require to be extended beyond single numeric values and adjectives, to support ranges, and variables. *Conditional sentences* need to support more complex structures, and references to other clauses. *Type sentences* can be simplified, including several sub-types in one single statement. A *neutral action* should be added into ConSpec to collect generic structural components. *Clause conditions* should support references to other classes, and an open field should be included to add information that cannot be written into SpeCNL. Finally, we consider that further research on *machine translation* could help to automate this process.

In spite of the need for some improvements (which will be addressed in the future versions), SpeCNL and ConSpec have successfully demonstrated to be able to cope with various requirements for academic, published and real-life systems for different domains.

7.2 Validation Calculus

We evaluated the reliability of our validation calculus for class diagram analysing common properties of formal systems, i.e. *consistency completeness* and *soundness* [Coo78; Kar93; MA14]. In order to avoid confusion with *model* completeness and soundness, we define these properties for our formal system as follows.

Definition 6 *A* formal system *is **consistent** 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 present within the axioms or the theorems of the system. Our formal system is composed by the sets of predicates representing class diagrams and ConSpec specifications, and the axioms to be checked. Negation of predicates cannot be derived by any means, hence it is impossible for contradictions to be present in our system. Thus it can be concluded that *the system is consistent.*

Definition 7 *A* formal system *is **sound** 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 correspond to individual elements of a class diagram or a specification. If all the possible predicates from the specification were to be considered, then the system would not be sound. However, the introduction of the subset S_v in Section 6 allows to limit these predicates, resulting in the following semantics.

Definition 8 ***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.*

Because the semantics of the system and its actions are the same, we argue that *the system is sound.*

Definition 9 *A* formal system *is **complete** with respect to a given property if every formula having that same property can be derived using that system.*

In other words, the system is complete if all the valid formulas can be generated from the axioms and the inference rules. Our system shows that all valid formulas, representing elements from the specification or the class diagram, are generated from the application of the validation axioms described in Section 6. And since no new formulas can be generated, we argue that our *validation calculus is complete.*

So far, we have evaluated the reliability of our validation calculus based on its formal properties, which allows us to establish a high degree of trust in the system itself, regardless of its usage. We briefly discuss now the *usage* of our calculus in the context of the library example. The formalisation of the class diagram shown in Example 2 is checked against the formalisation of the ConSpec specification shown in Example 1 using the axioms presented in section 6.

First we evaluate *soundness* of the model, i.e., we check that all elements in the model are derived from the specification. We attempt to apply the *class rule* over the predicate CLS(*Loan Transaction*), which requires a predicate ACTOR(*Loan Transaction*) to be present in the specification. Because this predicate is not present in the formalisation of the specification we conclude that the model is not sound.

We then evaluate model *completeness.* As an example we take the predicate ACTOR(*Customers*) from the specification, which requires to have a predicate of the form CLS(x) in the specification, where x is a word equivalent to *"Customers"*. If we assume that $Customer \approx Customers$, then we can conclude that the *actor rule* holds for this particular predicate. If we continue to evaluate all the relevant predicates (Definition 4) in the specification, it results that, with the appropriate equivalences, all predicates are also present in the model; hence, the model is complete.

We then conclude that the class diagram shown in Figure 4 is invalid with respect to the requirements shown in Text 1, because there are elements in the model that do not correspond to the specification.

In this section we evaluated the *reliability* of our validation calculus regarding its formal properties. However, the evaluation of its application (usage) requires us to formalise ConSpec requirements and class diagrams into their corresponding predicates. Though it is, in principle, possible to do this formalisation manually (as shown in this paper), the effort required to do it this way would quickly discourage the usage of our framework; hence, we are currently working on the development of a tool to automate formalisation and validation. A more comprehensive evaluation of the *usage* of our framework will be presented in a future publication about our the development of our tool.

8 Conclusion

In this paper, we introduced a framework for formal reasoning over models and specifications which addresses a major limitation in model verification and validation. The current work includes the formalisation of class diagrams and requirements specifications to validate the relation between them using well-defined predicates and a formal system. Requirements to be formalised must be written in a structured document (ConSpec) using simplified English (SpeCNL), which have also been proposed here.

We are currently extending TOMM to reflect the results presented in this paper. These include additional formal systems to support model inference and model comparison. Tool support is being developed to integrate automated formalisation and reasoning using SMT solvers. Machine learning is being applied to automate translation of requirements into specifications, and to identify semantic equivalences. In addition, model equivalence is being extended to Generative Adversarial Networks (GAN) in order to automate model generation.

A Predefined values for UML Class Diagrams

Table 3 – Class Diagram Predicates

Meta-field	**Possible values**
Classifier types	$C_U = \{class, abstract, interface\}$
Visibility	$V_U = \{+, -, \#, /, \sim, *\}$
Scope	$S_U = \{classifier, instance, \epsilon\}$
Primitive Types	$P_U = \{Integer, Boolean, String, Unlimited\ Natural, Real, \epsilon\}$
Cardinality Symbols	$\#_U = \mathbb{N} \cup \{m, n, \epsilon, *, +, ?\}$
Relation Types	$T_r = \{Association, Dependency, Aggregation, Composition, Realization, Relation, Classed\ Association\}$

B Grammar Rules for SpeCNL

Table 4 – Parts of Speech for SpeCNL

Tag	**Meaning**	**Tag**	**Meaning**
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		

Table 5 – Base Concepts for SpeCNL

Concept	**Substitution Rule**
Modals	MODAL → *can* | *must*
Comparators	COMPARATOR → INEQUALITY *or* EQUALITY | INEQUALITY | EQUALITY | COMPARATOR_SYMBOL INEQUALITY → *greater than* | *less than* EQUALITY → *equal to* COMPARATOR_SYMBOL → > | < | = | >= | <=
Entities	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
Attributes	ATTRIBUTE → ENTITY APOS NON | NEUTRAL_INDICATOR NON *of the* NON
Actions	ACTION → VB | MODAL VB

Table 6 – Sentences for SpeCNL

Sentence	**Structure**
Structural: define the properties of the objects within the problem domain	STRUCTURAL_SENTENCE → NON MODAL *have* STRUCTURAL_ITEM STRUCTURAL_ITEM → ENTITY | ENTITY, STRUCTURAL_ITEM | ENTITY*, and* STRUCTURAL_ITEM
Comparison: used to compare values of attributes	COMPARISON_SENTENCE → ATTRIBUTE CONSTRAINT COMPARATOR NUM | ATTRIBUTE CONSTRAINT ADJ CONSTRAINT → OBLIGATION | POSSIBILITY OBLIGATION → *must be* POSSIBILITY → *can be*
Cardinality: represent limits on the size of collections	CARDINALITY_SENTENCE → ENTITY ACTION LIMIT NUM NON LIMIT → *up to* | *at least* | *maximum* | *minimum*

Table 6 – Sentences for SpeCNL

Sentence	**Structure**
Conditional: identifies the actions to follow after specific cases	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: hierarchical classification of entities	TYPE_SENTENCE → SUBTYPE *are* TYPE SUBTYPE → PLURAL_ENTITY TYPE → PLURAL_ENTITY

C Grammar Rules for ConSpec

Table 7 – Structure of ConSpec document

Clause Element	**Structure**
Clause Number: identifies each requirement	CLAUSE_NUMBER → LETTER_C NUMBER NUMBER → NUMBER DOT NUMBER | NUMBER DIG | DIG
Activity: describes a task that must be supported by the system	ACTIVITY → VBB | VBB ENTITY | TO VBB | TO VBB ENTITY
Actors: represent the entities that can perform the current activity	ACTOR → ENTITY
Conditions: constraints that must be observed before, during, and after the execution of the corresponding activity	PRECONDITION → CONDITION ACTIVITY_CONDITION → CONDITION POSTCONDITION → CONDITION CONDITION → STRUCTURAL_SENTENCE | COMPARISON_SENTENCE | CARDINALITY_SENTENCE | CONDITIONAL_SENTENCE | TYPE_SENTENCE | CONSTRAINT_SENTENCE
Consequences: actions to be taken in case of errors resulting from the current task	CONSEQUENCE → VBB ENTITY | TO VBB ENTITY
Dependencies: indicates a relation between activities that must occur before the current one	DEPENDENCY → LETTER_C NUMBER NUMBER → NUMBER DOT NUMBER | NUMBER DIG | DIG

References

[17a] *Semantics Of Business Vocabulary and Rules (SBVR).* formal/2017-05-05. Available at `https://www.omg.org/spec/SBVR/1.4/PDF`, version 1.4. Object Management Group. May 2017.

[17b] *Unified Modeling Language (UML).* formal/2017-12-05. Available at `https://www.omg.org/spec/UML/2.5.1/PDF`, version 2.5.1. Object Management Group. Dec. 2017.

[BC11] 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.

[Ben+16] Wahiba Ben Abdessalem Karaa et al. "Automatic builder of class diagram (ABCD): an application of UML generation from functional requirements". In: *Software: Practice and Experience* 46.11 (2016), pp. 1443–1458.

[Ble+18] Yoann Blein et al. "Extending specification patterns for verification of parametric traces". In: *Proceedings of the 6th Conference on Formal Methods in Software Engineering.* ACM. 2018, pp. 10–19.

[BMM19] Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer. *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment.* Springer, 2019.

[BMS10] 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.

[Bre+97] Ruth Breu et al. "Towards a formalization of the unified modeling language". In: *European Conference on Object-Oriented Programming.* Springer. 1997, pp. 344–366.

[C+08] Jordi Cabot, Robert Claris, Daniel Riera, et al. "Verification of UML/OCL class diagrams using constraint programming". In: *2008 IEEE International Conference on Software Testing Verification and Validation Workshop.* IEEE. 2008, pp. 73–80.

[Cal+02] Andrea Calì et al. "A formal framework for reasoning on UML class diagrams". In: *International Symposium on Methodologies for Intelligent Systems.* Springer. 2002, pp. 503–513.

[Cal94] Robert E Callan. *Building Object-Oriented Systems: An introduction from concepts to implementation in C++.* Computational Mechanics, 1994.

[CCR07] 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.

[CE06] 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.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[Cha+09] Jayeeta Chanda et al. "Traceability of requirements and consistency verification of UML use case, activity and Class diagram: A Formal approach". In: *2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*. IEEE. 2009, pp. 1–4.

[Cla+05] Peter Clark et al. "Acquiring and Using World Knowledge Using a Restricted Subset of English." In: *Flairs conference*. 2005, pp. 506–511.

[Coo78] Stephen A Cook. "Soundness and completeness of an axiom system for program verification". In: *SIAM Journal on Computing* 7.1 (1978), pp. 70–90.

[Cro90] John N Crossley. *What is mathematical logic?* Courier Corporation, 1990.

[DB11] Leonardo De Moura and Nikolaj Bjørner. "Satisfiability modulo theories: introduction and applications". In: *Communications of the ACM* 54.9 (2011), pp. 69–77.

[Fit12] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[FKV91] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. "Informal and formal requirements specification languages: bridging the gap". In: *IEEE transactions on Software Engineering* 17.5 (1991), pp. 454–466.

[FSS98] 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.

[GBR07] 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.

[GMB94] Sol Greenspan, John Mylopoulos, and Alex Borgida. "On formal requirements modeling languages: RML revisited". In: *Proceedings of 16th International Conference on Software Engineering*. IEEE. 1994, pp. 135–147.

[HA12] Hatem Herchi and Wahiba Ben Abdessalem. "From user requirements to UML class diagram". In: *arXiv preprint arXiv:1211.0713* (2012).

[HG00] 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.

[HJR02] Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. "An authoring tool for informal and formal requirements specifications". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2002, pp. 233–248.

[IA10] Mohd Ibrahim and Rodina Ahmad. "Class diagram extraction from textual requirements using Natural language processing (NLP) techniques". In: *2010 Second International Conference on Computer Research and Development*. IEEE. 2010, pp. 200–204.

[Int95] The Standish Group International. *The CHAOS Report (1994)*. Tech. rep. The Standish Group, 1995.

[ISO11] IEC ISO. *IEEE. 29148: 2011-Systems and software engineering-Requirements engineering*. Tech. rep. IEEE, 2011.

[JL87] Joxan Jaffar and J-L Lassez. “Constraint logic programming”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1987, pp. 111–119.

[JM06] 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.

[Joh98] Mark Johnson. “PCFG models of linguistic tree representations”. In: *Computational Linguistics* 24.4 (1998), pp. 613–632.

[Kar93] G Neelakantan Kartha. “Soundness and Completeness Theorems for Three Formalizations of Action.” In: *IJCAI*. Vol. 93. 1993, pp. 724–729.

[KD99] 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.

[KS98] Gerald Kotonya and Ian Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[Lam77] Leslie Lamport. “Proving the correctness of multiprocess programs”. In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143.

[MA14] Maria Manzano and Enrique Alonso. “Completeness: from Gödel to Henkin”. In: *History and Philosophy of Logic* 35.1 (2014), pp. 50–75.

[MC01] William E McUmber and Betty HC Cheng. “A general framework for formalizing UML with formal languages”. In: *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society. 2001, pp. 433–442.

[MGB05] Tiago Massoni, Rohit Gheyi, and Paulo Borba. “Formal refactoring for UML class diagrams”. In: *Proceedings of the 19th Brazilian Symposium on Software Engineering*. 2005, pp. 152–167.

[MLL02] 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.

[OD08] Janis Osis and Uldis Donins. “Formalization of the UML class diagrams”. In: *International Conference on Evaluation of Novel Approaches to Software Engineering*. Springer. 2008, pp. 180–192.

[PD06] Christophe Ponsard and Emmanuel Dieul. “From Requirements Models to Formal Specifications in B.” In: *ReMo2V* 241 (2006).

[Pre05] Roger S Pressman. *Software engineering: a practitioner’s approach*. Palgrave Macmillan, 2005.

[SA91] Volker Sperschneider and Grigoris Antoniou. *Logic; A Foundation for Computer Science (International Computer Science Series)*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[Sch06] Douglas C Schmidt. “Model-driven engineering”. In: *COMPUTER-IEEE COMPUTER SOCIETY-* 39.2 (2006), p. 25.

[SF97] Malcolm Shroff and Robert B France. "Towards a formalization of UML class structures in Z". In: *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*. IEEE. 1997, pp. 646–651.

[Soe+10] 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.

[Som15] Ian Sommerville. *Software Engineering*. 10th. Pearson Education, 2015.

[Szl06] Marcin Szlenk. "Formal semantics and reasoning about uml class diagram". In: *2006 International Conference on Dependability of Computer Systems*. IEEE. 2006, pp. 51–59.

[TLK15] Walter F Tichy, Mathias Landhäußer, and Sven J Körner. *nlrpBENCH: a benchmark for natural language requirements processing*. Gesellschaft für Informatik eV, 2015.

[Woo14] Janet Woodcock. *Software engineering mathematics*. CRC Press, 2014.

[WS09] Colin White and Rolf Schwitter. "An update on PENG light". In: *Proceedings of the Australasian Language Technology Association Workshop 2009*. 2009, pp. 80–88.

[ZLL02] Marc K Zimmerman, Kristina Lundqvist, and Nancy Leveson. "Investigating the readability of state-based formal requirements specification languages". In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE. 2002, pp. 33–43.

About the authors

Juan Jose Mendoza Santana recently completed his PhD at the School of Computer Science, University of St Andrews, researching novel and efficient combinations between Software Engineering, Formal Methods and Natural Language Processing. Contact him at `jjm20@st-andrews.ac.uk`, or visit `https://jjm20.host.cs.st-andrews.ac.uk`.

Juliana Küster Filipe Bowles is a senior lecturer at the School of Computer Science, University of St Andrews. Her interests include formal logics and reasoning, applications of formal methods to healthcare and critical systems. She leads the school's Health Informatics research group, and coordinates the EU Horizon 2020 research project on *Securing Medical Data in Smart Patient-Centric Healthcare Systems (Serums)*. Contact her at `jkfb@st-andrews.ac.uk` or visit `https://juliana.host.cs.st-andrews.ac.uk`.