UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

COLEGIADO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO

Matheus Lenke Coutinho

Tonto: A Textual Language for Ontology-Driven Conceptual Modeling

Vitória, ES

2023

Matheus Lenke Coutinho

Tonto: A Textual Language for Ontology-Driven Conceptual Modeling

Monografia apresentada ao Curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Colegiado do Curso de Ciência da Computação

Supervisor: Prof. Dr. João Paulo Andrade Almeida

Vitória, ES

2023

Matheus Lenke Coutinho
Tonto: A Textual Language for Ontology-Driven Conceptual Modeling/ Matheus Lenke Coutinho. – Vitória, ES, 2023-
86 p. : il. (algumas color.) ; 30 cm.

Supervisor: Prof. Dr. João Paulo Andrade Almeida

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Colegiado do Curso de Ciência da Computação, 2023.

1. Conceptual modeling. 2. OntoUML. 3. DSL. 4. Textual Syntax. 5. Langium IV. Tonto: A Textual Language for Ontology-Driven Conceptual Modeling

CDU 02:141:005.7

Matheus Lenke Coutinho

Tonto: A Textual Language for Ontology-Driven Conceptual Modeling

Monografia apresentada ao Curso de Ciência da Computação do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Trabalho aprovado. Vitória, ES, 19 de julho de 2023:

Prof. Dr. João Paulo Andrade Almeida
Orientador – Departamento de Informática, UFES

Profa. Dra. Renata S. S. Guizzardi
Membro da Banca – Departamento de Informática, UFES

Profa. Dra. Monalessa Perini Barcellos
Membro da Banca – Departamento de Informática, UFES

Vitória, ES
2023

Dedico este trabalho em memória de meus avós, Laura, Solange e Antônio, que me viram chegar até a universidade mas precisaram ver minha conclusão de um lugar melhor.

Acknowledgements

Diversos momentos da vida somos obrigados a passar por fases que aprofundam o que sabemos sobre nós mesmos. Nada seria sem as pessoas à minha volta, e agradeço a todos que fizeram parte da minha jornada.

Agradeço primeiramente à minha família, à quem devo quem sou hoje e onde estou, sendo meu primeiro refúgio. Sem eles, eu não estaria aqui hoje. Agradeço principalmente à minha mãe, por estar ao meu lado em todos os momentos, bons e ruins, e mesmo sem entender todos os meus sonhos, nunca deixou de me apoiar e me aceitar por quem sou. Agradeço também a meus irmãos e minhas cunhadas, que me presentearam com ótimas memórias nos últimos anos. Agradeço também ao meu pai por ter lutado muito pra me dar a estrutura necessária para chegar aonde estou hoje. Aos meus padrinhos, Jorginho e Wilma. Agradeço especialmente às minhas primas: Priscilla, que considero irmã, e Anne, que despertou meu interesse pela ciência.

Agradeço também ao meu parceiro de vida, e hoje meu marido, Luis, que esteve ao meu lado nos últimos anos me dando todo o apoio necessário. Muito obrigado por todo o companheirismo e por ser meu conforto, sem você eu não estaria aqui, e te amo muito. Amo muito também Luna e Nix, minhas companheiras de 4 patas.

Agradeço imensamente aos amigos que fiz, principalmente USJ, Angela, Marcelly, Bibs, Mari, Cat, David, Sarcinelli, Cipri, Lorenzo, Marcela, Pedro, Igor, Rogério, Raphael, e ao meu companheiro de troca de curso Ryan. Não consigo imaginar ter passado pela universidade sem vocês, e agradeço a cada dia por ter amigos maravilhosos. Aos meus amigos de longa data: Letícia, Marcio, Victoria, Paulo, Caio, Maria e Artur. Obrigado por estarem ao meu lado por todo esse tempo.

Agradeço ao meu orientador, Prof. Dr. João Paulo Andrade Almeida, por toda a paciência e dedicação ao longo destes dois anos, durante a iniciação científica e este projeto de graduação. Agradeço também aos membros da banca, Profa. Dra. Renata Guizzardi e Profa. Dra. Monalessa Barcellos, por participarem deste trabalho. Agradeço também aos professores que me moldaram como o cientista da computação que irei me tornar, e me guiaram durante todo o percurso: GC, Vítor, Patrícia, Zambon, Rô, Varejão, Jordana e Roberta.

Por fim, agradeço à instituição Universidade Federal do Espírito Santo por todas as experiências proporcionadas. Dos responsáveis por garantir a estrutura da universidade, aos amigos da CT Junior e da Semana da Engenharia. É imensurável o crescimento profissional e pessoal que obtive neste lugar.

"You got your passion, you got your pride
But don't you know that only fools are satisfied?
Dream on, but don't imagine they'll all come true
When will you realize, Vienna waits for you?
(Billy Joel)

Resumo

Com o objetivo de criar modelos conceituais baseados em ontologias, diversas técnicas de modelagem foram desenvolvidas ao longo dos últimos anos. Uma dessas técnicas foi a OntoUML, criada com base nos diagramas de classe da UML e na ontologia de fundamentação UFO (Unified Foundational Ontology). Após sua proposição, diversos aprimoramentos e casos de uso foram publicados ao longo das últimas duas décadas. Isso resultou em melhorias nos elementos da linguagem, ferramentas ou plugins de edição, validação de modelos e de transformação para outras linguagens (como OWL).

Apesar dos vários avanços, por a OntoUML ser primariamente uma linguagem diagramática/visual, diversos desafios persistem associados a esse tipo de representação. Estes desafios incluem, por exemplo, o esforço investido em tarefas de diagramação e layout, a dificuldade de manutenção de modelos com grande número de elementos, e a inabilidade de aplicação de ferramentas baseadas em texto para manipulação destes modelos. Essas ferramentas poderiam facilitar não apenas a edição de modelos, mas também outras tarefas como o controle de versão e a comparação de versões de um mesmo artefato.

Assim, surge a linguagem Tonto, a partir da junção das palavras "Textual" e "Ontology". Tonto é uma linguagem textual desenvolvida com o objetivo de permitir a criação de ontologias de forma independente de ferramentas diagramáticas e visuais. Por meio da utilização da ferramenta Langium, desenvolveu-se uma gramática para Tonto e uma extensão para esta linguagem no editor VS Code, amplamente utilizado pela comunidade atualmente.

Por meio da Tonto, é possível desenvolver ontologias com uma linguagem textual que possui construtos correspondentes àqueles da OntoUML, além de ser possível realizar em tempo real validações semânticas que garantem a qualidade do modelo. Além disso, é possível serializar os modelos produzidos usando o JSON schema da OntoUML, e assim utilizar serviços do projeto ontouml-server, como a transformação de ontologias em OWL baseadas em gUFO. A extensão projetada permite a validação de modelos com a API ontouml-server, e a importação de modelos em formato JSON para Tonto.

Por fim, com o objetivo de garantir a modularização de projetos Tonto, contribuindo com a reutilização e organização de especificações, foi desenvolvido o TPM (Tonto Package Manager). O projeto de TPM foi inspirado em gerenciadores de pacote de linguagens de programação amplamente populares, como NPM (Node Package Manager) e SPM (Swift Package Manager).

Palavras-chaves: Modelagem Conceitual, OntoUML, Langium, Sintaxe Textual, Ontologias

Abstract

With the objective of creating Ontology-Driven Conceptual Models, several modeling techniques have been developed over the past years. One of these is the OntoUML language, which was created by specializing UML class diagrams with concepts of the Unified Foundational Ontology (UFO). Since its proposal, numerous enhancements and use cases have been published over the last two decades, improving language elements, and enabling model development with tools and plugins, including functionality for model validation and transformation into other languages such as OWL.

In spite of the numerous advancements, several challenges still persist due to visual/diagrammatic nature of the language. These challenges include, for example, the effort invested in layout diagramming tasks, the difficulty of maintaining models with a large number of elements, and the inability to apply text-based tools for manipulating these models. These tools could facilitate various tasks, such as version control and comparison of different versions of the same artifact. Considering the potential of text-based tools, there are numerous opportunities to explore them for conceptual modeling based on ontologies.

Thus, Tonto emerges from the combination of the words “Textual” and “Ontology”. Tonto is a textual language developed with the objective of allowing the creation of UFO-based conceptual models completely independently of diagrammatic and visual tools. By using the Langium tool, a grammar was developed for Tonto, along with a VS Code extension, leveraging thus one of the most popular text-based development IDEs.

By employing Tonto, it is possible to develop ontologies with text-based constructs that correspond to OntoUML constructs. The developed extension provides real-time semantic validations concerning the quality of the model. Additionally, it allows for transformations to the JSON format based on the JSON schema of OntoUML and to OWL implementations based on gUFO. It also enables model validation with the ontouml-server API and the importing of models serialized in the JSON format into Tonto. All functionalities are available in the extension, and they have also been provided in a Command-Line Interface (CLI).

Finally, with the objective of ensuring the modularization of Tonto projects, contributing to ontology reuse and organization, the Tonto Package Manager (TPM) was developed. The design of TPM was inspired in popular software development package managers such as NPM (Node Package Manager) and SPM (Swift Package Manager).

Keywords: Conceptual Modeling, OntoUML, Langium, Textual Syntax, Ontology

List of Figures

List of abbreviations and acronyms

UML	Unified Modeling Language
TPM	Tonto Package Manager
VSCode	Visual Studio Code
DSL	Domain-Specific Language
W3C	World Wide Web Consortium
CLI	Command Line Interface
API	Application Programming Interface
AST	Abstract Syntax Tree

Contents

1 Introduction

Conceptual modeling is a key task in information system design. A conceptual model describes selected aspects of the physical and social world in such a way as to support the creation of computational representations of these selected aspects. Since the conceptual model is often used as a starting point for building a system, the quality of the conceptual model is crucial in the development process. In recent decades, there has been a growing interest in using foundational ontologies in conceptual modeling, with the creation of ontology-driven modeling languages that incorporate the underlying conceptual distinctions of these foundational ontologies. Based on this view, the Unified Foundational Ontology (UFO) (GUIZZARDI, 2005; GUIZZARDI et al., 2022) was developed by combining various theories from linguistics and formal philosophical ontology.

UFO was used as the basis for defining the OntoUML modeling language (GUIZZARDI et al., 2018; GUIZZARDI, 2005). This language was originally developed as an extension of UML class diagrams (OMG, 2017), introducing various stereotypes that correspond to the concepts defined in UFO. Over the years, sophisticated tooling has been developed for OntoUML at the Conceptual Modeling & Ontologies Research Group (NEMO) of the Federal University of Espírito Santo (UFES) and other research groups in the Free University of Bolzano (Italy) and the University of Twente (the Netherlands), including functionalities for: (i) editing and syntactic verification of models (MOREIRA et al., 2016; FONSECA et al., 2021), (ii) model simulation (BRAGA et al., 2010; BENEVIDES et al., 2010), (iii) automatic generation of database schemas (GUIDONI; ALMEIDA; GUIZZARDI, 2020), (iv) detection of anti-patterns (SALES; GUIZZARDI, 2015), among others.

Despite many advances, the OntoUML language is still an extension of UML and primarily a diagrammatic and visual language. This provides significant benefits for communication among modelers, as well as problem-solving. Even with the advantages of a visual language, important challenges are associated with this type of representation. These challenges include the effort invested in diagram layout tasks, the difficulty of dealing with large models, and the inability to apply text-based tools to manipulate the models. Such tools could facilitate various tasks, such as version control, comparison between versions of the same artifact, merging different versions, and more. However, they are currently not applicable in their current form to diagrammatic languages. Considering the potential of text-based tools, there is a universe of opportunities to explore these tools for ontology-based conceptual modeling.

1.1 Context and Motivation

Artificial Intelligence, Semantic Web, Software Engineering, and Information Architecture are just a few of the domains that have greatly benefited from the indispensable tool of conceptual modeling in Computer Science for several years. This technique allows for representing knowledge about the world in a structured and organized way. Thus, the existence of theoretical foundations and tools for the development of these models is essential, ensuring consistency, ease of use, and support for a number of tasks (such as model analysis, verification, validation, etc.) An example of such theoretical foundations is the Unified Foundational Ontology (UFO), which is leveraged in practical settings with the OntoUML language, for which, since its inception, a series of tools have being built.

Visual languages such as OntoUML have a number of benefits, and allow diagrams to be used for visualization of the relations between elements and for efficient communication. However, since the model is strongly linked to its visual representation in the form of a diagram, there are various moments when attention to the layout of diagrams are necessary, e.g., to manually arrange elements and relations, to produce selected views. Layout efforts often need to be revisited when there are changes to the model. There is also the difficulty of dealing with large-scale models, especially when we have a large number of elements, attributes, and relationships. For example, a single element can have so many relationships that it may become difficult to layout the diagram in a manner conducive to the correct interpretation of the diagram.

In addition, there is the impossibility of applying text-based tools to manipulate the models. These tools can facilitate a series of tasks such as version control, comparison between versions of the same artifact, merging different versions, etc. However, they are not easily applicable to diagrammatic languages. The lack of appropriate versioning often hinders the collaboration of multiple developers working simultaneously on a project, generating conflicts when trying to merge the changes made by both parties. Considering the potential of text-based tools, there is a universe of opportunities to explore these tools for conceptual modeling based on ontologies. Therefore, there is great relevance in building a robust textual syntax that is easy to use and has a wide range of functionalities that bring advantages to the models developed in OntoUML.

Finally, one important tool in modern programming language is a way to facilitate code reuse and distribution with *package managers*. However, the existing modeling tools currently lack support for such a mechanism, resulting in the need for different projects to repeatedly redefine the same concepts and elements, leading to increased complexity.

From the point of view of a final graduation project in Computer Science, this work allows integration of theory and practice of concepts and techniques acquired during the bachelor course, favoring the consolidation of content of the disciplines involved in the training

cycle, such as software engineering, programming languages and compilers, among others.

1.2 Objectives

The general objective of this project is to create a textual conceptual modeling language for UFO-based models with constructs corresponding to those of OntoUML. The language should be supported by a rich language editor in the form of a Visual Studio Code Extension. The extension should enhance the development experience and should provide compatibility with existing elements in OntoUML. It should further permit the modularization of conceptual modeling projects into different code repositories.

The following specific objectives were defined:

- Objective 1: Propose a textual syntax and create the corresponding grammar for the language.
- Objective 2: Create a Visual Studio Code extension for the language, which should support features typical of modern source code editors, including syntax verification, syntax highlight, auto complete, etc.
- Objective 3: Create a package manager to support the modularization of conceptual modeling projects, including the management of dependencies to other reusable models.

1.3 Approach

In this section, we will present the methodology applied in this work. To carry out the objectives of this work, it was initially necessary to study the theories involved in the project, including reading various articles, dissertations, and the doctoral thesis "Ontological Foundations for Structural Conceptual Models" (GUIZZARDI, 2005), which was fundamental for understanding the problem domain. Other resources were studied, especially those that extended the theory and capabilities of OntoUML, for example, the latest taxonomy of UFO (GUIZZARDI et al., 2022). Other materials, for example, online content, specifications, documentation, and videos were also studied. Then, topics such as conceptual modeling, the Semantic Web, and the fundamental theory of ontologies, which forms the basis for OntoUML, were reviewed in the literature.

With a structured theoretical foundation, an in-depth study of the tool used for creating the Tonto language and extension was conducted. The Langium library[1] was studied to create basic DSL (Domain-Specific Language) examples, aiming to gain familiarity with it. One of the examples created was a simplified version of UML itself, serving as a pilot to initiate the

[1] <https://langium.org/>

formulation of a textual representation for classes, packages, and other artifacts also used in OntoUML. This experience was essential for planning the textual syntax of Tonto, considering the various artifacts present in the tool. To complement the understanding of the tool, a study was conducted on the API for creating extensions for the Visual Studio Code code editor, as well as its Language Server Protocol API, which supports programming languages and development tools used in the process.

As a source of inspiration, other textual languages that are part of the modeling and Semantic Web ecosystems were studied, such as RDF Schema, Turtle (W3C, 2014), OWL (W3C, 2012), Alloy (JACKSON, 2016), ML2 (FONSECA, 2017), among others.

With all this gathered information, an initial version of the Tonto grammar was planned, and a series of meetings between the student, supervisor, and other researchers helped refine the initial ideas obtained. Several concepts from current programming languages were replicated, aiming to create familiarity for developers with the syntax. The language was named "Tonto", and the development of the extension began. Throughout the process, the recently launched Langium tool received important updates, adding new functionalities and keeping up with releases. After creating the basic syntax, validators were developed to analyze the model created by the user in more depth, particularly in terms of the semantics of OntoUML rules.

Based on existing models, the strategy involved conducting different iterations of development, ensuring that new language elements were added and tested. Finally, in addition to the grammar elements, functionalities were developed in the Visual Studio Code extension to enable the serialization of OntoUML models into JSON files, following the OntoUML Schema (FONSECA et al., 2021). Furthermore, by utilizing the OntoUML server API (FONSECA et al., 2021), the possibility of validating and transforming the model into gUFO models (ALMEIDA et al., 2019) was added. Finally, a package manager was created to enable the capability of a Tonto project to have a dependency on other projects, reusing code without needing to copy it manually from another project.

Throughout the project development, unit tests were implemented to ensure the functionality of already implemented features. Finally, the strategy of publishing the CLI on the NPM marketplace[2] and the extension on the VS Code marketplace was adopted for validation purposes.

1.4 Structure

In addition to this introduction, this work consists of four other chapters.

- Chapter 2 presents aspects related to the theoretical content relevant to the work;

[2] <https://www.npmjs.com/>

- Chapter 3 presents the main contribution of this work, explaining the various elements of the language Tonto;
- Chapter 4 presents an example of a model using Tonto and compares it to the diagrammatic (OntoUML-based) version of it;
- Chapter 5 presents the concluding remarks of this work.

2 Theoretical Reference and Technologies Used

The following sections present the theoretical foundation that supported the development of this project. First, we introduce Ontology-Based Conceptual Modeling and its role in Computer Science and Information Systems. Subsequently, we provide details about the OntoUML language and its specifications. Finally, we present related works covering examples of textual languages for conceptual modeling or software engineering.

2.1 Ontology-Based Conceptual Modeling and ontologies

Ontology is a branch of philosophy that studies concepts such as existence, its nature, and its relationships. The Cambridge Dictionary defines the term "Ontology" as "the part of philosophy that studies what it means to exist" (CAMBRIDGE, 2023). As addressed by Guizzardi (2005), we can define that ontology is broader than other scientific disciplines:

> As opposed to the several specific scientific disciplines (e.g., physics, chemistry, biology), which deal only with entities that fall within their respective domain, ontology deals with transcategorical relations, including those relations holding between entities belonging to distinct domains of science, and also by entities recognized by common sense. Ontology aims to develop theories about, for example, persistence and change, identity, classification and instantiation, causality, among others.

That means that when we are trying to define what exists, we need to make relations between elements that normally would be ignored in a specific context.

In Computer Science and Information Systems many areas have interest in ontologies. For example, "the need to create principled representations of domain knowledge in the knowledge sharing and reuse community in AI" (GUIZZARDI, 2005), or in data and information modeling and processing. Also, we have in the software engineering field the importance of *domain engineering*, motivated by the need of reducing costs in software maintenance and the need to reinforce software reuse in a higher level of abstraction than merely programming code. For a database to be more valuable in its domain-specific applications, it is crucial to have a precise conceptualization of the entities that domain experts perceive as significant. The problem was that this field also lacked concrete and consistent formal bases for making modeling decisions (GUIZZARDI, 2005).

In addition to that, we have the application of ontology in the Semantic Web, as a formal artifact. The idea of the Semantic Web is to make the next step on the Web being

more than *machine-readable*, but *machine-understandable*. That means that computers should be able not only to hold information in a structured way, but they should also be able to manipulate the information more deeply. This is done by annotating with meta-data written in a formal knowledge representation language, and can take advantage of advancements in the knowledge representation community. Here, ontologies are key artifacts for integrating human comprehension of symbols with the machine's ability to process them effectively.

Based on this, the W3C defined on top of the XML layer of a document the *Resource Description Framework (RDF)* which is a foundational representation framework in the Semantic Web stack. This stack includes the Web Ontology Language (OWL) (W3C, 2012) which was designed to represent rich and complex knowledge about things, groups of things, and relations between things (W3C, 2012). It provides a formal and expressive framework for defining classes, properties, individuals, and relationships between them. Also, OWL allows the specification of logical constraints and inference rules, enabling automated reasoning and semantic processing of knowledge.

OWL (and any RDF content) can be serialized using the Terse RDF Triple Language (Turtle) (W3C, 2014) textual syntax for RDF. It allows an RDF graph to be completely written in a compact and natural text form, with abbreviations for common usage patterns and datatypes (W3C, 2014). It is a human-readable and compact representation of RDF triples, where triples are written in a subject-predicate-object format.

2.2 UFO

While we can have examples of ontologies in a specific domain, an ontology can also be a *foundational* one, defining aspects that are independent of a domain. Guizzardi (2005) proposes in his work a foundational ontology called UFO (Unified Foundational Ontology) through the composition of several theories from areas such as linguistics and formalizations of ontologies of philosophy. This ontology is divided into three parts: UFO-A, UFO-B, and UFO-C.

For the scope of this work, we will talk mostly about UFO-A, which is an ontology of endurants, and UFO-B, which is an ontology of perdurants. "Endurants are individuals that exist in time with all their parts. They have essential and accidental properties and, hence, they can qualitatively change while maintaining their numerical identify (i.e., while remaining the same individual)" (GUIZZARDI et al., 2022). Billie Eilish, the Moon, and John's weight are all examples of endurants. On the other hand, we have elements that are Perdurants in contrast to Endurants (GUIZZARDI et al., 2013). "Perdurants are individuals that unfold in time accumulating temporal parts. An endurant can change while maintaining its identity, a perdurant cannot" (GUIZZARDI et al., 2022). An example of perdurant is the event of composing a new song, made by an artist. Figure 1 shows the taxonomy of UFO as presented in (GUIZZARDI et al., 2022).

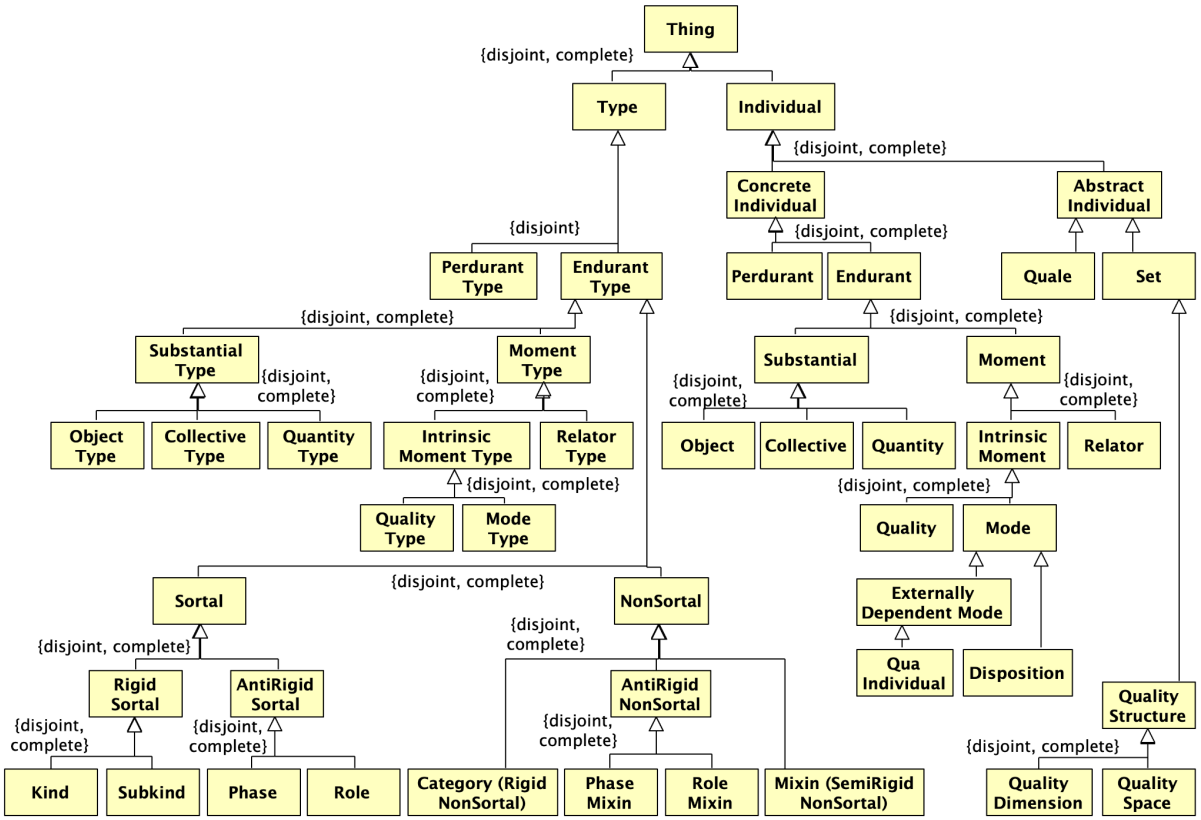

Figure 1 – Diagram describing UFO taxonomy. Source: Guizzardi et al. (2022)

2.3 OntoUML

Based on UFO, a class diagram profile was created for UML, introducing various stereotypes for classes that correspond to the concepts defined in UFO. Over time, it was dubbed 'OntoUML'. Its focus is on domain models, which consequently places more emphasis on endurants (objects, present in UFO-A) than perdurants (events, processes, present in UFO-B), as the foundation of the ontology. We can understand more about each category on the work proposed by Guizzardi (2005) and further improvements of the theory (ALMEIDA; FALBO; GUIZZARDI, 2019; GUIZZARDI et al., 2013; GUIZZARDI et al., 2018). Because OntoUML is a work that had many contributions in the past almost two decades, some elements changed from the first specification until this present taxonomy.

In OntoUML, selected classes in UFO and the relations between them are represented by stereotypes of classes or associations in UML, with syntactic formal constraints that are semantically motivated. "This combination of stereotypes and constraints enforces conformance, making every valid OntoUML model compliant to UFO" (ALMEIDA; FALBO; GUIZZARDI, 2019).

UFO divides all elements into some ontological categories, the first division being between types and individuals. A type for example would be the kind *Computer Operating System*, while an individual would be the available operating systems that we have, like Linux,

Windows and MacOS. The relation between an individual and a type is called *instantiation*, meaning that while types determine the characteristics that something needs to have in order to be considered of that type, an individual is the thing that exhibits these characteristics. OntoUML, being a profile of UML class diagrams, only supports the definition of types, meaning that it does not address the specification of individuals.

Types are further categorized into endurant and perdurant types. Endurant types are classified into two orthogonal hierarchies: they are partitioned into substantial types or moment types in one of these hierarchies, and partitioned into sortals and non-sortals in the other. Substantials are independent entities that exist without the need of another, while moments are endurants that existentially depend on other entities (GUIZZARDI et al., 2022). "A sortal is either a kind or a specialization of a kind, and those who are not a kind need to specialize exactly one kind" (GUIZZARDI et al., 2022). A non-sortal is a type that represents common properties of individuals of multiple kinds.

Note that there are in fact two orthogonal *Endurant Type* taxonomies in UFO: (i) one whose types reflect UFO's taxonomy of individuals such as *Substantial Type*, *Moment Type*, *Object Type*, *Relator Type*, etc.; and (ii) one structured in terms of the *sortality*, *rigidity* and *external dependence* of types, ultimately leading to the notions of *Kind*, *Subkind*, *Phase*, *Role*, *Category*, *Phase Mixin*, *Role Mixin* and *Mixin*. The existence of two orthogonal hierarchies means that combinations are possible: a domain type such as *Person* can instantiate *Object Type* and *Kind* simultaneously. This means that the distinctions that were applied originally to types of substantials (GUIZZARDI, 2005) are also applicable for moment types (GUIZZARDI et al., 2022).

2.3.1 Sortals

The following sections focus on presenting every stereotype of OntoUML, starting with those representing sortals. A fundamental sort of endurant type is *Kind*, a type which provides uniform principles of individuation, identity, and persistence to its instances. For example, the types person, dog, computer, car, organization and marriage are typically considered to be kinds. Kinds apply to instantiating individuals in all possible situations in which these individuals exist (GUIZZARDI et al., 2022). In OntoUML, the stereotype ≪*kind*≫ is a shortcut for *Object Kind*, i.e., an *Object Type* that is also a *Kind*. Because of this, instances of classes stereotyped ≪*kind*≫ are instances of *Object* (also termed 'functional complex' in UFO). Since the notion of ultimate sortals (kinds) is also applicable to *Collective Types*, *Quantity Types*, *Quality Types*, *Mode Types* and *Relator Types*, specific stereotypes are introduced: a class stereotyped ≪*kind*≫ is an *Object Kind*, a class stereotyped ≪*collective*≫ is a *Collective Kind*, a class stereotyped ≪*relator*≫ is a *Relator Kind*, a class stereotyped ≪*mode*≫ is a *Mode Kind*, a class stereotyped ≪*quality*≫ is a *Quality Kind*, and a class stereotyped ≪*quantity*≫ is a *Quantity Kind* (GUIZZARDI et al., 2018).

The meaning of each of these stereotypes (representing kinds of endurants) is as follows:

- ***≪collective≫***: An instance of a class stereotyped *≪collective≫* is a collective entity whose parts (members of the collective) fulfill identical roles in relation to the whole, for example, a deck of cards or a forest as a collective of trees (GUIZZARDI et al., 2022).
- ***≪quantity≫***: An instance of a class stereotyped *≪quantity≫* is a portion of homeomerous amount of matter. For example, a portion of water, soda or sand.
- ***≪quality≫***: An instance of a class stereotyped *≪quality≫* is a particularized property that can be understood as a value in a conceptual space, for example, the weight or height of a person which can be measured in centimeters, or the color of an eye that can be represented in an RGB tuple.
- ***≪mode≫***: An instance of a class stereotyped *≪mode≫* is a particularized property that is not conceived as a value in a conceptual space. For example, the ability of speaking a language that a person can have, or a disease that is affecting a dog.
- ***≪relator≫***: An instance of a class stereotyped *≪relator≫* is a truth-makers of a material relation, an entity that needs to exist for two or more related individuals to be connected through a material relation. For example, a handshake depends on two individuals of the kind element *Person*. Examples or relators include social objects such as *Marriage*, or a purchase order from an online store.

The additional sortal stereotypes *≪subkind≫*, *≪phase≫* and *≪role≫* represent their counterparts in UFO. They must specialize a unique *kind* from which they inherit a principle of identity for their instances. Whether their instances are objects, collectives, quantities, qualities, modes or relators is already settled by specialized class (which will be stereotyped *≪kind≫*, *≪collective≫*, *≪quantity≫*, *≪quality≫*, *≪mode≫* or *≪relator≫*). These additional sortal stereotypes have the following semantics:

- ***≪subkind≫***: Subkinds are rigid specializations of a kind. For example, we can have *Man* as a subkind of *Person*.
- ***≪phase≫***: Phases are "sortals whose contingent classification conditions are intrinsic" (GUIZZARDI et al., 2022). They represent changes in intrinsic properties of instances of a kind, for example, in the case of the age of instances of the *kind* person, we can have phases such as *Child*, *Teenager* and *Adult*.
- ***≪role≫***: Roles are "sortals whose contingent classification conditions are relational" (GUIZZARDI et al., 2022). They are anti-rigid specializations of kinds, for example, the role *student* of the kind *person*.

2.3.2 Non-Sortals

As opposed to sortals, "non-sortals are types that represent common properties of individuals of multiple Kinds." (GUIZZARDI et al., 2022). The non-sortals are *categories*, *phase-mixins*, *role mixins* and *mixins*.

- ***≪category≫***: Categories are "rigid types that define essential properties for their instances, e.g., the category 'physical object' describing the properties of having a mass and a spatial extension, common to things of the kinds car, person, bridge, cow, etc.;"(GUIZZARDI et al., 2022). Or, for example, we can have the category *Furniture*, which describes properties of things that are usually used in a house by humans for many purposes.
- ***≪phaseMixin≫***: Phase mixins are "anti-rigid types that define contingent properties for their instances. Their instantiation is characterized by intrinsic contingent conditions. For example, the phase mixin 'living animal' may apply to instances of the kinds person, dog, and horse" (GUIZZARDI et al., 2022).
- ***≪roleMixin≫***: Role mixins are "anti-rigid types that define contingent properties for their instances" (GUIZZARDI et al., 2022), aggregating instances with different identity principles. For instance, the role mixin *customer* can be specialized by the role *personal customer* of the kind *person*, or by the role *corporate customer* of the kind *company*.
- ***≪mixin≫***: Mixins are "semi-rigid types that define properties that are essential to some of their instances but accidental to some other instances (e.g., being a 'music artist' is essential to bands but accidental to people)." (GUIZZARDI et al., 2022).

Figure 2 illustrates an OntoUML example model of a University, showcasing some details of a Person with attributes, various life phases, and the roles encompassing a Person's involvement as a University Student and a University Professor. The model encapsulates essential elements found within a university setting, including classrooms, staff, and departments. Additionally, it incorporates two relators that establish the contractual relationship between students and professors within the university.

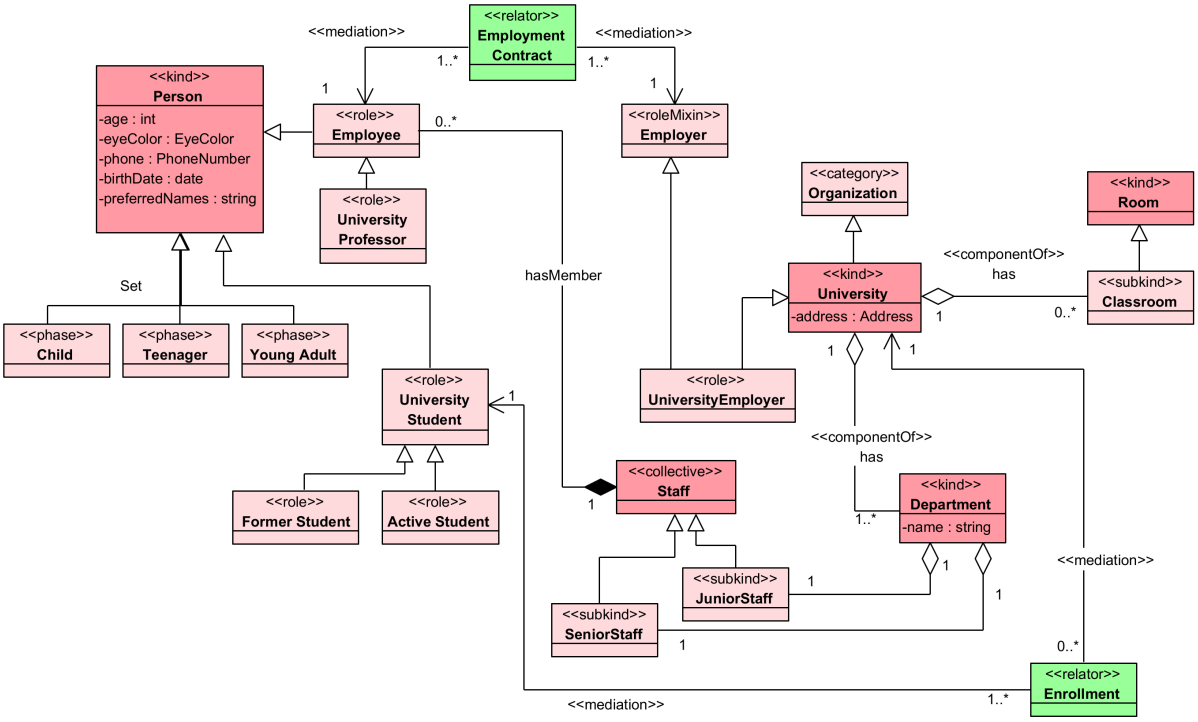

Figure 2 – OntoUML model example representing a University with departments and classrooms, professors and students.

2.3.3 Ontological Natures

As part of OntoUML's evolution, OntoUML 2.0 was proposed changing some existing elements of UFO and introducing a new concept called *ontological natures* (GUIZZARDI et al., 2018). When we look at an ultimate sortal stereotype, i.e. ≪*kind*≫, ≪*collective*≫, ≪*quantity*≫, ≪*quality*≫, ≪*mode*≫, and ≪*relator*≫ stereotypes, based on the stereotype kind, one can already determine the nature of the instances of this element. For instance, a class with the stereotype ≪*relator*≫ would have the nature of relators. Also, other sortals, i.e. ≪*subkind*≫, ≪*role*≫, ≪*historicalRole*≫ and ≪*phase*≫ need to specialize a unique kind to provide their nature. The same is not valid for non-sortals, i.e. ≪*category*≫, ≪*mixin*≫, ≪*phaseMixin*≫, ≪*roleMixin*≫, and ≪*historicalRoleMixin*≫ stereotypes. At first, an abstract category like *Social Entity* would be able to be specialized by kinds (functional complex) or relator kinds without the need to specify anything. In order to improve the specification of non-sortals, the functionality to define the ontological nature of these elements was added, defining that instances of that category would have to follow this specific nature. In that way, for example, a category could have its instances restricted only to relator kinds by specifying its nature to relators. This was added to OntoUML using the *restrictedTo* UML 'tagged value' that can be assigned to a non-sortal to establish the possible natures of their instances (GUIZZARDI et al., 2022).

2.3.4 Beyond Endurant Types

After the development of the core concepts of UFO-A in OntoUML, UFO-B elements were presented to include perdurants, expanding the theory. They are presented in Figure 1 as opposed to *Endurant* types and individuals. Two stereotypes were included, ≪*events*≫ and ≪*situations*≫. Events are individuals composed of temporal parts and happen in time, meaning that they are a set of temporal parts accumulated (GUIZZARDI et al., 2013). Some examples are a party, a basketball game, or a music festival. They can be composed of other events, for example, the fall of the Roman Empire is composed of many events like Rome being attacked by barbarians and weather problems that disrupted harvests, deepening the crisis. An event composed of other events is a Complex Event, while an event without any smaller part is an Atomic event. They are ontologically dependent entities in the sense that they existentially depend on objects in order to exist.

Another important detail is that "...by introducing events in the model, our universe of discourse contains not only the entities that exist in a given circumstance but also all entities that have existed in that history of our universe of discourse up to that point" (ALMEIDA; FALBO; GUIZZARDI, 2019). That means that historical semantics were included in models because we are only taking into account events that occurred in the past. For example, in Figure 3 we can see two different ways to represent a person being a soccer player. One of them considers the event of the soccer match, therefore, the role played by this person is a historical role, that happened at the same temporal part of that match. The other option is representing the contract established between a player and a soccer club, meaning that this person is a soccer player because of this ongoing relationship instead of only one event. The participation stereotype of this relation represents the player participating in the soccer match. Also, another important detail is that events are immutable, in contrast to relators, which can change their properties while remaining the same (ALMEIDA; FALBO; GUIZZARDI, 2019).

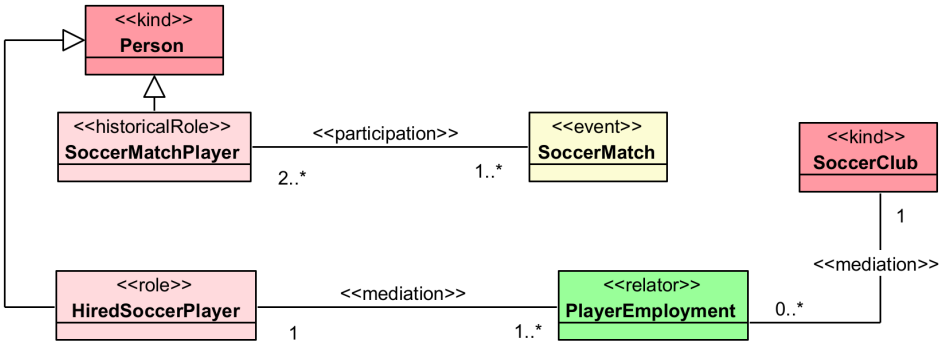

Figure 3 – Diagram describing a soccer match with roles of a Person. Source: Almeida, Falbo e Guizzardi (2019)

On the other hand, situations are inherently tied to specific points in time. Therefore, two qualitatively identical situations occurring at different time points are regarded as distinct in numerical terms (GUIZZARDI et al., 2013). For example, the situation of "Mary having

long hair today" and the situation of "Mary having long hair some moment in the past" are considered separate instances.

2.3.5 High order types as Endurants

Until recently, OntoUML reflected UFO's foundation which defines that domain entities are divided in types and instances. However, not all entities conform to that definition, and accumulate at the same time instance-like and type-like characteristics. One example is from the context of software development, where we could define types of tasks that need to be executed during software development, and also classify the types of types of tasks. Therefore, in order to conceptualize the software development domain, we require representing entities in different classification levels, in this case, tasks, types of tasks, and types of types of tasks. This differentiates from the classic two-level division between classes and instances, and allows classes that are instances of other classes. This is presented in the Multi-level Theory (MLT) (ALMEIDA; FONSECA; CARVALHO, 2017). Based on this theory, high-order types were incorporated in UFO and OntoUML.

To incorporate high-order types, the notion of instantiation (iof) provided by MLT is integrated in UFO, "where iof is a primitive relation that holds between an instance *e* and a type *t* in a world *w* where *t* classifies *e*" (FONSECA et al., 2022). In order to enable the declaration of high-order kinds, the stereotype ≪*type*≫ was introduced in OntoUML. The ≪*instantiation*≫ is "used to provide specialized semantics to an association between a high-order type and a base type" (FONSECA et al., 2022). Also, new tagged values were included. *restrictedTo* now includes *type*, in order to account for the type's ontological nature. *isPowertype* is used to determine if "it is a Cardelli powertype of the base type" or if it is an Odell powertype, i.e., "a categorizer of the base type" (FONSECA et al., 2022). Lastly, an *order* tagged value was included to allow defining the order (ALMEIDA; FONSECA; CARVALHO, 2017) of the declared type. Figure 4 shows an example of type definition in a model about Ships. The type *Ship Type* is a `powertype` of order 2, and is associated with the kind *Ship* with an instantiation relation.

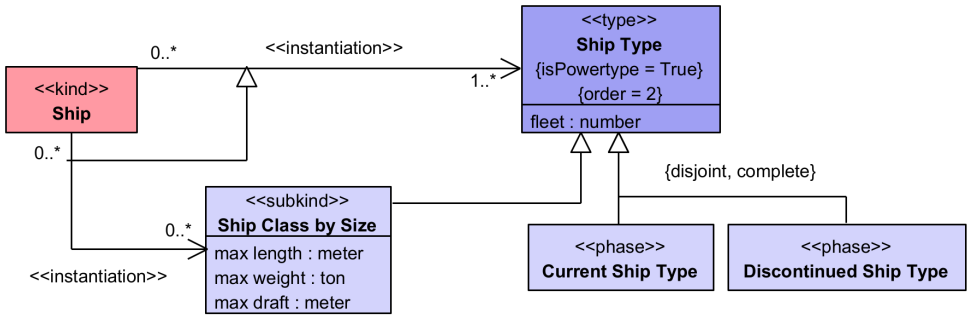

Figure 4 – Diagram describing a Ship Type with. Source: Fonseca et al. (2022)

2.3.6 OntoUML as a Service

With this foundation, a group of tools was developed to support the development of UFO-based models through OntoUML using microservices. Called *OntoUML as a Service infrastructure* (OaaS) (FONSECA et al., 2021), this infrastructure includes:

- **ontouml-js**: A library developed in TypeScript that allows the creation and serialization of OntoUML models using JavaScript methods.
- **ontouml-schema**: A JSON schema that defines a set of rules to how OntoUML models are serialized in this format.
- **ontouml-server**: An HTTP server created to provide validation and transformation features to OntoUML models in JSON format. The validation service takes in a serialized JSON model, applies syntactic validations to it, and if there are any errors, it returns an array of those errors, otherwise it return that the model is valid. The transformation service transforms a serialized JSON model into gUFO-based OWL ontologies.
- **ontouml-vp-plugin**: A plugin for the Visual Paradigm tool created originally for UML. The plugin offers a range of helpful features that allow the modeler to visualize OntoUML constructs with ease and modify them. Additionally, the plugin is designed to connect with ontouml-server, providing users with validation and transformation capabilities.

2.4 Technologies Used

In this section, we will present the main technologies that were used during the development of this work, with a brief explanation of each.

2.4.1 Langium

Langium[1] is an open-source tool developed by TypeFox[2] with first-class support for the Language Server Protocol (LSP)[3]. It utilizes TypeScript and runs on the Node.js runtime. With *Langium*, it is possible to develop textual Domain Specific Languages (DSLs) that can be used in IDEs such as VS Code, Eclipse Theia, and also in Web applications.

The tool generates a typed Abstract Syntax Tree (AST) along with default implementations for many LSP features. Thus, it automatically generates extensions for VS Code based on the created DSL, enabling features such as auto-complete, syntax highlighting, validators, code generators, and the development of a command-line interface (CLI).

[1] <https://langium.org/>
[2] <https://www.typefox.io/>
[3] <https://microsoft.github.io/language-server-protocol/>

Being created by the same team that produced a widely popular similar tool (for the Eclipse world), Xtext[4], *Langium* is robust and aims to maintain the concepts that made Xtext successful while improving them for another platform in a simple and clear manner.

2.4.2 Visual Studio Code

Visual Studio Code[5], also known as VS Code, is a source code editor available for Windows, Linux, and macOS, developed by Microsoft. It includes support for debugging, built-in git version control, syntax highlighting, intelligent code completion, snippets, and code refactoring. It is highly customizable, allowing users to modify themes, keyboard shortcuts, and various preferences, which can be synchronized across devices.

Although the base version of VS Code may not have support for certain languages and functionalities, it can be extended by installing extensions. With the help of extensions, it becomes a powerful programming tool and can be considered an Integrated Development Environment (IDE). The extensions assist in syntax highlighting, code formatting, adding new snippets, refactoring, code execution and debugging, version control, error visualization, and many other functionalities.

In this project, VS Code is essential as it executes the extension that enables the utilization of all Tonto features. The IDE also provides extensive support for extension development, along with rich and detailed online documentation.

2.4.3 Visual Studio Code Extensions API

Visual Studio Code was created with extensibility in mind, allowing developers the freedom to add their own functionalities. Virtually any part of the code editor can be customized using its extension API, including elements of the User Interface (UI) and icons. Some built-in editor elements are developed using this API, contributing to its improvement by the project's own contributing team.

In this project, Tonto extensively utilizes this API to enhance its developer interface and facilitate the visualization of the developed models. Part of this usage is done directly through the Langium library, while another part is done natively. Thus, the utilization of this technology is essential to ensure the quality of the project.

2.4.4 Development and Execution Technologies

TypeScript[6] is a strongly typed programming language that is built on top of another programming language, JavaScript. It adds additional syntax to JavaScript to support error

[4] <https://www.eclipse.org/Xtext/>
[5] <https://code.visualstudio.com/>
[6] <https://www.typescriptlang.org/>

identification at compile time instead of runtime. TypeScript transpiles all its code to JavaScript, which allows it to run in any environment that supports JavaScript. In the case of this project, it is executed in the Node.js environment.

Node.js[7] is an open-source, cross-platform JavaScript runtime environment. It was developed to execute JavaScript code beyond the browser, targeting current operating systems. Node.js is designed for building scalable network applications and is an asynchronous, event-driven runtime, allowing multiple connections to be executed concurrently. In this project, Node.js serves as the execution environment for Tonto as it runs within the Language Server of VS Code. Therefore, it is possible to leverage all the available APIs of Node.js if needed.

Vitest[8] is a framework that enables the development of unit tests for Node.js, utilizing TypeScript and compatible with the Jest framework. It focuses on performance, ensuring fast test execution.

2.5 Related Work

Numerous works present conceptual modeling techniques using diagrammatic/visual or textual tools, with some being based on ontologies. Here, we also present works that, in some way, develop textual syntax for modeling languages, even though their focus is not on conceptual modeling. The first work to be highlighted is ML2, which, as presented by Fonseca (2017), is a language focused on multi-level modeling. It allows for an approach where classes can be instances of other classes, instead of the conventional approach where we have a separation between classes and individuals that instantiate these classes. The syntax of ML2 is defined using the Xtext tool[9] (a precursor of Langium in the Eclipse environment), enabling the definition of elements such as Classes, Datatypes, Generalization sets, Attributes, among others. Overall, ML2 demonstrates the potential of textual modeling languages in the context of multi-level conceptual modeling.

PlantUML[10] is a textual language that allows a modeler to write textual specifications of UML diagrams, including class diagrams, use case diagrams, state diagrams, and many others. It has a simple syntax and is able to create visual elements based on textual input. It focuses on UML as a general purpose language.

Ontological Modeling Language (OML)[11] is a language developed by the Jet Propulsion Laboratory. It allows users to create ontology-based models using a user-friendly syntax, and is based on UML and inspired by OWL 2. Moreover, it is implemented using the Eclipse Modeling Framework (EMF) and has a plugin for the Eclipse IDE and for VSCode IDE, allowing syntax

[7] <https://nodejs.org/en/>
[8] <https://vitest.dev/>
[9] <https://www.eclipse.org/Xtext/>
[10] <https://plantuml.com/>
[11] <http://www.opencaesar.io/oml/>

highlighting, live validation, and content assist. Also, it allows checking for logical consistency of created models.

None of these languages target UFO-based models specifically. Nevertheless, they have been used loosely as sources of inspiration for the syntax of Tonto.

3 Tonto: A textual syntax for modeling

In this chapter, we present the created language Tonto, along with its tool support. First, we discuss the requirements that were defined for the language and associated tools. Then, we present the Tonto grammar[1] in detail and show how each definition relates to corresponding UFO/OntoUML elements. After that, we present the Visual Studio Code extension that was developed to allow the creation of models in Tonto. Lastly, we introduce the Tonto Package Manager and discuss how it facilitates the modularization of Tonto projects.

3.1 Requirements

The following requirements were defined for the creation of Tonto language:

- Use a textual syntax that is familiar to users of mainstream object-oriented programming languages (such as Java, Typescript and Swift) and UML.
- Cover the maximum amount of OntoUML constructs, including classes, stereotypes of classes, relations, stereotypes of relations, cardinalities, meta-properties, generalization sets, ontological natures, high-order types, specializations, attributes, datatypes and enumerations.
- Support the production of syntactically correct models.

Also, to increase usability, some requirements for the VS Code extension and Tonto tooling were defined:

- Assist the modeler in understanding model errors with (real-time) syntax verification based on OntoUML/UFO rules.
- Assist the modeler in producing correct Tonto models with features such as code snippets and auto-complete.
- Enable the importing of models defined in OntoUML (and serialized in JSON) to Tonto.
- Enable the exporting of Tonto models to OntoUML tools.
- Enable transparent access to the ontouml-server functionalities, leveraging these functionalities available for OntoUML models also for Tonto models (including generation of OWL implementations).
- Support the modularization of Tonto-based conceptual modeling projects.

[1] <https://github.com/matheuslenke/Tonto>

3.2 Tonto Grammar

We focus here on every element that can be declared in a Tonto specification (a `.tonto` file). The full Tonto grammar is defined in the Langium EBNF syntax and can be found in Appendix A.1.

The first declaration in a Tonto specification is equivalent to the definition of a *package* in UML/OntoUML, using the keyword `package`. It establishes that all the declarations within a file belong to the named package at the top of a file. After the package name, we have a set of declaration statements. Every statement can be either: (i) a class declaration or (ii) an auxiliary declaration.

Every class declaration (i) follows the idea of declarations in popular programming languages like Java, where we have a keyword for the type of the element followed by its name or identifier. A number of UFO types mentioned in Chapter 2 and corresponding to OntoUML stereotypes lead to keywords of a class declaration in Tonto: `kind`, `collective`, `quantity`, `quality`, `mode`, `intrinsicMode`, `extrinsicMode`, `relator`, `type`, `powertype`, `subkind`, `phase`, `role`, `historicalRole`, `event`, `situation`, `category`, `mixin`, `phaseMixin`, `roleMixin`, `historicalRoleMixin`. There is also the possibility of using the neutral `class` keyword when the modeler has not yet specified the ontological category applicable to a class.

Auxiliary declarations (ii) include datatypes, enumerations, generalization sets and associations (when defined outside the body of class declarations).

Figure 2 in Chapter 2 contains a diagram with an example of an OntoUML model. Throughout this chapter, we will explain Tonto grammar using some elements of this model.

3.2.1 Class and Datatype Declarations

Figure 5 shows a piece of OntoUML model for a domain involving persons, some of their phases and roles. The class *Person* is stereotyped ≪*kind*≫ and includes some attributes that we will address later in this chapter. The classes *Child*, *Teenager*, and *Young Adult* are (incomplete) phases of a person's life, and because of that, they have the stereotype ≪*phase*≫. Also, we have the role *University Student* that a person can play in the scope of a relationship with a university, and hence the class is marked with the stereotype ≪*role*≫. Further, there are roles for *University Student*s that classify them into *Former Student* or an *Active Student*, depending on whether this person's studies are finished.

We can now have a look at the Tonto version of this model in Listing 3.1. First, we have the package definition related to the context of our model. Using the `kind`, `phase`, and `role` keywords, we declare each element of the model. To represent the generalization relation, Tonto uses the `specializes` keyword after the name of the class, and each class can specialize as many elements as it needs.

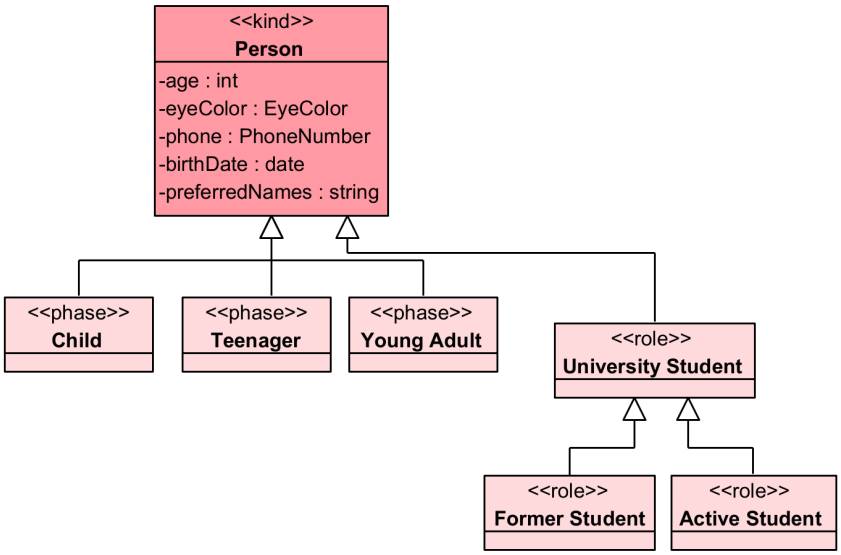

Figure 5 – Example model representing a person's life stages and a possible role as a student, built in the OntoUML plugin of Visual Paradigm tool.

Listing 3.1 – Tonto model based on Figure 5.

```
1  package PersonPhases
2
3  kind Person
4
5  phase Child specializes Person
6  phase Teenager specializes Person
7  phase YoungAdult specializes Person
8
9  role UniversityStudent specializes Person
10 role FormerStudent specializes UniversityStudent
11 role ActiveStudent specializes UniversityStudent
```

One thing missing on Listing 3.1 with respect to Figure 5 is the definition of *Person* attributes. We will focus now only on the kind *Person* with its attributes, and we can notice that we have attributes with 4 different datatypes. By default, Tonto defines the following datatypes: *number, string, boolean, date, time* and *datetime*. We don't have pre-defined datatypes for *int*, *EyeColor*, and *PhoneNumber*, so we need to define them in order to be able to use them. Due to Tonto's multi-file system, every `.tonto` file refers to only one package. To be more organized, we could define the datatypes of our model in a separate package called *CoreDatatypes*. We can see the definition of this package on Listing 3.2. The first element is the datatype *PhoneNumber*, created to be able to hold information about a simple phone number, with country code and the rest of the number at *bodyNumber*. We define each attribute as int because phone numbers can only have integer digits. Because numbers are broader than integers, we need to declare this datatype called *int*, which is a specialization of *number*, representing all integer numbers. This definition of integer is only nominal, and Tonto does not allow the creation of any logic

restriction to guarantee that the number is an integer. After the type of the attribute, we can declare the cardinality of that attribute. Cardinality declarations are optional, with default exactly one (i.e., `[1]`).

The last declaration is a specific kind of datatype, an *enumeration*. In enumerations, we have *literals* declared inside it, and when we define an attribute datatype as an enumeration, the meaning is that a value of this datatype is one of the possible literals defined. In this case, the eye color could be blue, green, brown, or black.

Listing 3.2 – Tonto package of custom DataTypes.

```
1  package CoreDatatypes
2
3  datatype PhoneNumber {
4    countryCode: int [1]
5    bodyNumber: int [1]
6  }
7
8  datatype int specializes number
9
10 enum EyeColor {
11   Blue,
12   Green,
13   Brown,
14   Black
15 }
```

After defining the core datatypes in a different Tonto file, we want to use it to define the attributes of *Person*. In this case, because elements are on different packages, we must first import the contents of our *CoreDatatypes* package to our *PersonPhases* package. If these are not imported, the package will not be able to make a reference to the other element. This decision was made to prevent many suggestions from polluting the auto-completion functionality of VS Code as models grow.

On Listing 3.3, we can see the new version of the kind *Person* with attributes and the correct datatypes. There are two options for using imported elements: with their name (e.g., EyeColor); with a qualified name (e.g., CoreDatatypes.PhoneNumber); the developer can decide which one is better to avoid ambiguity. In order to make it more complete, the cardinality of each attribute was added. The default cardinality is exactly one [1] for attributes and association ends if none is defined. Because people usually have a name, and they can have more than one, the cardinality of this attribute is [1..*].

There is also the possibility of specifying meta-attributes of attributes, enclosed by brackets. The *birthDate* attribute has a meta-property called `const` (i.e. immutable). In addition, the attribute *preferredNames* has the meta-property `ordered`, meaning that this list of preferred

names has an order of preference. Every meta-property in attributes needs to be defined inside of brackets at the end of the attribute declaration. In classes, by default, every attribute is not const, not ordered, and not derived. In *Datatypes*, attributes are always immutable. Finally, people could have one eye color or two in special cases, and everyone can have no phone number or many of them.

Listing 3.3 – Kind Person with attributes with custom Datatypes.

```
1  import CoreDatatypes
2
3  package PersonPhases
4
5  kind Person {
6    name: string [1]
7    preferredNames: string [1..*] { ordered }
8    age: number [1]
9    birthDate: date [1] { const }
10   eyeColor: EyeColor [1..2]
11   phoneNumber: CoreDatatypes.PhoneNumber [0..*]
12 }
```

3.2.2 Generalization Set

Another important construct in OntoUML is a *Generalization set*, based on the same construct in UML. An association is formed between a general element and a group of its specializations through the use of a generalization set. They can be decorated with the keywords `disjoint` and `complete`. The first keyword indicates that each instance of the superclass can only instantiate a maximum of one of the subclasses. The second keyword defines scenarios where instances of the general class are required to instantiate at least one of the subclasses. Similarly to the direct specializations on the model in Figure 3.1, we could utilize generalization sets. Tonto generalization set syntax is based on ML2 (FONSECA, 2017). On Listing 3.4, line 7, we have a Tonto short syntax implementation of this generalization set, being the simplest form to declare it and fitting in one line.

On Listing 3.5, from lines 5 to 8, we have the same generalization set in an expanded syntax using brackets that is more organized for declarations specializing more elements. The keywords `general` determine the general element, while the keyword `specifics` defines all specific elements, separated by a comma.

Lastly, on Listing 3.6 and based on an example presented by Fonseca (2017), we can add the type *PersonTypeByAge* as a categorizer of the generalization set. The type *PersonTypeByAge* is declared, and each phase previously declared now are an instance of this type. From lines 7 to 11, we have the same generalization set in an expanded syntax, however, now specifying

the categorizer of this generalization set. It is important to notice that gensets mentioned on Listings 3.4, 3.5 and 3.6 are all defined as `disjoint` and `complete` on this model, but if the model requires, it could be only disjoint, only complete, or none of them, omitting its keywords.

Listing 3.4 – Example of simple usage of Generalization Sets.

```
type PersonTypeByAge

phase Child
phase Teenager
phase Adult

disjoint complete genset PhasesOfPerson where Child, Teenager
    , Adult specializes Person
```

Listing 3.5 – Example of Generalization Set with complete syntax.

```
phase Child
phase Teenager
phase Adult

disjoint genset PhasesOfPerson {
  general Person
  specifics Child, Teenager, Adult
}
```

Listing 3.6 – Example of Generalization Set with categorizer.

```
type PersonTypeByAge

phase Child (instanceOf PersonTypeByAge)
phase Teenager (instanceOf PersonTypeByAge)
phase Adult (instanceOf PersonTypeByAge)

disjoint complete genset PhasesOfPerson {
  general Person
  categorizer PersonTypeByAge
  specifics Child, Teenager, Adult
}
```

3.2.3 Ontological Natures

Additionally, we have the possibility of declaring nature restrictions for elements in Tonto. As explained in Chapter 2, elements could have nature restrictions based on their

stereotypes. Because ultimate sortals already have a nature restriction, they cannot specify any different nature. Taking for example the model in Figure 5, classes *Young Adult* and *University Student* are base sortal stereotypes, ≪*phase*≫, and ≪*role*≫, respectively. It means that they don't carry an ontological nature restriction by themselves, and this should be provided by the ultimate sortal that they specialize. As a consequence of both elements specializing the kind *Person*, it means that they carry the nature of a `kind`, which is functional-complex.

Another option is to define nature restrictions explicitly in the class declaration. This is useful when having non-sortal elements that can specify the nature of their instances (and since they are non-sortals, they do not specialize an ultimate sortal from which to inherit the nature of their instances). One example could be a `category` named *Social Entity* that is restricted to functional complexes, and is specialized by the kind *Hospital*. In Listing 3.7 we can see the nature declaration of this example.

One addition that Tonto has in comparison with OntoUML is a syntactic sugar for a nature called *of objects*, representing Substantial Endurants. It is a short syntax to declare the 3 natures of substantials: *functional complexes*, *collectives* and *quantities*.

Listing 3.7 – Example of nature declaration in Tonto

```
1 package TontoNatures
2
3 category SocialEntity of functional-complexes
4
5 kind Hospital specializes SocialEntity
```

3.2.4 Relations

Relations (or UML associations) are an important part of conceptual modeling in OntoUML, providing meaning to the connections between elements. They are also a core concept in Tonto and can be declared in two different ways. The first way is called an internal relation, which is defined inside the body of the declaration of a class. The second way is called an external relation and can be defined outside the body of class declarations.

We use the OntoUML model in Figure 6 to discuss relations in the sequel. This model presents University as a ≪*kind*≫ that specializes the category *Organization* of functional complexes. The university possibly has many *Classrooms*, and that is stated with the aggregation relation between them named 'has'. Because the classroom is a specific kind of room, it specializes the kind *Room*. Also, the university has one or more *Departments*, and this is expressed with an aggregation relation with the stereotype ≪*componentOf*≫. The department is composed by a *JuniorStaff* and a *SeniorStaff*, also represented with an aggregation relation. These staffs are subkind of the collective *Staff*. Moreover, we have a roleMixin *Employer*, that aggregates properties in common to everything that hires someone. The role *UniversityEmployer*

represents the capacity that this university have to hire employees, for example for its junior and senior staff.

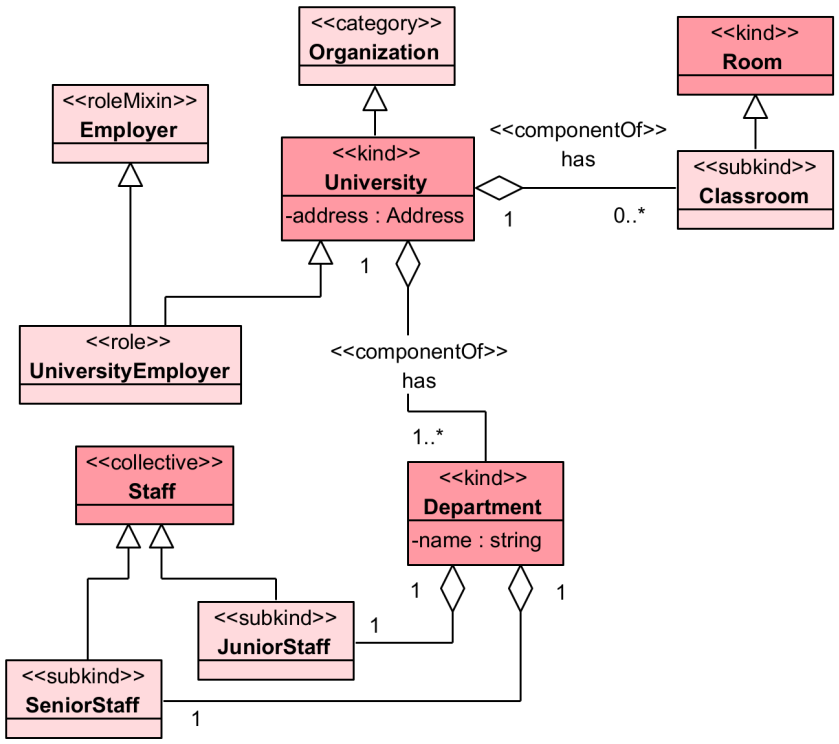

Figure 6 – Example model representing a University with departments and classrooms.

From Figure 6 we can represent in Tonto the kind *University* with its relations in Listing 3.8. First, on lines 3 and 4 we have a relation named *has* with the stereotype ≪*componentOf*≫, stated with a keyword prefixed with the `@` symbol (a syntax inspired in Java annotations). Every OntoUML relation stereotype can be declared that way. Being an internal relation, the element of the first end is the class holding the relation and its name is omitted. Relation specification works like a mirror (inspired in the UML notation for association, which is a line with opposing association ends), where the definition order for the element in the first end is inverted for the second end. The order of declaration in the first relation end is, after the relation stereotype: class ID (in case of an external relation); first end meta-attributes; first end name; first end cardinality and the relation connector with its name. Meta-properties of a relation can be `ordered`, `const` and `derived`.

Moreover, the first cardinality represents the cardinality on the University end, and the second cardinality of [1] represents the Department end of the relation. The default cardinality of relations is one to one, and can be omitted in that case. Because it is an aggregation relation, the keyword `‘<>- -’` is used. For associations, the keyword is `‘- -’`, and for composition is `‘<o>- -’`. These keywords defines the ‘middle’ point of an relation, where the relation name can be defined after this connector keyword, and an extra keyword `‘- -’` is added after its name.

After the second end class ID, two other properties can be defined: a specialization of the relation and the inverse of this relation. Specialization uses the same `specializes` keyword used in classes, and inverses use the `inverseOf` keyword, both of them followed by the relation ID.

Listing 3.8 – Example of the University kind in Tonto with internal relations.

```
kind University specializes Organization {
  address: Address
  @componentOf
  [1] <>-- has -- [1..*] Department
  @componentOf
  [1] <>-- [0..*] Classroom
}
```

Further, from Figure 6 we can express the relation between *JuniorStaff* and *SeniorStaff* externally. Listing 3.9 shows the definition of these two external relations. The only difference between them is the need for the keyword *relation* and the name of the first end class; every other component of the relation is declared in the same way as an internal relation and in the same order.

Listing 3.9 – Example of an external Relations.

```
relation Department [1] <>-- [1] JuniorStaff
relation Department [1] <>-- [1] SeniorStaff
```

Listing 3.10 shows the model from Figure 6 completely transformed to Tonto, including every class and relation.

Listing 3.10 – Example of basic University model in Tonto.

```
import CoreDatatypes
package University

category Organization

kind University specializes Organization {
  address: Address
  @componentOf
  [1] <>-- has -- [1..*] Department

  [1] <>-- [0..*] Classroom
}

```

```
14 kind Room
15 subkind Classroom specializes Room
16
17 kind Department {
18   name: string
19   [1] <>-- [1] JuniorStaff
20 }
21
22 collective Staff
23
24 subkind JuniorStaff specializes Staff
25 subkind SeniorStaff specializes Staff
26
27 relation Department [1] <>-- [1] JuniorStaff
28 relation Department [1] <>-- [1] SeniorStaff
29
30 roleMixin Employer
31
32 role UniversityEmployer specializes Employer, University
```

Lastly, we could define relations between elements in different packages. The final model, joining the University and PersonPhases packages in Figure 2, has relator elements that connect both packages. The first one is the *EmploymentContract* relator, representing the relationship between an *Employee* and an *Employer*. The second relator represents the Enrollment of a university student and the university. In Listing 3.11, we can see how this is declared on the University package. The final version of the University model in Tonto can be found in Appendix B.

Listing 3.11 – Example of relators added to the University package.

```
1 role Employee specializes Person
2 role UniversityProfessor specializes Employee
3
4 relator EmploymentContract {
5   @mediation
6   [1..*] -- [1] Employee
7
8   @mediation
9   [1..*] -- [1] Employer
10 }
11
```

```
12 relator Enrollment {
13   @mediation
14   [0..*] -- [1] University
15
16   @mediation
17   [1..*] -- [1] UniversityStudent
18 }
```

A template for the full definition of internal and external relations can be found in Listing 3.12, containing every possible element to be declared on a relation. In lines 2 and 3, we have the template of an internal relation from the kind *Person*, and on lines 6 and 7 we have the template of an external relation.

Listing 3.12 – Template of relation definitions in Tonto

```
1 kind Person {
2   @stereotype
3   ({ metaAttributes } firstEndName) [1] <>-- relationName --
      [1] ({metaAttributes} secondEndName) SecondClass
      specializes OtherRelation
4 }
5
6 @stereotype
7 relation FirstClass ({ metaAttributes } firstEndName) [1] --
    relationName -- [1] ({metaAttributes} secondEndName)
    SecondClass specializes OtherRelation
```

Table 1 shows a full list of available stereotypes for relations, with their compatible kind of association.

3.3 Tonto Visual Studio Code Extension

With Langium, a Language Server Protocol (LSP) is generated for a textual language, and a Visual Studio Code extension is created. Figure 7 shows the usage of this extension, demonstrating the University example. In VSCode, we open a directory that is the root of our workspace, and every file inside it is part of the project; similarly, the Tonto extension[2] considers every file currently in the workspace with the `.tonto` extension to be part of the same project, with every file being a different package. Every file is automatically processed by the langium library, creating a LangiumDocument. This document goes through seven states, from parsing the AST to computing scopes, linking cross-references, doing validations, and then waiting for new changes.

[2] <https://marketplace.visualstudio.com/items?itemName=Lenke.tonto>

Table 1 – Table of available relation stereotypes in Tonto

Stereotype	**Association (--)**	**Aggregation (<>--)**	**Composition(<o>--)**
material	x		
derivation	x		
comparative	x		
mediation	x		
characterization	x		
externalDependence	x		
componentOf		x	x
memberOf		x	x
subCollectionOf		x	x
subQuantityOf			x
instantiation	x		
termination	x		
participational	x		
participation	x		
historicalDependence	x		
creation	x		
manifestation	x		
bringsAbout	x		
triggers	x		
inherence	x		
value	x		
formal	x		

The extension's first feature provides a way to visualize errors in the model because of the validators implemented automatically by Langium and those custom validators added to the extension based on OntoUML semantic rules. In Figure 8, we have an example of the semantic error flagged by a custom validator. This error occurs because a class with the stereotype *role* must specialize one ultimate sortal, and VScode also displays the error inline. This real-time visualization allows the modeler to spot errors in every new element added (without the need for special interventions such as clicking on a button).

Another important feature is the commands available at the bottom of the editor and at the command palette. They are responsible for executing the same commands available for the Tonto CLI. From Figure 7, we can see the following buttons:

- **Tonto -> JSON**: This generates a JSON from the model.
- **Tonto -> gUFO**: Validate the model with ontouml-server API.
- **JSON -> Tonto**: This generates a Tonto project from a JSON file.
- **Validate Model**: Validate the model with ontouml-server API.
- **TPM Install**: Install dependencies based on the Tonto Package Manager (TPM)

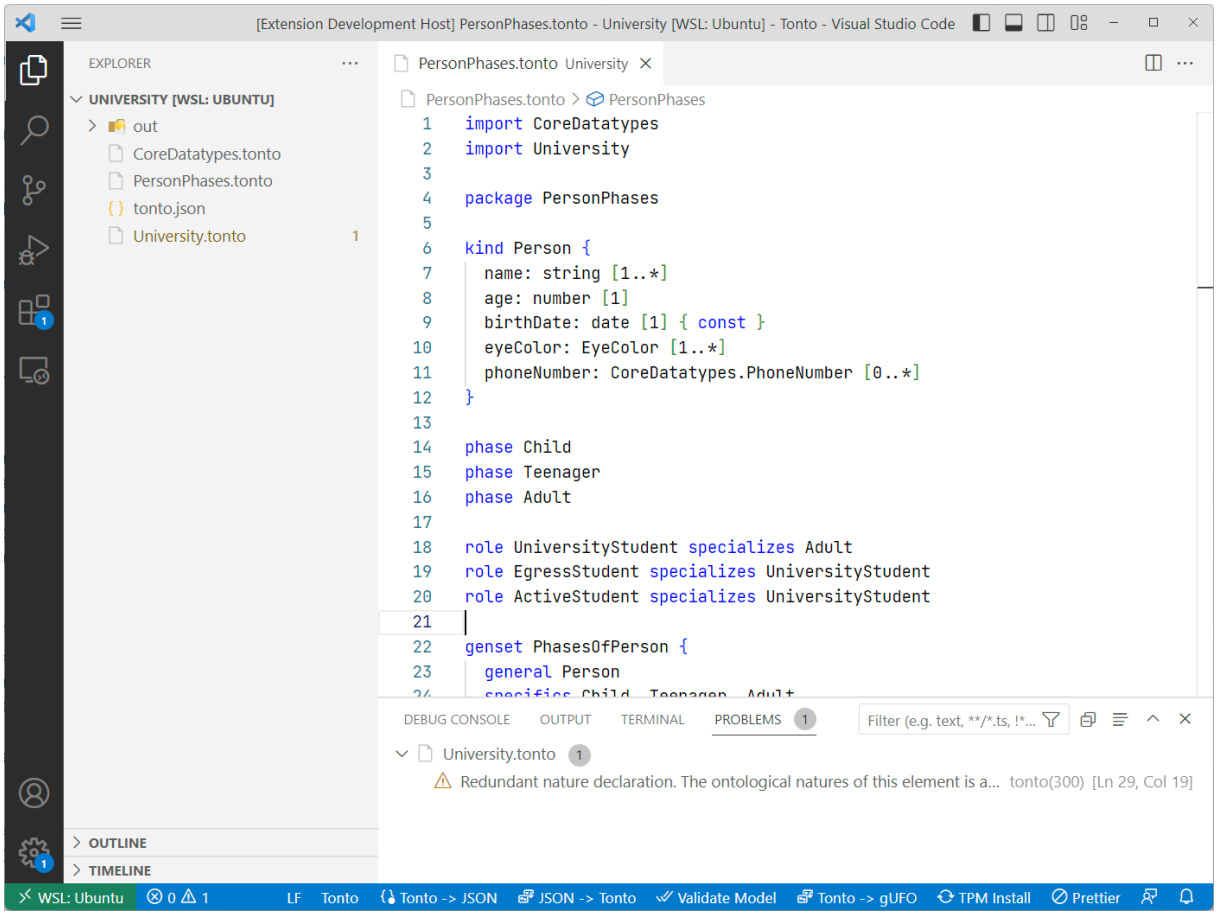

Figure 7 – VS Code running the Tonto extension with the PersonPhases package code

Every command automatically recognizes the workspace's root and executes from that directory. But, in some cases, it is needed to run a command in another directory in the same way provided by the CLI. In that case, they are available in the command palette and will ask the user to input the directory to be analyzed.

Finally, this extension enables the user with smart code completion and ready-to-use snippets that empower and fasten the development of models. An example of how code completion works can be seen in Figure 9. Because of that feature, it is easy to get used to the syntax, and there is no need to write some boilerplate code that could slow development. Figure 10 shows an example of a list of snippets available for Tonto in the current context, and Figure 11 shows the usage of an internal relation snippet.

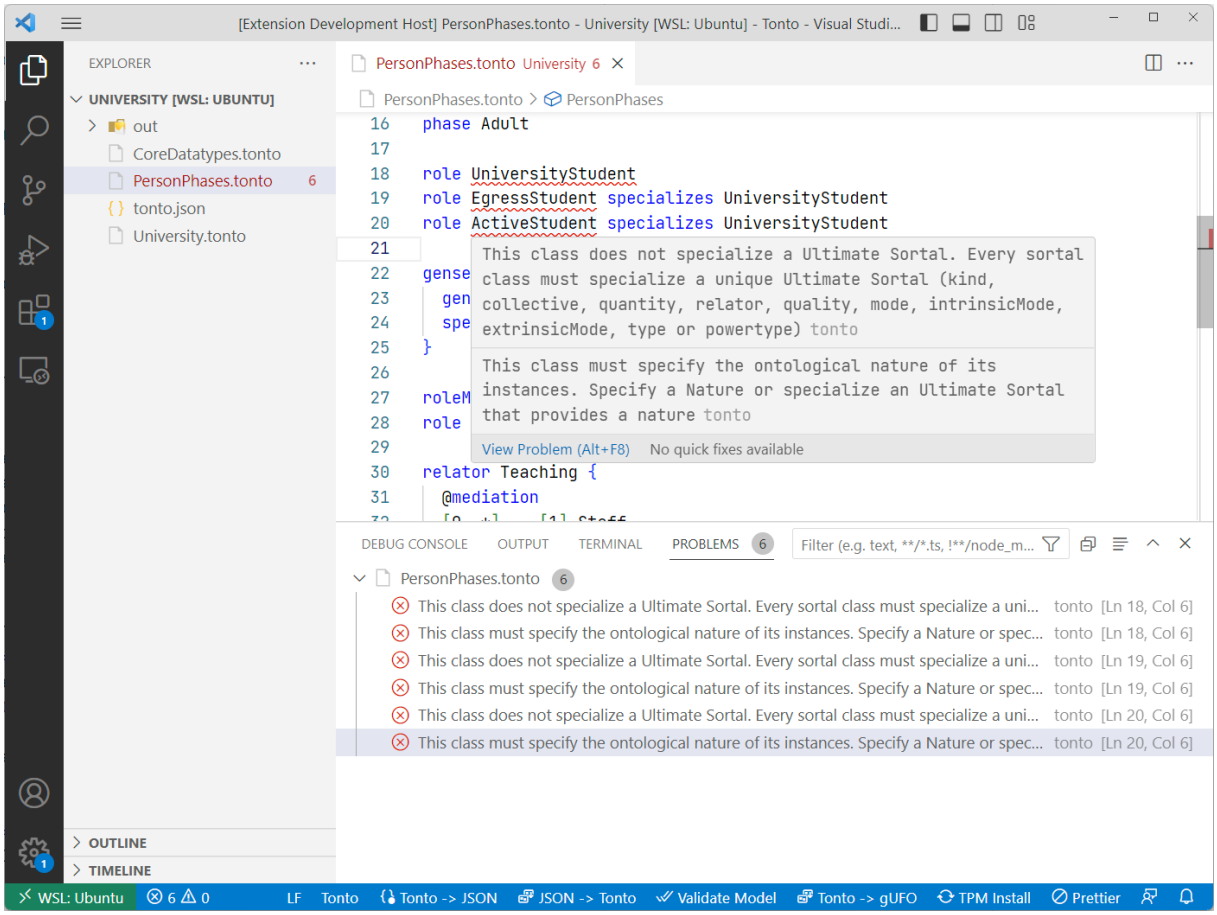

Figure 8 – VS Code running the Tonto extension showing the problems of the model in the problems tab and at the current line with error

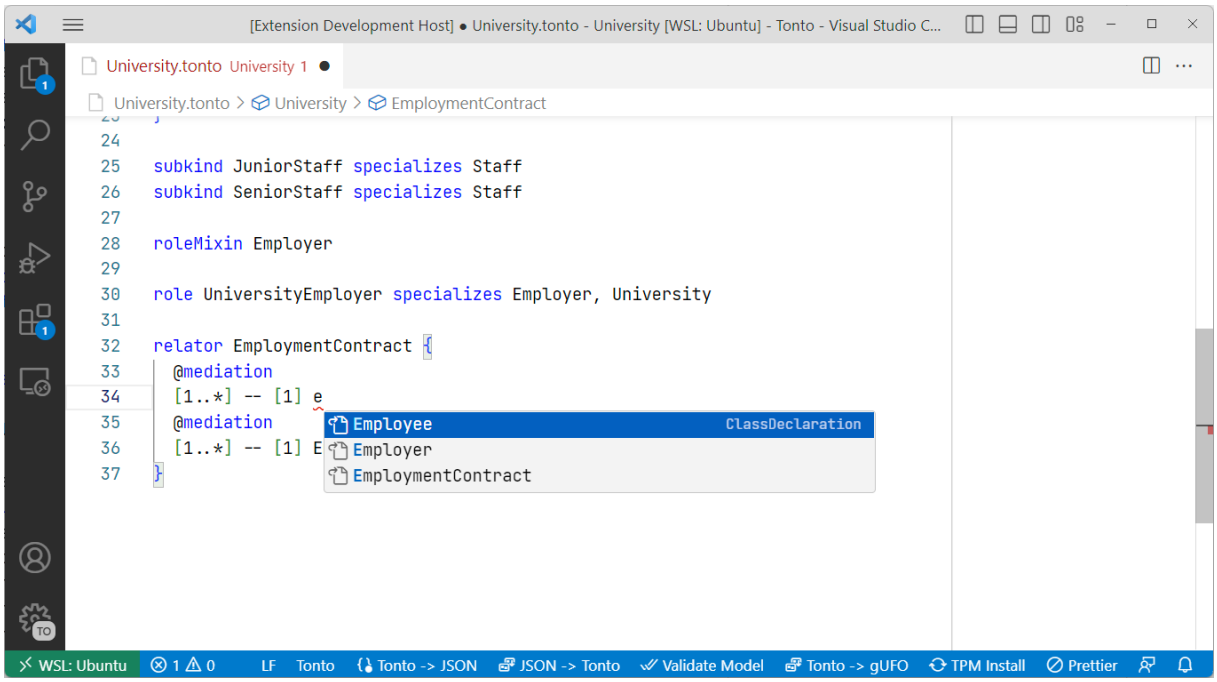

Figure 9 – Tonto VSCode extension showing an auto-complete example.

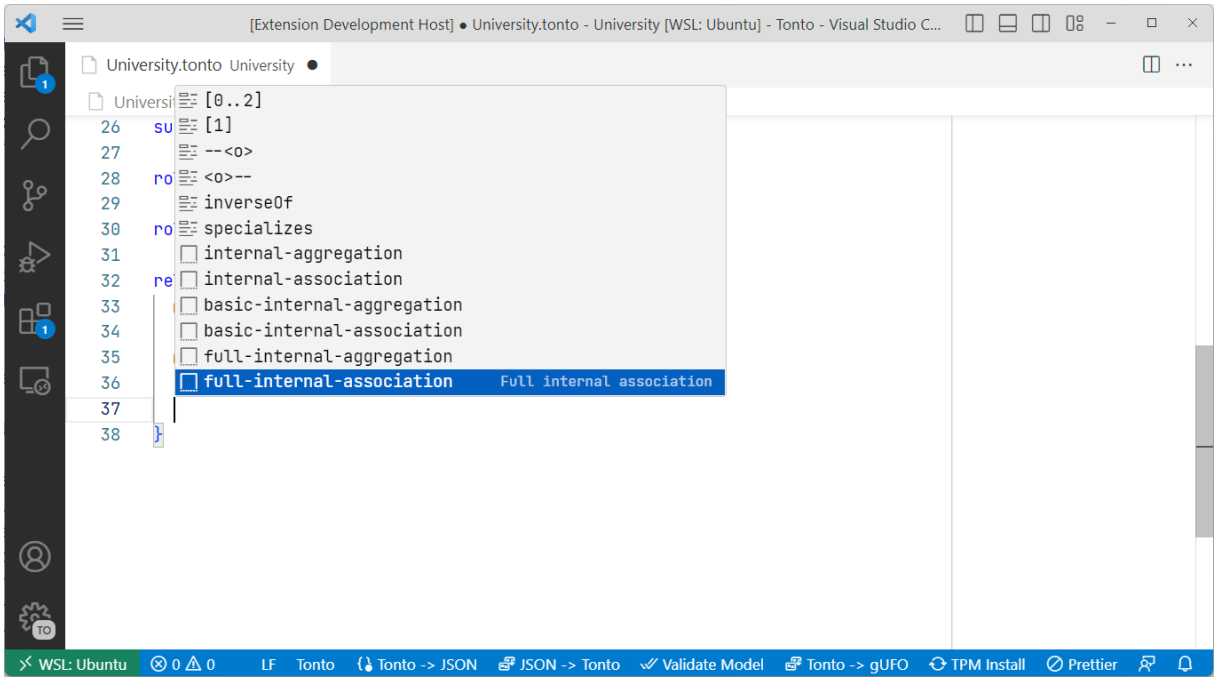

Figure 10 – Tonto VSCode extension showing snippets list.

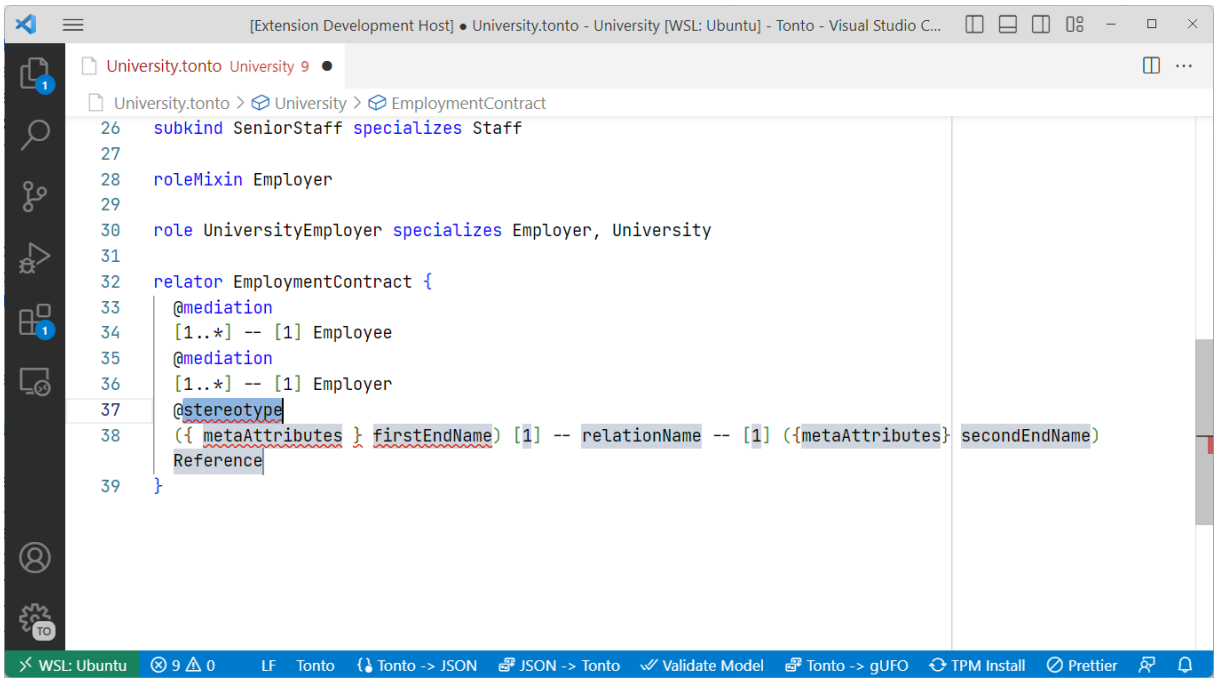

Figure 11 – Tonto VSCode extension showing the usage of an internal relation snippet.

3.4 Tonto Validators

In this section, we will describe each validator implemented on Tonto LSP. Every validator runs in the context of a package automatically when there are changes in code. All validators were inspired by the ones developed at ontouml-js, with some changes due to differences between Tonto and JSON syntaxes.

3.4.1 Ultimate Sortal specialization validator

This validator checks whether the class declaration is an Ultimate Sortal and whether it specializes other Ultimate sortals. Because ultimate sortals should not do this specialization, this shows an error as presented in Figure 12.

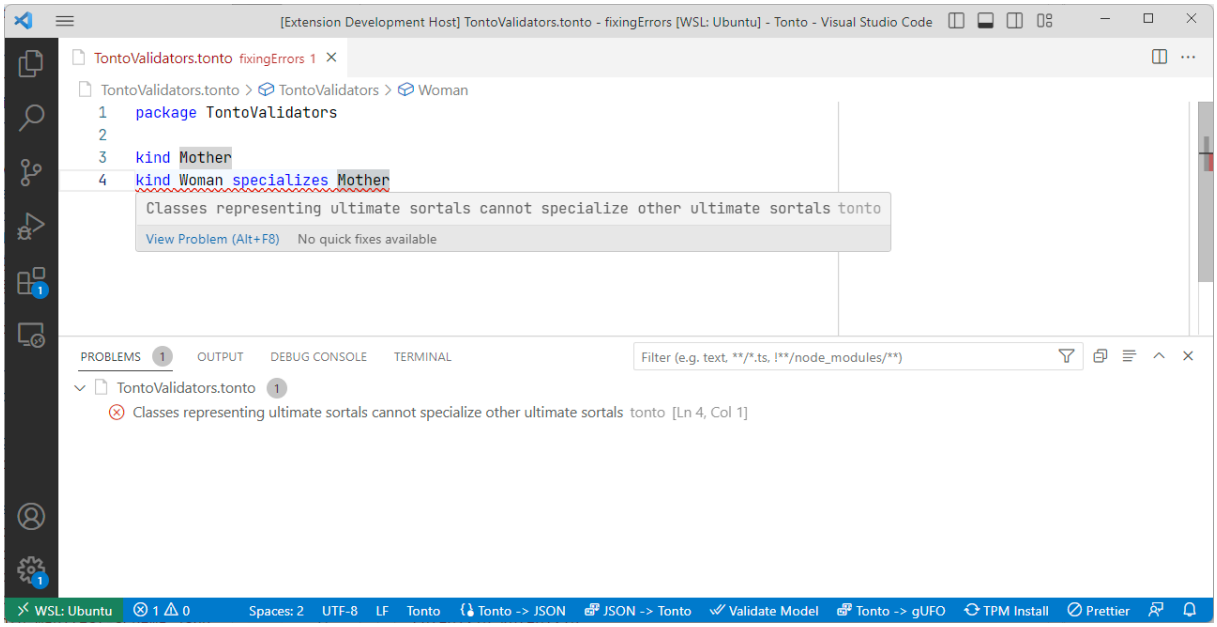

Figure 12 – Tonto VSCode extension showing an error of the Ultimate sortal validator.

3.4.2 Sortal specializes more than one ultimate sortal validator

This validator checks whether the class declaration is a sortal and calculates how many ultimate sortals it specializes directly or indirectly, analyzing every element until the topmost one. If it specializes in more than one, it displays an error as shown in Figure 13, informing the modeler that the class should specialize only one ultimate sortal.

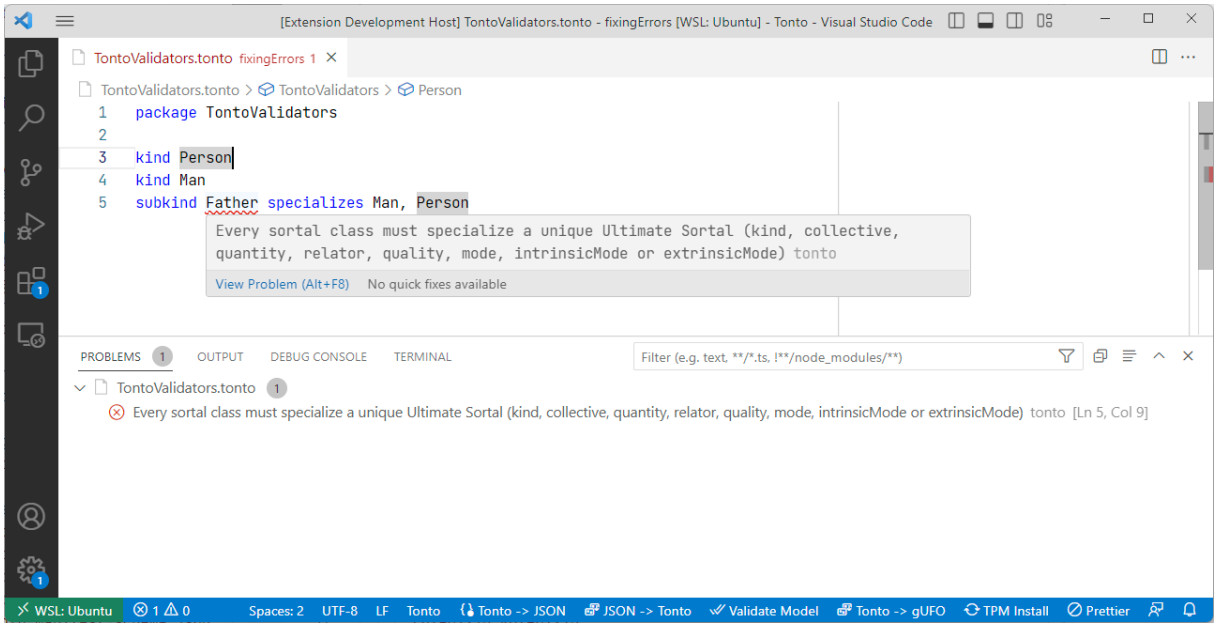

Figure 13 – Tonto VSCode extension showing an error of the Sortal specializes Ultimate sortal validator.

3.4.3 Sortal should specialize ultimate sortal validator

This validator checks whether a class declaration is a sortal and calculates how many ultimate sortals are specialized, similarly to the previous validator. If it does not specialize any, it shows an error that it should specialize one. In Figure 14 we can see an example of this error.

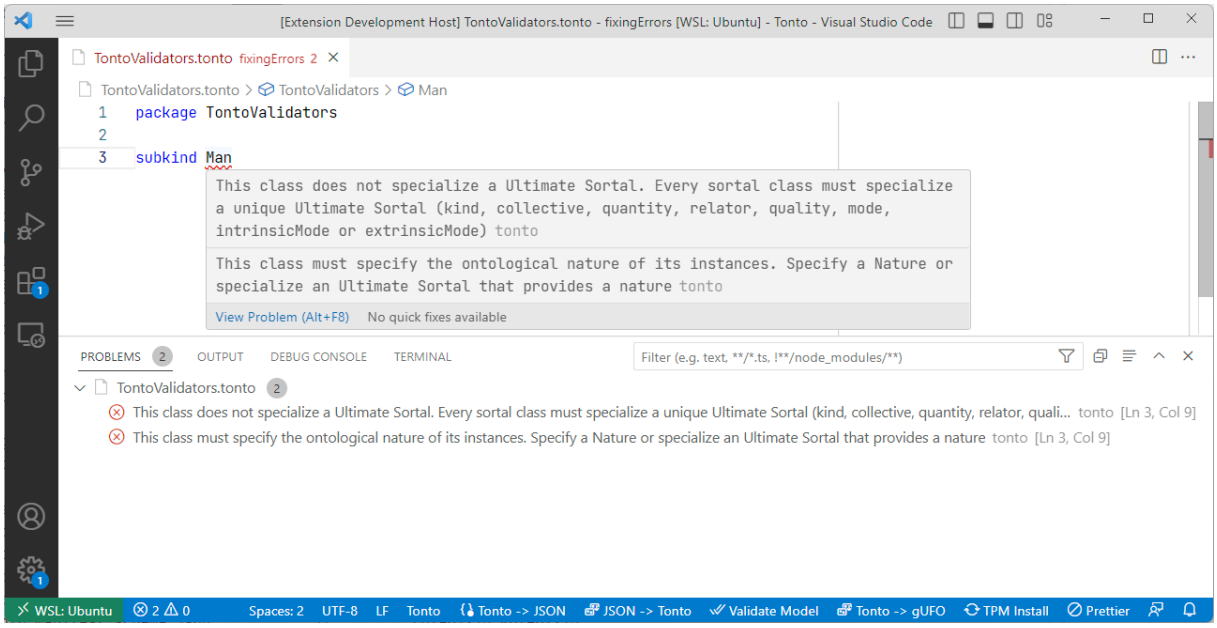

Figure 14 – Tonto VSCode extension showing an error of the sortal should specialize ultimate sortal validator.

3.4.4 Rigid specializes Anti-rigid validator

This validator verifies whether, in a generalization between two classes, the general class has an Anti-rigid stereotype and the specific class has a rigid or semi-rigid stereotype, showing an error like in Figure 15 if that is the case.

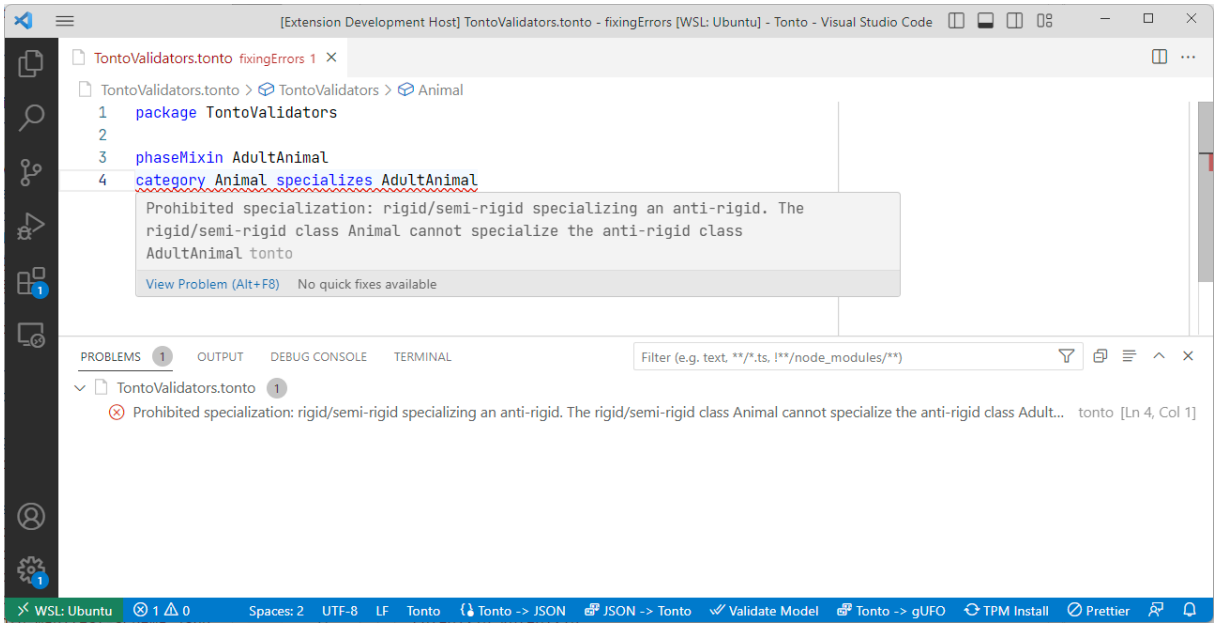

Figure 15 – Tonto VSCode extension showing an error of Rigid element specializing Anti-rigid validator.

3.4.5 Compatible Natures of sortals validator

This validator verifies whether the class declaration is a (non-kind) base sortal (subkind, phase, role, historicalRole) and whether it defines a nature for its instances. The nature can be defined explicitly with tonto grammar, or it could be inherited from specialized classes. It shows an error when a nature specification is missing. For example, *subkind Customer* should specialize an ultimate sortal like *kind Person*. Figure 16 shows an example of this error.

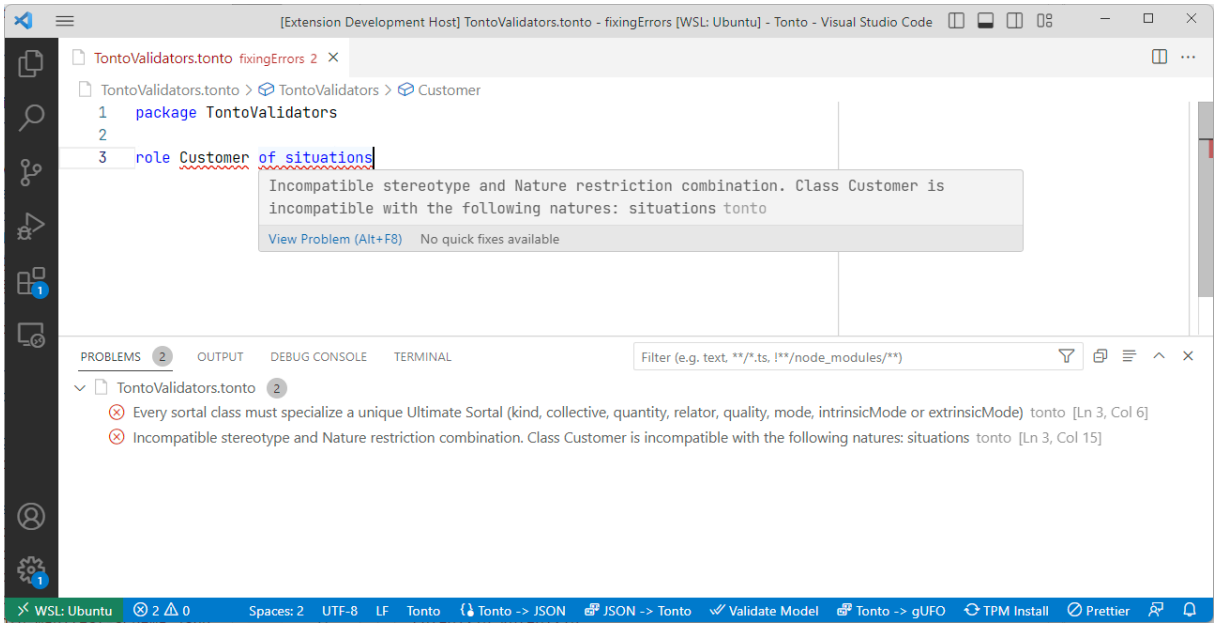

Figure 16 – Tonto VSCode extension showing an error of the Compatible natures of sortals validator.

3.4.6 Compatible Natures validator

This validator verifies if non-sortals or base sortals specialize incompatible natures based on elements that it specializes in. For example, if a class is declared as *role Customer of objects* and it specializes in the kind *Person*, it should have an error like in Figure 17

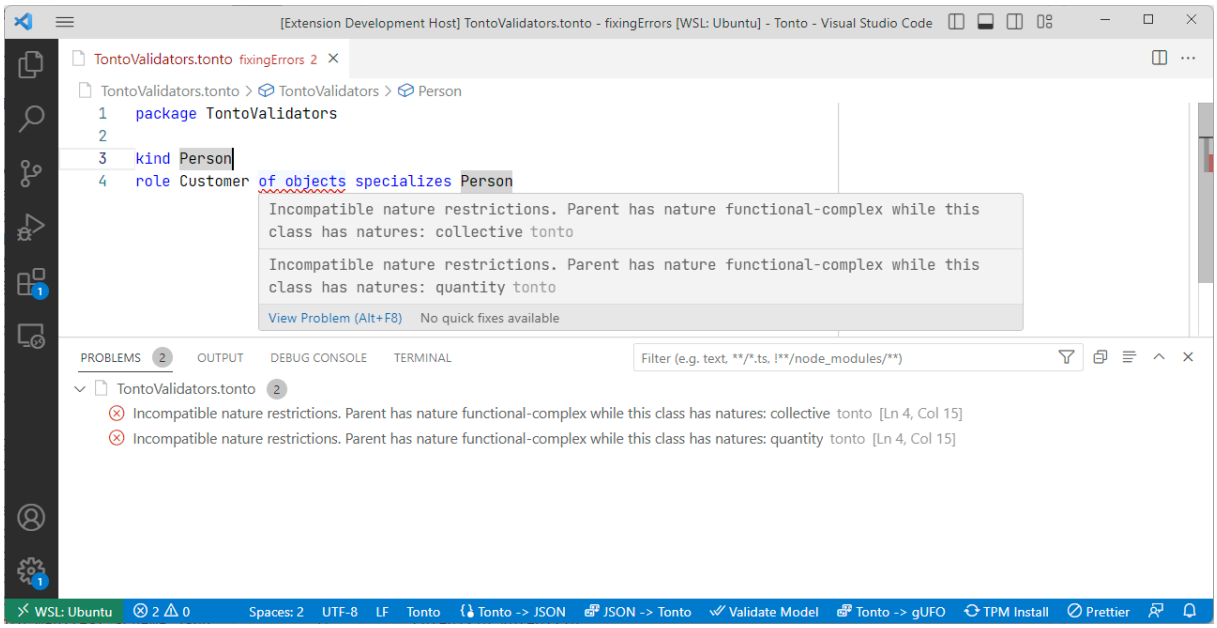

Figure 17 – Tonto VSCode extension showing an error of the Compatible Natures validator

3.4.7 Redundant Natures

This validator verifies if an explicit declaration of nature is redundant for the element. This happens for ultimate sortals that already have a defined nature and cannot specialize other natures. Also, this happens for cases when a sortal or non-sortal inherits its nature from an ultimate sortal and tries to explicitly define the same nature when that is not needed. On Figure 18 we have two examples: the first is a kind, that already have the nature of functional complexes, so there is no need to specify it, and an example of a role Student that already have its nature defined by specializing the kind *Person*, with no need to specify again the nature of functional complexes.

Figure 18 – Tonto VSCode extension showing a warning of the Redundant Natures validator

3.4.8 Other validators

Lastly, we defined other smaller validators to ensure code quality of the model:

- **Class without more specific keywords:** This validator shows a warning if an element is declared using the keyword `class` and do not provide any nature.
- **Check duplicates:** This validator checks for duplicate names of class declarations, relations, datatypes, and attributes.
- **Check circular specialization:** This validator checks whether an element with specializations does not have a cyclic specialization.

3.4.9 Tonto CLI

To be able to expand the use of Tonto models, a desired feature is to export to another format with external usage. Also, instead of recreating every model on another form from scratch, being able to transform an existing model into the Tonto format is useful. That is the purpose of the creation of Tonto CLI[3], a Command Line Interface that enables Tonto projects to be transformed into JSON files following an OntoUML JSON schema or to gUFO-based OWL ontologies, to import a model from JSON to Tonto and to validate the model on the ontouml-server API.

The CLI is a Node Package Manager (NPM) package and requires a proper installation of node.js. Because of that, Tonto-CLI works on Windows, MacOS, and Linux and can be installed as a global package, being able to call the command "tonto-cli" from any folder in a terminal. The following commands are available:

- **Generate**, used with `tonto-cli generate [directoryName]`, which transforms a model in the provided directory to JSON.
- **Import**, used with `tonto-cli import [jsonFileName]`, which imports a JSON file as provided by input and generates a Tonto project from it.
- **Transform**, used with `tonto-cli transform [directoryName]`, which transforms the model provided to a gUFO-based OWL ontology using the Turtle syntax.
- **Validate**, used with `tonto-cli validate [directoryName]`, which sends the model to the ontouml-server API and returns every validation error provided.

3.5 Tonto Package Manager

With Langium and VSCode Extension, it is possible to create powerful Tonto projects with many files and packages, allowing the definition of large models. However, a great way developed in popular programming languages is the possibility to encapsulate code as a package to be distributed and reused on other projects, some of them being open-source, with its source code available for everyone. This was one of the reasons for the increasing development speed in many companies doing software development. Tonto Package Manager (TPM)[4] was created to provide the same capabilities for Tonto. TPM is based on popular solutions, for example, NPM (Node Package Manager)[5], SPM (Swift Package Manager)[6] and Cargo [7], the package managers for JavaScript, Swift, and Rust, respectively.

[3] <https://www.npmjs.com/package/tonto-cli>
[4] <https://www.npmjs.com/package/tonto-package-manager>
[5] <https://www.npmjs.com/>
[6] <https://www.swift.org/package-manager/>
[7] <https://doc.rust-lang.org/cargo/>

As a consequence of Package Managers being complex projects requiring a lot of effort to build every function, some decisions were made to simplify this task in our context. First, every project must be hosted on a git repository, with the possibility of defining a directory that the package is relative to the root folder, a version tag published on the repository, or a branch in case it is different from the main branch. Also, no *lock file* is created to manage current versions installed of dependencies, unlike the mentioned package manager npm. Because there is no complex code compilation, this file is not required in Tonto. Finally, every Tonto dependency is downloaded to a folder named *tonto_dependencies* that must not be uploaded to code versioning tools like git.

3.5.1 Manifest File

In order to be able to identify a project and its dependencies, a package manifest file is needed to be defined. A package manifest file is a file containing every piece of information required to make it complete and to distribute it for other projects to use. In Tonto, the manifest file must be named **tonto.json** and is defined in a JSON format with a correct definition of key-value properties. Listing 3.13 provides an example of a manifest file for the University Example. One possibility of dependency defined is for the CoreDatatypes package, defined on Listing 3.2, which could be a separate package.

Listing 3.13 – Example of a package manifest file for Tonto University example.

```
{
  "projectName": "UniversityModel",
  "displayName": "University Model",
  "authors": ["Matheus Lenke Coutinho"],
  "license": "MIT",
  "version": "1.0.0",
  "publisher": "UFES",
  "dependencies": {
    "CoreDataTypes": {
      "url": "https:github.com/url-to-package",
      "directory": "path/to/package"
    }
  },
  "outFolder": "out"
}
```

To download dependencies there are two possibilities. The first is by installing TPM as a global CLI in the same way that tonto-cli is installed. After doing that, it is only needed to run the command `tpm install` in the same directory as the tonto.json file of the project. The other option is to use the command button at the bottom of the extension, as shown by Figure 7, or at the command palette. After executing successfully, every dependency will be downloaded

to a temporary folder, processed by tpm, and moved correctly to the `tonto_dependencies` folder. All dependencies are downloaded recursively, so they can also be Tonto projects with their dependencies. Figure 19 shows an example of a TPM message that dependencies were installed successfully.

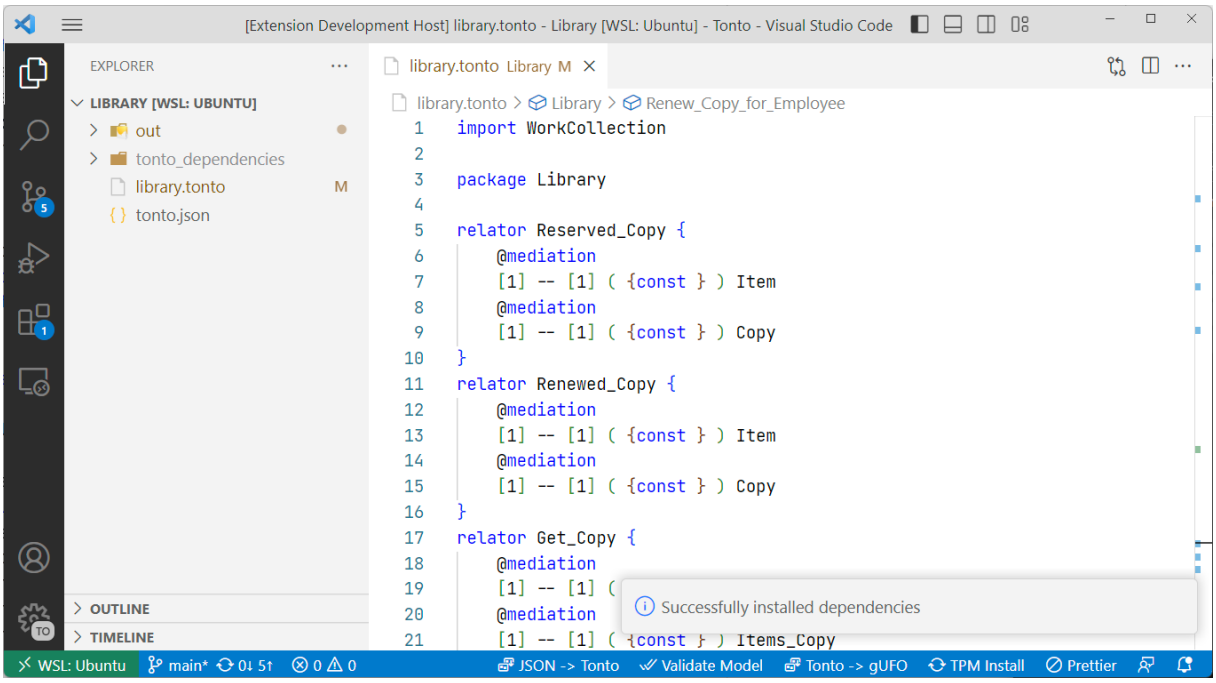

Figure 19 – Tonto VSCode extension showing TPM message that dependencies were installed successfully.

3.6 How to use Tonto

In order to use Tonto, the VS Code extension can be downloaded from the marketplace (<https://marketplace.visualstudio.com/items?itemName=Lenke.tonto>). In order to create a project, there is a getting started tutorial on <https://github.com/matheuslenke/Tonto> that can be followed to understand more about how Tonto works. Every functionality is already available at the VS Code extension, but if a CLI is needed, Tonto-CLI <https://www.npmjs.com/package/tonto-cli> and Tonto Package Manager <https://www.npmjs.com/package/tonto-package-manager> can both be installed globally via NPM.

4 An Example of Tonto Model: Library

In this chapter, we will experiment with features of Tonto tools in a complex model to evaluate its performance and capabilities. The selected model is the Library Ontology model[1], selected from a catalog of OntoUML models created by third parties over the years. It describes a library system, containing elements about collections of materials like videotapes and books and copies of these items provided to students and professors. It describes the student's and employee's ability to borrow materials from these collections, and what roles are played by these students. It also describes the contracts that students and employees have with a university. Appendix D contains the diagrams of this model. Figure 25 shows the original university diagram in OntoUML.

At first, the model is imported from JSON to Tonto, where some validations are made in order to check how information is transformed. Then, we export again the model from Tonto to JSON and to gUFO-based OWL, in order to evaluate if any information is lost. Lastly, JSON files and the Visual Paradigm Plugin concerning versioning the model will be evaluated in order to analyze the efficiency of each approach. Figure 20 shows a flowchart diagram representing this process of manipulating OntoUML models between Visual Paradigm, JSON, and Tonto that was made in this experiment.

4.1 Importing Model to Tonto

This model is also available as a JSON resource following OntoUML JSON schema[2]. Because Tonto allows importing models using this format, we can use Tonto CLI to do most of the work of creating it. The result of this import command can be seen in appendix C. It is important to note that the import command does not guarantee that the generated model will be equivalent to the source model, as not every element from OntoUML is already implemented in Tonto, or it is not mapped in the CLI import function. One example is the lack of support for the link between a relator and a material relation that is grounded on the relator.

After importing, because Tonto cannot represent declaration names with complex strings containing special characters and whitespaces, every whitespace is replaced by an underscore. Tonto does not allow duplicated element names on the same package, and a problem occurs with the element *Employee_Loan* on this model. In OntoUML, we have both a ≪*role*≫ and a ≪*relator*≫ with the same name in the same package, which is not allowed in Tonto. Because of Tonto validators, this error will instantly show on the problems tab using the VS Code extension, allowing the user to inspect duplicated names and modify them. We

[1] <https://scs-ontouml.eemcs.utwente.nl/model/c26c4ec3-c554-4988-9b19-581855aa4646>
[2] <https://scs-ontouml.eemcs.utwente.nl/distribution/74d94cee-b9f6-4f92-948c-e95ba4001530>

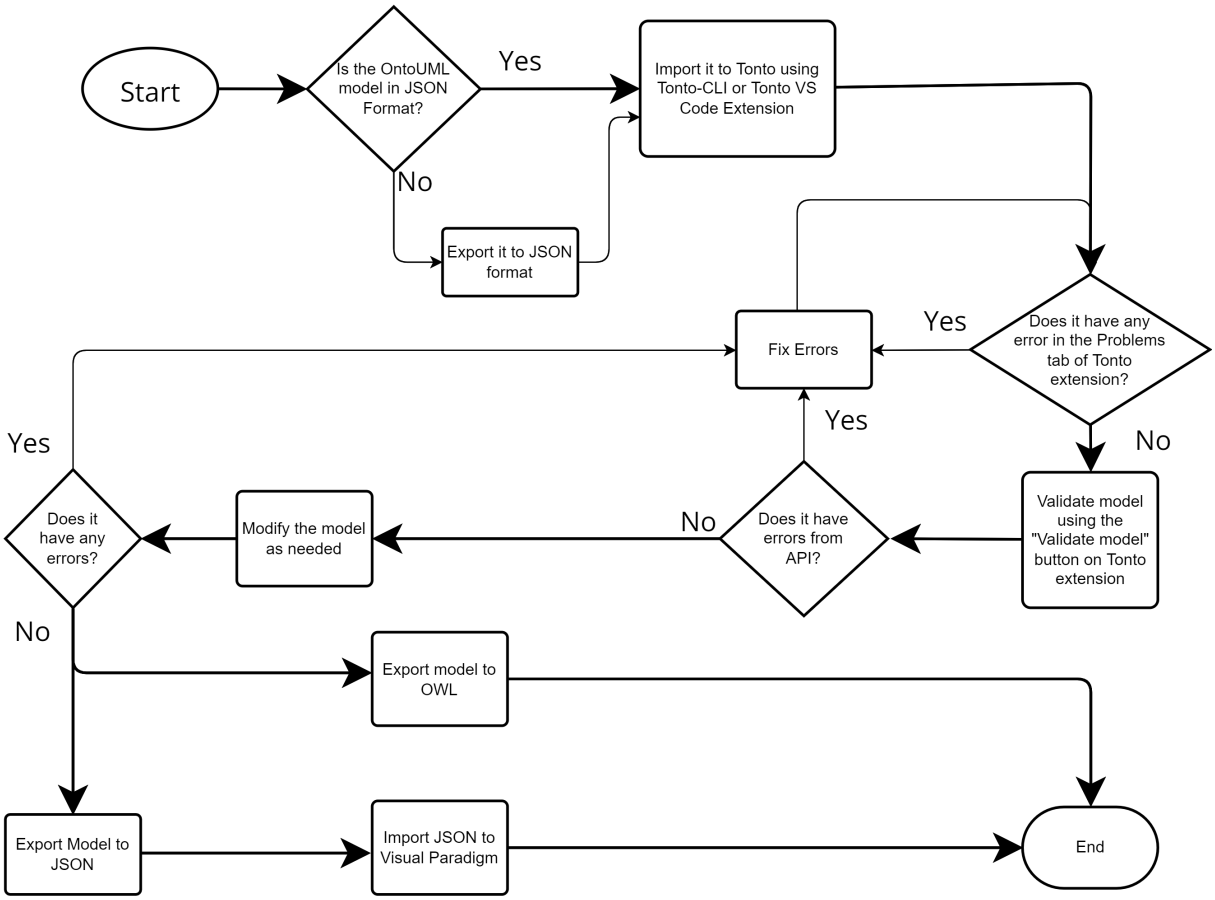

Figure 20 – Flowchart diagram showing the process of importing, exporting and validating a Tonto model.

then changed the name of the role element to *Employee_Loan_Role*. Another detail is that every cardinality was explicitly created for every relation as [1], however, that is the default value and does not need to be stated. The original name of the library package is 'Diagram1', and due to this name not expressing exactly the meaning of this package, its name was changed to 'Library'. With that, we can conclude that almost every element present in the original resource was created in Tonto, with some minor changes required. The only missing property is the link between the relator *Employee Contract* and the material relation because Tonto does not support it.

4.2 Validating the Tonto Model

Then, we can experiment with the validation method of the extension, to compare the returned errors from the ontouml-server API to what errors the extension is showing. After executing the command, we have as a result 14 validation errors, with the first two errors presented on Listing 4.1. Only two kinds of error are returned: (i) is an error about sortal classes needing to specialize a unique ultimate sortal and (ii) is an error about the missing tagged value of 'restrictedTo' in a class. Both of them are related to the role *Items_Copy* (from

the original *Item's Copy* name). The simplified result from the original model on the Visual paradigm validator is on Listing 4.2, demonstrating the same validation errors. Every error is identical between tools.

Listing 4.1 – Example of validation output from Visual Paradigm

```
1 ERROR: Every sortal class must specialize a unique ultimate sortal. The
    class Item's Copy must specialize (directly or indirectly) a unique
    class decorated as one of the following: kind, collective, quantity,
    relator, quality, mode
2 ERROR: The meta-property 'restrictedTo' is not assigned. The meta-
    property 'restrictedTo' of class Item's Copy must specify the
    possible ontological natures of its instances
```

Listing 4.2 – Example of validation output from Tonto validation command.

```
1 [error] Every sortal class must specialize a unique ultimate sortal:
2 The class Items_Copy must specialize (directly or indirectly) a unique
    class decorated as one of the following: <<kind>>, <<collective>>, <<
    quantity>>, <<relator>>, <<quality>>, <<mode>>
3 [error] The meta-property 'restrictedTo' is not assigned:
4 The meta-property 'restrictedTo' of class Items_Copy must specify the
    possible ontological natures of its instances
```

4.3 Exporting the Model

Now, we take the opposite path in order to evaluate the quality of transformations. We recreate the same diagram exporting the *Library* package to JSON, and then importing this JSON into the Visual Paradigm OntoUML plugin. After exporting to JSON, the file has 3335 lines because of the verbose syntax of JSON. The original JSON file of this model has 5209 files due to having information about the diagrams of this model, and this information is not generated by Tonto. The result of the process of importing it to Visual Paradigm is shown in Figure 26. Because Tonto lack of information about diagrams, it is necessary to adjust the layout in order to look similar to the original. This is a disadvantage at the moment, not allowing Tonto to save diagrams. Comparing them, there are only two differences: the names that were adjusted replacing whitespaces, and the lack of the link between the relator *Employee Contract* and a material relation that is grounded on the relator.

On Listing 4.3 there is a small part of the transformation of the *Library* model in Tonto to gUFO-based OWL. However, it was impossible to generate this model without modifying it, because ontouml-server API does not allow the transformation of models with semantic errors. Then, every class missing specialization of an ultimate sortal and nature definition needed to be fixed first. As a consequence of this format being really verbose, the generated model is too extensive and we need to omit information. The original generated file has 759 lines, being

larger than the Tonto version with 148 lines on the main package. We can verify that every element presented is created correctly by evaluating the generated file.

4.4 Analyzing Version Control

This section presents how version control is dealt with in Tonto, JSON files and Visual Paradigm, considering the use of the tool *git*. In order to test that, we add a new role called *AssistantProfessor* that specializes the existing role *Professor*. At first, we have Figure 21 containing the difference in code after adding that role in Tonto. On the right, we have the result of the git command to analyze differences in a versioned file , showing that the Tonto file only had 1 insertion in 1 line.

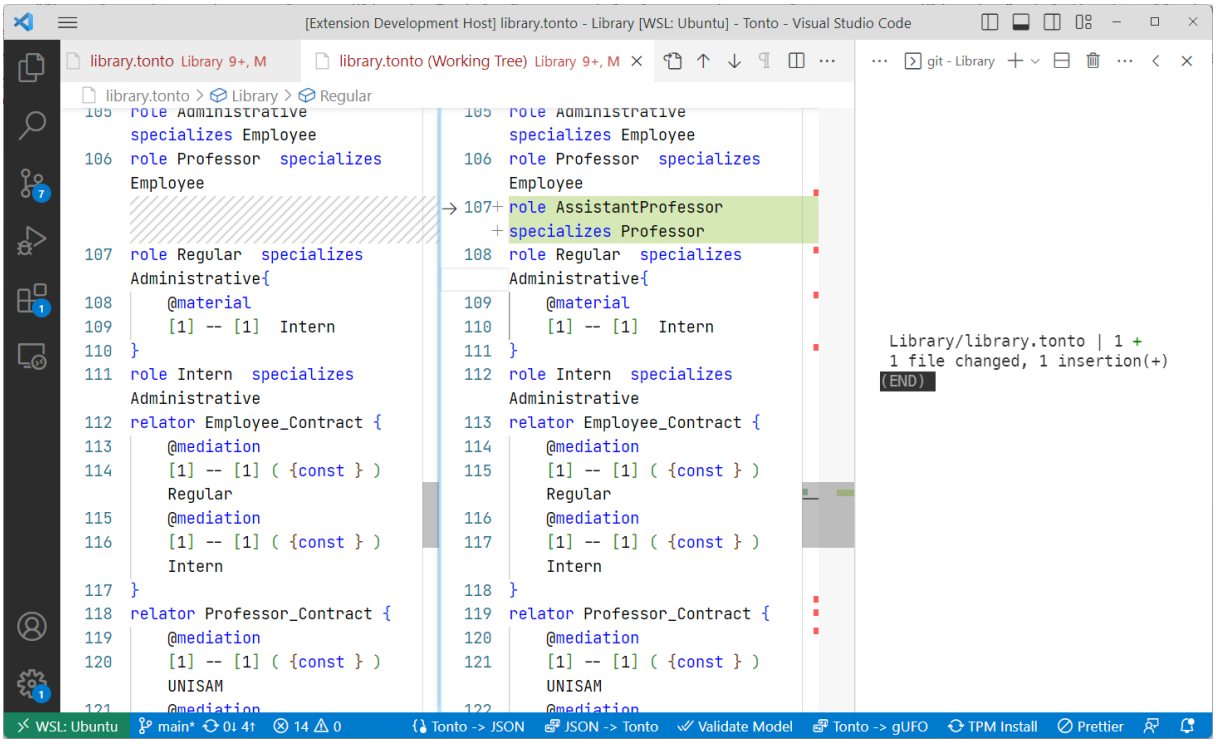

Figure 21 – VSCode showing the differences in Tonto file after adding the role *AssistantProfessor*.

We can compare the differences in a JSON file using the transformation from Tonto to JSON before the addition of *AssistantProfessor* (i) and after adding it (ii). Figure 22 shows the difference between (i) and (ii) in VS Code git difference explorer. The file on the right, in green, shows the inclusion of the *AssistantProfessor* and every other property related to it, based on ontouml JSON schema. From the left, we can visualize that we have 367 insertions and 316 deletions. The majority of the modifications are ids that were randomly generated again, however, even considering only the insertion of a new class and its specialization, we can verify that it changes more lines than in Tonto. In that way, a modeler can verify information faster when looking at a model in Tonto.

Listing 4.3 – Slice of a gUFO-based OWL file transformed from a Tonto Model

```
@prefix : <https://example.com#>.
@prefix gufo: <http://purl.org/nemo/gufo#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.

<https://example.com> rdf:type owl:Ontology;
    owl:imports gufo:.
:Reserved_Copy rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:Relator;
    rdfs:label "Reserved_Copy"@en.
:Renewed_Copy rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:Relator;
    rdfs:label "Renewed_Copy"@en.
:Get_Copy rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:Relator;
    rdfs:label "Get_Copy"@en.
:Item rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:FunctionalComplex;
    rdfs:label "Item"@en.
:Items_Copy rdf:type owl:Class, gufo:Role, owl:NamedIndividual;
    rdfs:label "Items_Copy"@en.
:Work rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:FunctionalComplex;
    rdfs:label "Work"@en.
:Videotape rdf:type owl:Class, gufo:SubKind, owl:NamedIndividual;
    rdfs:label "Videotape"@en.
:Periodical rdf:type owl:Class, gufo:SubKind, owl:NamedIndividual;
    rdfs:label "Periodical"@en.
:Book rdf:type owl:Class, gufo:SubKind, owl:NamedIndividual;
    rdfs:label "Book"@en.
:Dvd rdf:type owl:Class, gufo:SubKind, owl:NamedIndividual;
    rdfs:label "Dvd"@en.
:Collection rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:VariableCollection;
    rdfs:label "Collection"@en.
:Copy rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:FunctionalComplex;
    rdfs:label "Copy"@en.
:Copy_Reservation rdf:type owl:Class, gufo:Role, owl:NamedIndividual;
    rdfs:label "Copy_Reservation"@en.
:Student_Loan rdf:type owl:Class, gufo:Kind, owl:NamedIndividual;
    rdfs:subClassOf gufo:Relator;
    rdfs:label "Student_Loan"@en.
```

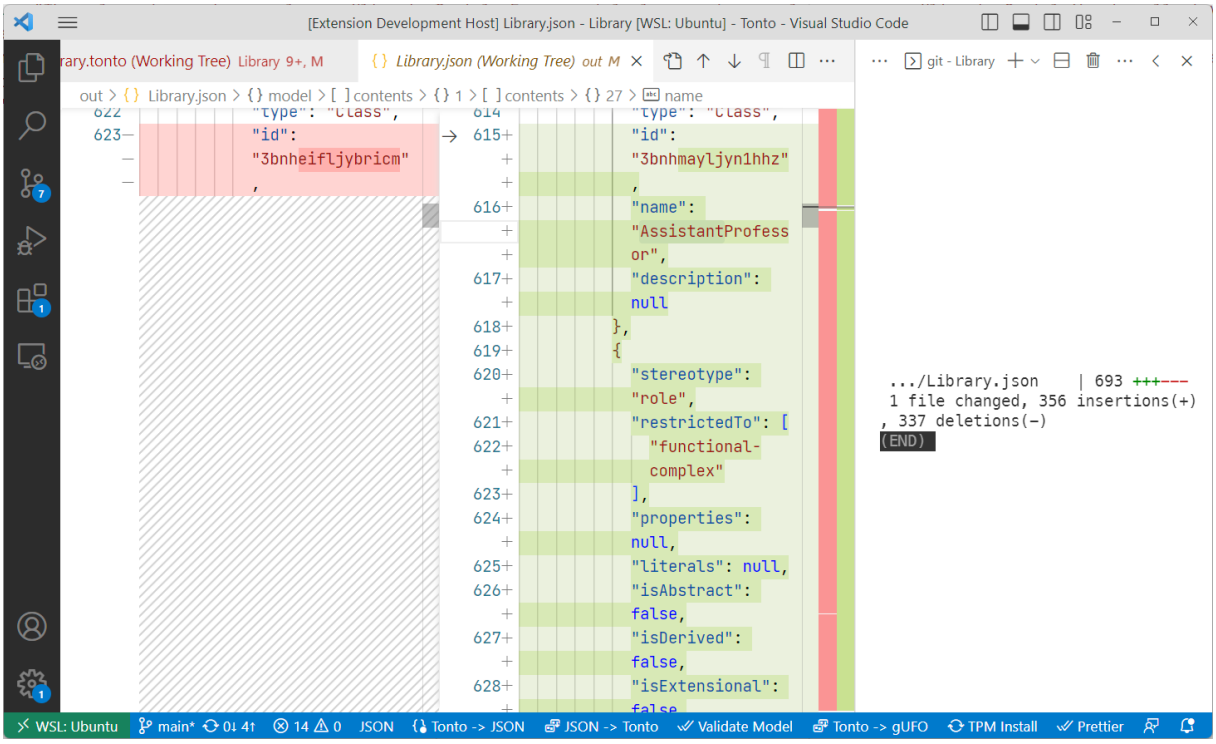

Figure 22 – VSCode showing the differences in a JSON file after adding the role *AssistantProfessor* to the Tonto file and using the transformation command.

On the other hand, version control in Visual Paradigm files has more problems. In Figure 23 we can verify that because .vpp files used by Visual Paradigm uses a specific format only for its tool, git cannot analyse its contents. In that way, its impossible to enable version control in it with git. However, Visual Paradigm has its own built-in approach to manage differences between models and diagrams. Because we cannot apply any other textual tool, this approach does not enable the same possibilities as Tonto and JSON files.

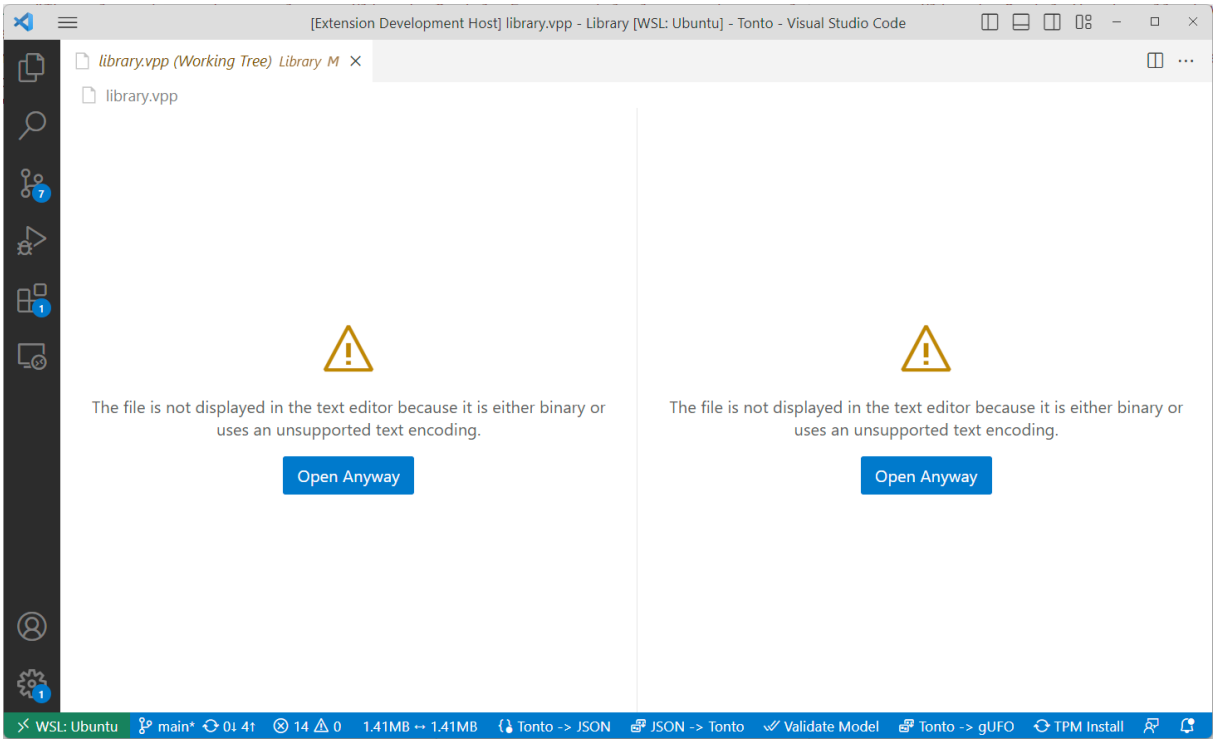

Figure 23 – VSCode showing that changes in a Visual Paradigm model and diagram cannot be versioned by git.

One possibility to version Visual Paradigm projects is using JSON files generated by the OntoUML plugin. Because ontouml-schema for JSON files holds information on the position of every element in the diagram, it's possible to deal with this versioning by importing and exporting JSON files. However, this approach also has some disadvantages, taking into consideration that every modification will generate changes in multiple lines of the JSON file. In files with thousands of lines, this can be confusing for someone to analyze and approve these changes. Figure 24 shows an example of the same modification, adding the role *AssistantProfessor* specializing Professor and then generating the JSON file from the modified version. Making some small adjustments in elements positions already generate many changes, as shown by the example, showing 1200 inserted lines and 1127 deletions. With a lot of non-structured changes from generated code, it is hard to deal for example with merge conflicts, a common necessity of projects with more than one developer. Considering that changes in a model could modify way more than one element, this approach is not as practical as Tonto.

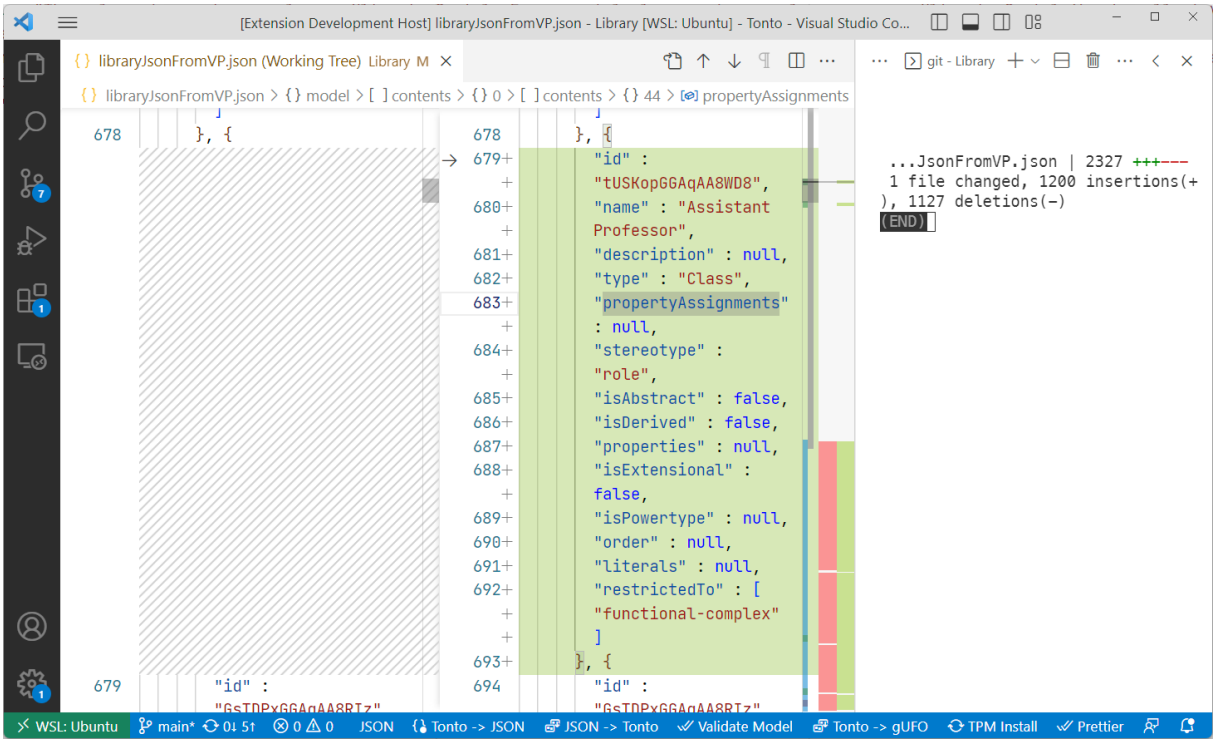

Figure 24 – VSCode showing changes in a JSON file generated by a modified diagram in Visual Paradigm.

5 Conclusion

In the following sections, the final considerations of this work will be presented, addressing its main contributions and the challenges encountered (Section 5.1). Then, some lessons learned with the tools used will be discussed (Section 5.2). Additionally, some suggestions for future work will be provided, aiming to bring new contributions to the accomplished project (Section 5.3).

5.1 Final Considerations

In this work, we have presented the foundation of Ontology-driven Conceptual Modeling and the OntoUML Language built as a UML extension based on the Unified Foundational Ontology (UFO). We observed in the literature the relevance of this theme and different approaches to enabling conceptual modeling for computer and information science fields. Some of these presented works have a textual syntax to represent ontology-based models, like OWL. This served as a theoretical basis for developing the language Tonto and all tools related to it: Tonto CLI, Tonto VS Code extension, and Tonto Package Manager. The main goal of this work was to create a textual syntax to enable the representation of OntoUML elements, focusing on developer experience with the VS Code extension and allowing transformation into other formats, e.g. gUFO-based OWL and JSON. Also, it allows modularization of projects, aiming to increase language use and better code organization.

After that, we presented the advantages of having a textual syntax and how it can benefit the development and maintainability of models, such as having better version control with git tools, making it easier to compare different versions of the same model, and being able to merge updates on a model easily.

With Tonto CLI, we showed how models created with Tonto could connect with already consolidated tools like ontouml-server, gUFO-based OWL, and OntoUML JSON representation that can be imported to the Visual Paradigm plugin.

With Tonto VS Code extension, users can utilize every feature of Tonto grammar powered by modern programming language features like smart code completion, jump-to-definition functions, error visualization, and commands easily accessed with buttons at the bottom bar.

With Tonto Package Manager, every Tonto project can take advantage of code reuse, enabling the development of foundational models for different areas that can be referenced in any project.

Lastly, we experimented Tonto toolkit on an existing model with a high amount of

elements, addressing differences between versioning control on all approaches. In conclusion, Tonto has a simpler way of changing the definition of models, allowing better readability and fewer conflicts on merging tools. In addition to that, it facilitates the scalability of projects, being able to represent many elements in an organized way without losing readability, and allowing code reuse with TPM.

However, textual syntaxes have some limitations, such as the lack of a more visual interface. The importance of these limitations is reduced when you consider all the features that can be integrated into textual environments to facilitate model development and visualization. Also, with the possibility of transformation between textual syntax and diagrams, we could have the advantages of all situations.

Finally, the development of this work required the knowledge acquired during years of study in the Computer Science course at UFES (Federal University of Espírito Santo), including, but not limited, to the knowledge of: object-oriented programming, programming languages and deep knowledge of parsers and compilers, web development, data structures, algorithms, software engineering and domain engineering. During this course, I was able to learn and practice each one of these topics, being able to masterfully project and create software. All this knowledge was essential to make the creation of Tonto possible, together with the theoretical foundation of ontology-based conceptual modeling.

5.2 Lessons Learned

One of the biggest challenges of building Tonto was dealing with Langium. Like a double-edged sword, Langium has many benefits and disadvantages associated with it. The first benefit is getting an out-of-the-box implementation of AST elements for every element of a defined grammar, with a generated LSP that allows the usage of this grammar in a VSCode extension. The second benefit is getting a basic implementation of a VS Code extension, allowing the usage of this grammar with code completion, basic syntax highlighting, the possibility of defining validators that run on each element of the grammar, and scope and cross-reference calculations already implemented. With pre-defined services, Langium allows the personalization of every functionality described above by the usage of the inheritance concept from Object-Oriented Programming, allowing many changes.

However, getting a lot of implementations at the beginning can make it more difficult to personalize when trying to do something more specific. One example was the challenge of implementing scope computation based on the imports of a project because Langium automatically considers every element in every file globally. Also, when implementing validators, because Tonto allows the specification of generalizations both directly on an element and with generalization sets, some validators needed to be created by scratch using validators at the top level of the model definition, creating more complexity. The last problem was dealing with

Langium updates during the development of this project. Because Langium was released in 2021, it had many breaking changes since then, improving core features and creating the need for refactoring code in order to update. This allowed Tonto to be more powerful as new features were added while creating a lot of rework in order to update. This was a natural consequence of early adoption.

Finally, the benefits of using Langium certainly outweigh the harms. Without it, all the processes of creating an AST from a Grammar and adding every functionality to it would be needed to be done by hand, and with time being a limiting factor, this project would not reach its current state without this library.

5.3 Future Work

- **Add more functions to Tonto Package Manager:** In its current state, the package manager recursively searches for inner dependencies of the dependencies on `tonto.json` manifest every time it makes an install. In order to optimize this process, it should be done only when necessary. One way of caching it is creating a *lock* file that will be created on the first install and updated after dependencies change. This file contains information about all the required dependencies and URLs, so it is easier to download them without recursively searching for every dependency.

- **Add a way to generate diagrams with Tonto:** One of the biggest problems of textual syntaxes is losing the advantage of visualizing every element of a model and its relation easily on the screen. With a tool that creates diagrams based on Tonto elements and code annotations on comments in these elements, we could have the advantages of both worlds. The idea is not to force the definition of diagram information in every Tonto model, but allowing the user to decide between declaring diagram information or not without losing agility in development.

- **Improve the compatibility of transformations:** On its current state, Tonto CLI does not allow 100% of elements from other formats to be converted into Tonto. One possibility is to ensure every element from OntoUML JSON format is created when importing it to Tonto and in the opposite direction.

- **Increse unit test coverage:** Unit tests are a great way of guaranteeing code quality and maintainability. Tonto grammar and validators already have great coverage of unit tests, but that can be increased on transformation functionalities.

- **Update project to ESM Modules:** ESM Modules (EcmaScript Modules) are the new official standard format to package JavaScript code for reuse, utilizing *import* and *export* syntax. The support for it is increasing and when a library supports it, this project need to support it as well. One of the features on the Langium 2.0 roadmap is to move completely

to ESM, and that change would need to be done on Tonto in order to follow Langium updates.

- **Experiment the Tonto framework:** In order to evaluate more of Tonto capabilities, a good experiment would be to process a large number of OntoUML models presently available in JSON serialization. These models should be transformed into Tonto and evaluated for the number of elements that were correctly translated. This would also allow a comparison into how Tonto validators perform when contrasted with the ontouml-server validation API. Lastly, one could also evaluate the Tonto to JSON transformation.

Bibliography

ALMEIDA, J. P. A.; FALBO, R. A.; GUIZZARDI, G. Events as entities in ontology-driven conceptual modeling. In: . Cham: Springer, 2019. (LNCS, v. 11788), p. 469–483. Disponível em: <https://dx.doi.org/10.1007/978-3-030-33223-5_39>. Cited 3 times on pages 9, 21, and 26.

ALMEIDA, J. P. A. et al. *gUFO: A Lightweight Implementation of the Unified Foundational Ontology (UFO)*. 2019. Disponível em: <http://purl.org/nemo/doc/gufo>. Cited on page 17.

ALMEIDA, J. P. A.; FONSECA, C. M.; CARVALHO, V. A. A comprehensive formal theory for multi-level conceptual modeling. In: MAYR, H. C. et al. (Ed.). *Conceptual Modeling*. Cham: Springer International Publishing, 2017. p. 280–294. ISBN 978-3-319-69904-2. Cited on page 27.

BENEVIDES, A. B. et al. Validating modal aspects of ontouml conceptual models using automatically generated visual world structures. *J. UCS*, v. 16, p. 2904–2933, 2 2010. Cited on page 14.

BRAGA, B. F. B. et al. Transforming ontouml into alloy: towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering*, v. 6, p. 55–63, 2010. ISSN 1614-5054. Disponível em: <https://doi.org/10.1007/s11334-009-0120-5>. Cited on page 14.

CAMBRIDGE. *Cambridge Dictionary*. 2023. Disponível em: <https://dictionary.cambridge.org/>. Cited on page 19.

FONSECA, C. M. *ML2: An Expressive Multi-Level Conceptual Modeling Language*. 2017. M.Sc. thesis, Federal University of Espírito Santo, Brazil. Disponível em: <https://dx.doi.org/10.13140/RG.2.2.16142.00327>. Cited 3 times on pages 17, 30, and 36.

FONSECA, C. M. et al. Incorporating types of types in ontology-driven conceptual modeling. In: *Conceptual Modeling - 41st International Conference, ER 2022, Hyderabad, India, October 17-20, 2022, Proceedings*. Springer, 2022. (Lecture Notes in Computer Science, v. 13607), p. 18–34. Disponível em: <https://doi.org/10.1007/978-3-031-17995-2_2>. Cited 2 times on pages 9 and 27.

FONSECA, C. M. et al. Ontology-driven conceptual modeling as a service. In: *JOWO 2021 The Joint Ontology Workshops*. CEUR-WS, 2021. Disponível em: <https://ceur-ws.org/Vol-2969/paper29-FOMI.pdf>. Cited 3 times on pages 14, 17, and 28.

GUIDONI, G.; ALMEIDA, J.; GUIZZARDI, G. Transformation of ontology-based conceptual models into relational schemas. In: *39th International Conference on Conceptual Modeling (ER 2020)*. [S.l.]: Springer, 2020. p. 315–330. ISBN 978-3-030-62522-1. Cited on page 14.

GUIZZARDI, G. *Ontological foundations for structural conceptual models*. Enschede, The Netherlands: Centre for Telematics and Information Technology, 2005. ISBN 9075176813. Cited 6 times on pages 14, 16, 19, 20, 21, and 22.

GUIZZARDI, G. et al. Ufo: Unified foundational ontology. *Applied Ontology*, IOS Press BV, v. 17, p. 167–210, 2022. ISSN 18758533. Cited 9 times on pages 9, 14, 16, 20, 21, 22, 23, 24, and 25.

GUIZZARDI, G. et al. Endurant types in ontology-driven conceptual modeling: Towards ontouml 2.0. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, v. 11157 LNCS, p. 136–150, 2018. ISSN 16113349. Cited 4 times on pages 14, 21, 22, and 25.

GUIZZARDI, G. et al. Towards ontological foundations for the conceptual modeling of events. In: *International Conference on Conceptual Modeling (ER 2013)*. Springer, 2013. (Lecture Notes in Computer Science, v. 8217), p. 327–341. Disponível em: <https://dx.doi.org/10.1007/978-3-642-41924-9_27>. Cited 3 times on pages 20, 21, and 26.

JACKSON, D. *Software abstractions: Logic, Language, and Analysis*. [S.l.]: MIT Press, 2016. Cited on page 17.

MOREIRA, J. et al. Menthor editor: an ontology-driven conceptual modeling platform. In: *JOWO 2016 The Joint Ontology Workshops*. [s.n.], 2016. Disponível em: <https://ceur-ws.org/Vol-1660/demo-paper1.pdf>. Cited on page 14.

OMG. *OMG Unified Modeling Language (OMG UML) Version 2.5.1*. 2017. Disponível em: <http://www.omg.org/spec/UML/2.5.1>. Cited on page 14.

SALES, T. P.; GUIZZARDI, G. Ontological anti-patterns: Empirically uncovered error-prone structures in ontology-driven conceptual models. *Data & Knowledge Engineering*, v. 99, 2 2015. Cited on page 14.

W3C. *OWL*. 2012. Disponível em: <https://www.w3.org/OWL/>. Cited 2 times on pages 17 and 20.

W3C. *Turtle*. 2014. Disponível em: <https://www.w3.org/TR/turtle/>. Cited 2 times on pages 17 and 20.

Appendix

APPENDIX A – Tonto Grammar

This appendix presents a specification of Tonto concrete syntax in langium, using its grammar similar to EBNF.

Listing A.1 – Tonto Grammar defined in Langium EBNF form.

```
grammar Tonto

Model:
  imports+=Import*
  module=ContextModule;

ContextModule:
  (isGlobal?='global')? 'package' (name=QualifiedName | name=
     STRING)
  declarations+=Declaration*;

QualifiedName returns string:
  ID ('.' ID)*;

Import: 'import' referencedModel=[ContextModule:QualifiedName
   ] ('as' packageAlias=ID)?;

Declaration: ClassDeclaration | AuxiliaryDeclaration;

AuxiliaryDeclaration: DataType | Enum | GeneralizationSetImpl
    | ExternalRelation;

ClassDeclaration: classElementType=OntologicalCategory name=
   QualifiedName ontologicalNatures=ElementOntologicalNature?
    ('(' 'instanceOf' instanceOf=[ClassDeclaration:
       QualifiedName] ')')?
    ('specializes' specializationEndurants+=[ClassDeclaration
       :QualifiedName]
    (',' specializationEndurants+=[ClassDeclaration:
       QualifiedName])?
    )?
```

```
25      ('{' (attributes+=Attribute | references+=
            InternalRelation)*'}')?;
26
27  OntologicalCategory: ontologicalCategory=(UnspecifiedType |
        NonEndurantType | EndurantType);
28
29  UnspecifiedType returns string: 'class';
30  NonEndurantType returns string: 'event' | 'situation';
31  EndurantType returns string: NonSortal | UltimateSortal |
        Sortal;
32  NonSortal returns string:
33    'category' | 'mixin' | 'phaseMixin' | 'roleMixin' | '
          historicalRoleMixin';
34  UltimateSortal returns string: 'kind' | 'collective' | '
        quantity' | 'quality' | 'mode' | 'intrinsicMode' | '
        extrinsicMode' | 'relator' | 'type' | 'powertype';
35  Sortal returns string: 'subkind' | 'phase' | 'role' | '
        historicalRole';
36
37  ElementOntologicalNature: 'of' natures+=OntologicalNature (',
        ' natures+=OntologicalNature)*;
38
39  OntologicalNature returns string: 'objects' | 'functional-
        complexes' | 'collectives'  | 'quantities' |
40      'relators' | 'intrinsic-modes' | 'extrinsic-modes' | '
            qualities' |
41      'events' | 'situations' |  'types' | 'abstract-
            individuals';
42
43  ElementRelation: InternalRelation | ExternalRelation;
44
45  InternalRelation infers ElementRelation:
46  ('@'relationType=RelationStereotype)?
47  (
48      '('
49          ('{' metaAttributes+=RelationMetaAttribute
50          (',' metaAttributes+=RelationMetaAttribute )* '}')?
51          (firstEndName=ID)?
52      ')'
```

```
53 )?
54 firstCardinality=Cardinality?
55 (isAssociation?='--' | isAggregation?='<>--' | isComposition
      ?='<o>--') (name=QualifiedName '--')?
56 secondCardinality=Cardinality?
57 (
58     '('
59         ('{' secondEndMetaAttributes+=RelationMetaAttribute
60         (',' secondEndMetaAttributes+=RelationMetaAttribute )
              * '}')?
61         (secondEndName=ID)?
62     ')'
63 )?
64 secondEnd=[ClassDeclaration:ID]
65 ('specializes' specializeRelation=[ElementRelation:
      QualifiedName])?
66 (hasInverse='inverseOf' inverseEnd=[ElementRelation:
      QualifiedName])?;
67
68 ExternalRelation infers ElementRelation:
69   ('@'relationType=RelationStereotype)?
70   'relation'
71     firstEnd=[ClassDeclaration:QualifiedName]
72   (
73     '('
74     ('{' firstEndMetaAttributes+=RelationMetaAttribute
75     (',' firstEndMetaAttributes+=RelationMetaAttribute )* '}'
         )?
76     (firstEndName=ID)?
77     ')'
78   )?
79
80   firstCardinality=Cardinality?
81   (isAssociation?='--' | isComposition?='<>--' |
        isComposition?='<o>--') (name=QualifiedName '--')?
82
83   secondCardinality=Cardinality?
84   (
85     '('
```

```
86        ('{' secondEndMetaAttributes+=RelationMetaAttribute
87        (',' secondEndMetaAttributes+=RelationMetaAttribute )*
             '}')?
88        (secondEndName=ID)?
89      ')'
90    )?
91
92    secondEnd=[ClassDeclaration:ID]
93    ('specializes' specializeRelation=[ElementRelation:
         QualifiedName])?
94    (hasInverse='inverseOf' inverseEnd=[ElementRelation:
         QualifiedName])?
95 ;
96
97 Attribute:
98    name=ID ':' attributeTypeRef=[DataType:QualifiedName]
99    cardinality=Cardinality?
100   ('{'((isOrdered?='ordered') & (isConst?='const') & (
         isDerived?='derived'))? '}')?;
101
102 RelationMetaAttribute:
103   isOrdered?='ordered' | isConst?='const' | isDerived?='
         derived' |
104   ('subsets' subsetRelation=[ElementRelation:QualifiedName] )
          |
105   ('redefines' redefinesRelation=[ElementRelation:
         QualifiedName] );
106
107 RelationStereotype returns string:
108   'material' |
109   'derivation' |
110   'comparative' |
111   'mediation' |
112   'characterization' |
113   'externalDependence' |
114   'componentOf' |
115   'memberOf' |
116   'subCollectionOf' |
117   'subQuantityOf' |
```

```
118   'instantiation' |
119   'termination' |
120   'participational' |
121   'participation' |
122   'historicalDependence' |
123   'creation' |
124   'manifestation' |
125   'bringsAbout' |
126   'triggers' |
127   'composition' |
128   'aggregation' |
129   'inherence' |
130   'value' |
131   'formal';
132
133 Cardinality:
134   '[' lowerBound=(INT | '*')
135   ('..' upperBound=(INT | '*'))? ']';
136
137 GeneralizationSet:
138   (disjoint?='disjoint')? (complete?='complete')?
139   'genset' name=ID '{'
140       (
141         'general' generalItem=[ClassDeclarationOrRelation:
              QualifiedName]
142         ('categorizer' categorizerItems+=[
              ClassDeclarationOrRelation:QualifiedName])?
143         'specifics' specificItems+=[
              ClassDeclarationOrRelation:QualifiedName]
144         (',' specificItems+=[ClassDeclarationOrRelation:
              QualifiedName])*
145       )
146   '}';
147
148 GeneralizationSetShort returns GeneralizationSet:
149   (disjoint?='disjoint')? (complete?='complete')?
150   'genset' name=ID 'where'
151   specificItems+=[ClassDeclarationOrRelation:QualifiedName]
        (',' specificItems+=[ClassDeclarationOrRelation:
```

```
      QualifiedName])*
152   'specializes' generalItem=[ClassDeclarationOrRelation:
      QualifiedName]
153 ;
154
155 type ClassDeclarationOrRelation = ClassDeclaration |
    ElementRelation;
156
157 // <--- DataTypes --->
158 type DataTypeOrClass = DataType | ClassDeclaration;
159
160 DataType:
161   'datatype' name=ID ontologicalNature=
      ElementOntologicalNature?
162   ('specializes' specializationEndurants+=[DataTypeOrClass:
      QualifiedName]
163   (',' specializationEndurants+=[DataTypeOrClass:
      QualifiedName])?
164   )?
165   ('{'
166     (attributes+=Attribute)*
167   '}')?;
168
169 // <--- Enums --->
170 Enum infers DataType:
171   isEnum?='enum' name=ID
172   ('specializes' specializationEndurants+=[DataTypeOrClass:
      QualifiedName]
173   (',' specializationEndurants+=[DataTypeOrClass:
      QualifiedName])?
174   )?
175   '{'
176     (elements+=EnumElement
177     ((',') elements+=EnumElement)*)?
178   '}';
179
180 EnumElement: name=ID;
181
182 hidden terminal WS: /\s+/;
```

```
terminal ID: /[_a-zA-Z][\w_]*/;
terminal INT returns number: /[0-9]+/;
terminal STRING: /"[^"]*"|'[^']*'/;

hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

APPENDIX B – University model in Tonto

This appendix presents the University model in Tonto

Listing B.1 – University package from University model in Tonto.

```
import CoreDatatypes
import PersonPhases
package University

category Organization

kind University specializes Organization {
  address: Address
  @componentOf
  [1] <>-- has -- [1..*] Department
}

kind Department {
  name: string
  @componentOf
  [1] <>-- [1] JuniorStaff
  @componentOf
  [1] <>-- [1] SeniorStaff
}

collective Staff {
  [1] <o>-- hasMember -- [0..*] Employee
}

subkind JuniorStaff specializes Staff
subkind SeniorStaff specializes Staff

roleMixin Employer

role UniversityEmployer specializes Employer, University

relator EmploymentContract {
  @mediation
```

```
34   [1..*] -- [1] Employee
35   @mediation
36   [1..*] -- [1] Employer
37 }
```

Listing B.2 – PersonPhases package from University model in Tonto.

```
1  import CoreDatatypes
2  import University
3
4  package PersonPhases
5
6  kind Person {
7    name: string [1..*]
8    age: number [1]
9    birthDate: date [1] { const }
10   eyeColor: EyeColor [1..*]
11   phoneNumber: CoreDatatypes.PhoneNumber [0..*]
12 }
13
14 type PersonTypeByAge
15
16 phase Child (instanceOf PersonTypeByAge)
17 phase Teenager (instanceOf PersonTypeByAge)
18 phase Adult (instanceOf PersonTypeByAge)
19
20 genset PhasesOfPerson {
21   general Person
22   categorizer PersonTypeByAge
23   specifics Child, Teenager, Adult
24 }
25
26 role UniversityStudent specializes Person
27 role FormerStudent specializes UniversityStudent
28 role ActiveStudent specializes UniversityStudent
29
30 role Employee specializes Person
31 role UniversityProfessor specializes Employee
```

APPENDIX C – Library model in Tonto

This appendix presents the Library model transformed from JSON to Tonto with a few changes

Listing C.1 – Library model in Tonto.

```
package Library

relator Reserved_Copy {
    @mediation
    [1] -- [1] ( {const } ) Item
    @mediation
    [1] -- [1] ( {const } ) Copy
}
relator Renewed_Copy {
    @mediation
    [1] -- [1] ( {const } ) Item
    @mediation
    [1] -- [1] ( {const } ) Copy
}
relator Get_Copy {
    @mediation
    [1] -- [1] ( {const } ) Item
    @mediation
    [1] -- [1] ( {const } ) Items_Copy
}
kind Item {
    @formal
    [1] -- [1]  Work
}
role Items_Copy {
    @mediation
    [1] -- [1] ( {const } ) Copy
}
kind Work
subkind Videotape  specializes Work{
    @memberOf
    [1]  --<o> [1]  Collection
```

```
}
subkind Periodical  specializes Work{
    @memberOf
    [1]  --<o> [1]  Collection
}
subkind Book  specializes Work{
    @memberOf
    [1]  --<o> [1]  Collection
}
subkind Dvd  specializes Work{
    @memberOf
    [1]  --<o> [1]  Collection
}
collective Collection
kind Copy {
    @mediation
    [1] -- [1] ( {const } ) Copy_Reservation
    @mediation
    [1] -- [1] ( {const } ) Renew_Copy
    @mediation
    [1] -- [1] ( {const } ) Reserve_Copy_for_Employee
    @mediation
    [1] -- [1] ( {const } ) Employee_Loan_Role
    @mediation
    [1] -- [1] ( {const } ) Renew_Copy_for_Employee
}
role Copy_Reservation {
    @mediation
    [1] -- [1] ( {const } ) Make_Reservation_for_Student
}
relator Student_Loan {
    @mediation
    [1] -- [1] ( {const } ) Copy
    @mediation
    [1] -- [1] ( {const } ) Delay
    @mediation
    [1] -- [1] ( {const } ) Student
}
role Delay {
```

```
72      @mediation
73      [1] -- [1] ( {const } ) Generate_Student_Delay
74  }
75  relator Make_Reservation_for_Student {
76      @mediation
77      [1] -- [1] ( {const } ) Student
78  }
79  relator Renew_Copy_for_Student {
80      @mediation
81      [1] -- [1] ( {const } ) Copy
82      @mediation
83      [1] -- [1] ( {const } ) Student
84  }
85  role Renew_Copy {
86      @mediation
87      [1] -- [1] ( {const } ) Student
88  }
89  role Student  specializes Person
90  relator Generate_Student_Delay {
91      @mediation
92      [1] -- [1] ( {const } ) Return_Deadline
93      @mediation
94      [1] -- [1] ( {const } ) Student
95  }
96  kind Return_Deadline {
97      [*] -- [*]  Fine
98  }
99  kind Fine
100 phase Daily  specializes Fine
101 phase Monthly  specializes Fine
102 role Undergraduate_Student  specializes Student{
103     @mediation
104     ( {const } )[1] -- [1] ( {const } )
          Undergraduate_Contract
105 }
106 role Graduate_Student  specializes Student{
107     @mediation
108     ( {const } )[1] -- [1] ( {const } ) Graduate_Contract
109 }
```

```
relator Undergraduate_Contract {
    @mediation
    [1] -- [1] ( {const } ) UNISAM
}
relator Graduate_Contract {
    @mediation
    [1] -- [1] ( {const } ) UNISAM
}
kind UNISAM
kind Person
role Employee  specializes Person
role Administrative  specializes Employee
role Professor  specializes Employee
role Regular  specializes Administrative{
    @material
    [1] -- [1]  Intern
}
role Intern  specializes Administrative
relator Employee_Contract {
    @mediation
    [1] -- [1] ( {const } ) Regular
    @mediation
    [1] -- [1] ( {const } ) Intern
}
relator Professor_Contract {
    @mediation
    [1] -- [1] ( {const } ) UNISAM
    @mediation
    [1] -- [1] ( {const } ) Professor
}
relator Employee_Loan {
    @mediation
    [1] -- [1] ( {const } ) Employee
}
relator Renew_Employee {
    @mediation
    [1] -- [1] ( {const } ) Employee
}
relator Make_Reservation_for_Employee {
```

```
149     @mediation
150     [1] -- [1] ( {const } ) Employee
151 }
152 role Renew_Copy_for_Employee {
153     @mediation
154     [1] -- [1] ( {const } ) Renew_Employee
155 }
156 role Reserve_Copy_for_Employee {
157     @mediation
158     [1] -- [1] ( {const } ) Make_Reservation_for_Employee
159 }
160 role Employee_Loan_Role {
161     @mediation
162     [1] -- [1] ( {const } ) Employee_Loan
163 }
```

APPENDIX D – Library model diagrams

This appendix presents the Library model diagrams. First, the file from the original model, and then the model generated by exporting from Tonto.

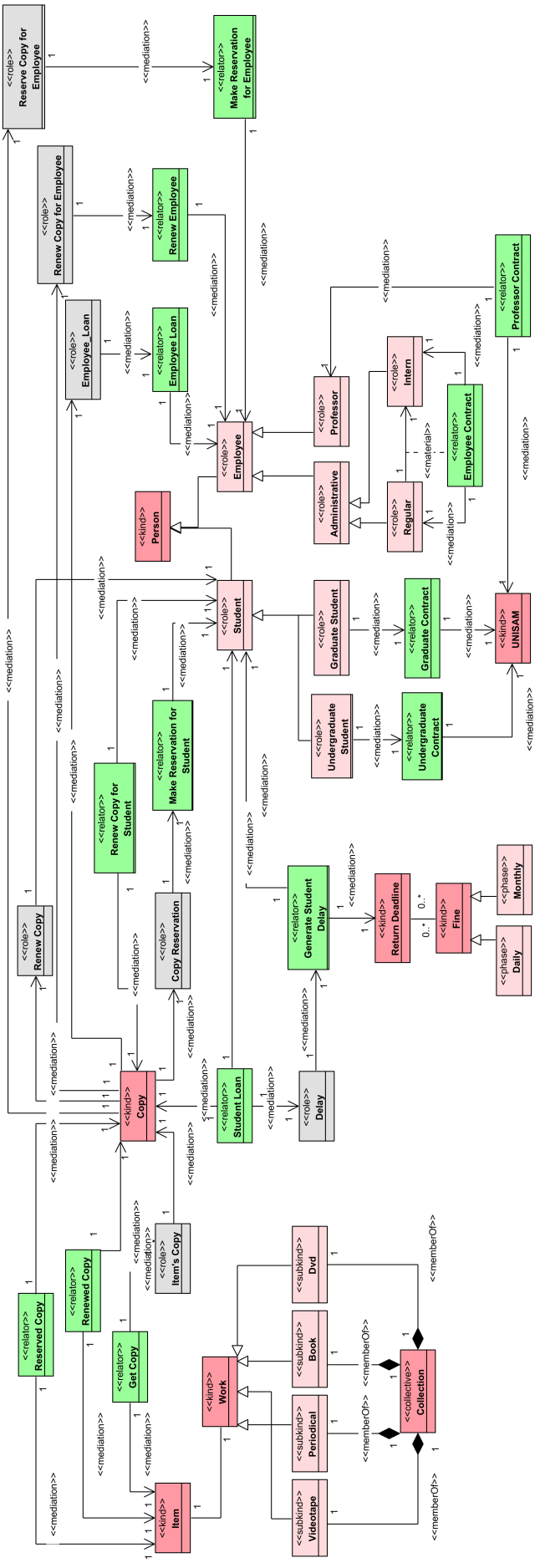

Figure 25 – Library model created using OntoUML

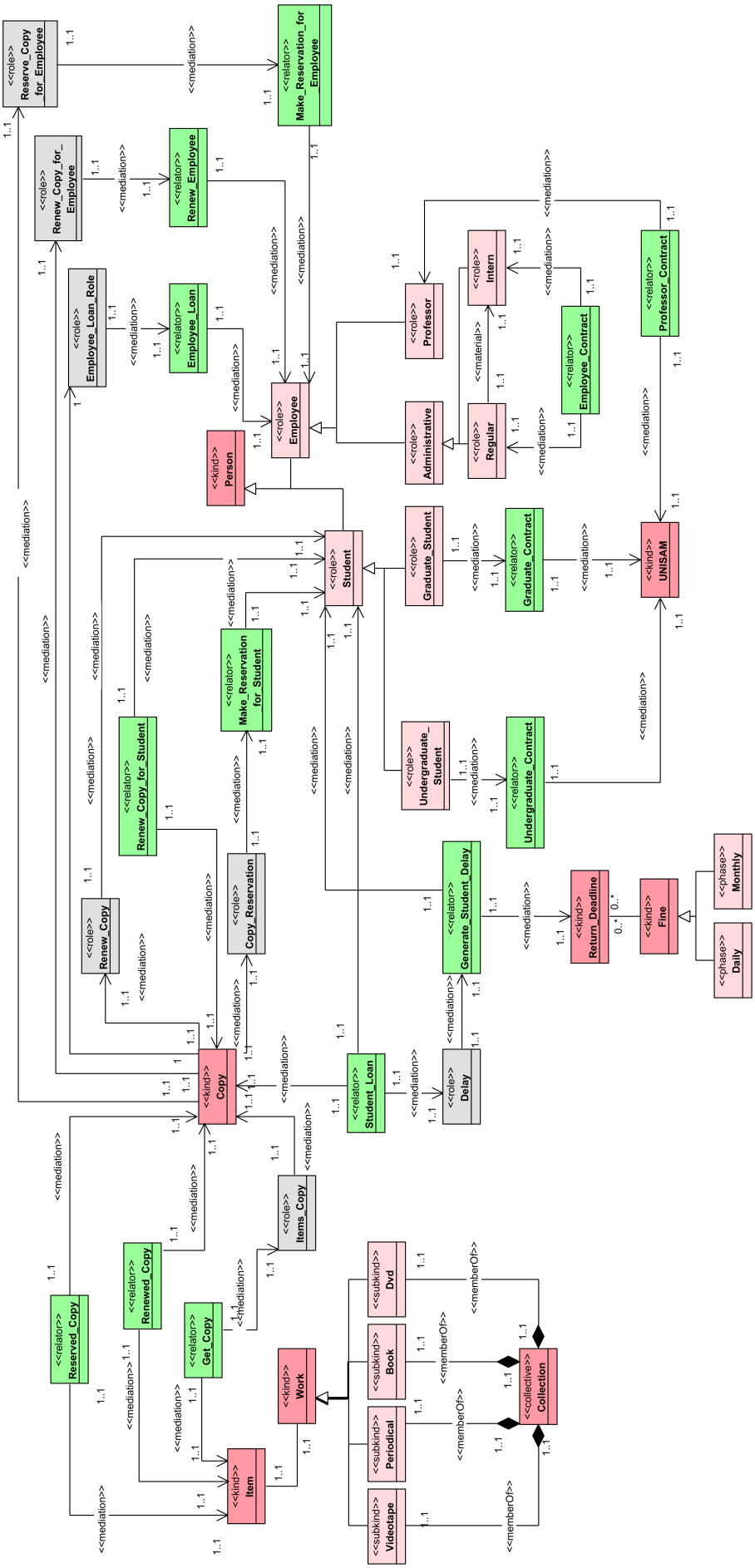

Figure 26 – Library model diagram imported from a JSON file generated from Tonto