Formal Ontology in Information Systems
C. Trojahn et al. (Eds.)

doi:10.3233/FAIA241308

A Textual Syntax and Toolset for Well-Founded Ontologies

Matheus L. COUTINHO [a,1], João Paulo A. ALMEIDA [a], Tiago Prince SALES [b] and Giancarlo GUIZZARDI [b]

[a] *Ontology & Conceptual Modeling Research Group (NEMO), Federal University of Espírito Santo (UFES), Brazil*
[b] *Semantics, Cybersecurity and Services (SCS), University of Twente, The Netherlands*

Abstract. Diagrammatic and textual languages differ significantly with respect to the experience they offer to language users. While diagrammatic languages leverage visual variables to improve communication and problem solving, textual languages facilitate significantly a number of tasks including version control, model editing, model merging, parsing, etc. In this paper, we explore the design of a textual language for UFO-based ontologies, whose constructs mirror those of the OntoUML language. The language is supported by a rich VS Code-based editor, supporting (semantically-motivated) syntax verification, syntax highlight, autocomplete, and full integration into the OntoUML server ecosystem. A package manager is also offered to support ontology modularization and reuse, drawing inspiration from software package managers. Such functionality is currently not available to languages such as (Onto)UML and Semantic Web languages such as OWL.

Keywords. Textual Language, Ontology Development, UFO, OntoUML

1. Introduction

In recent decades, there has been a growing interest in using foundational ontologies in conceptual modeling and ontology engineering, with these ontologies being used in the revision of conceptual modeling languages and in the creation of ontology-driven modeling languages. A notable example in this domain is OntoUML [1,2], which was developed by incorporating the distinctions underlying the Unified Foundational Ontology (UFO) [2,3] in the UML class diagrams. This has leveraged a foundation for conceptual modeling constructs, which reflects the various theories from linguistics, cognitive science, and formal philosophical ontology used in the development of UFO.

OntoUML introduces various stereotypes that correspond to the concepts defined in UFO as well as grammatical formal constraints that reflect UFO's axiomatization. A key goal for the language was to support the definition of high-quality well-founded reference ontologies. Over the years, sophisticated tooling has been developed for OntoUML, including functionalities for: (i) editing and syntactic verification of models to conform with UFO's axioms [3,4]; (ii) model simulation, respecting the modal aspects of UFO [5];

[1] Corresponding Author: Matheus L. Coutinho, matheuslenke@gmail.com.

(iii) automatic generation of database schemas guided by UFO metaproperties [6]; (iv) detection of anti-patterns [7], among others.

OntoUML is an extension of UML class diagrams (technically, a UML profile) and, hence, primarily a diagrammatic/visual language for modeling ontologies. This provides significant benefits for communication among ontologists, as well as (visual) problem-solving. However, despite these advantages, there are also drawbacks associated with exclusively visual representations, e.g., the effort invested in diagram-layouting tasks, the difficulty of dealing with large diagrams, and the inability to apply mature 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, auto-complete, and more. Unfortunately, the infrastructure that has matured over the years for text-based languages cannot be applied directly to diagrammatic languages. Furthermore, and in line with the Dual-Channel Processing theory [8], there are clear benefits to communication, learning and problem-solving when diagrammatic and textual modalities are employed in tandem to represent domain information. These benefits come from leveraging complementary cognitive processes, as well as in the active process of paying attention to different modalities and mentally integrating them into coherent representations.

The benefits of a textual notation for ontologies have motivated a number of developments in this area, including textual notations for Semantic Web ontologies, such as the Turtle [9] and XML [10] serializations of OWL [11], and the more recent OML [12] language. However, as discussed in [2], despite the name ("Web Ontology Language"), OWL is to a large extent *ontologically neutral*. OML, in its turn, while largely based on OWL, introduces a distinction between sortal types and non-sortal types [2] (which in OML are called 'concept' and 'aspect', respectively). In any case, unlike OntoUML, these languages do not make a full commitment to a foundational ontology and, hence, do not systematically reap the benefits of reflecting the distinctions and guidelines put forth by one. To address this gap, we propose here a textual syntax for ontologies based on UFO, and integrated with the tools developed in the OntoUML ecosystem. The language is termed Tonto (for **T**extual **onto**logies) and draws inspiration from the functionality support typically offered to programmers in professional coding platforms, including: (semantically-motivated) syntax highlighting and verification, auto-complete, code snippets, as well as package management supporting modularization and reuse.

The Tonto grammar reflects in its constructs the notions of UFO's taxonomy of types, thus allowing language users to establish the meta-properties of types in an ontology (including rigidity, sortality, external dependence, etc.) [3]. The grammar also reflects the notions of UFO's taxonomy of individuals, allowing language users to distinguish between types of objects, of events, of relators, etc. Its formal syntactic constraints reflecting UFO axioms, support the automatic identification of ontological mistakes.

This grammar was then used to generate a parser and an editor in the form of a VS Code extension. VS Code was selected because of its extensibility, lightweightness and ample user base. The Tonto VS Code extension provides a rich ontology editing experience supporting real-time (semantically-motivated) syntax verification, syntax highlighting, autocomplete, but also native transformation to OntoUML (JSON serialization). Integration with the OntoUML server enables a number of additional services, including OWL transformation (based on gUFO [13]), anti-pattern detection, model validation via visual simulation, code generation, etc. [4]. Finally, a package manager is offered for Tonto, inspired by popular software development package managers, to support the

modularization of projects, thereby enhancing ontology reuse and organization.

This paper is further structured as follows. Section 2 presents our baseline: the diagrammatic OntoUML language, including a running example we use throughout the paper to present Tonto; Section 3 presents Tonto's requirements, while Section 4 elaborates on its main design choices and introduces the grammar of the language; Section 5 presents the Tonto infrastructure and discusses our validation efforts for it, including the preparation of a catalog with 168 Tonto specifications; Section 6 compares Tonto with other textual languages for ontologies; Section 7 presents conclusions and future work.

2. Research Baseline: OntoUML

In what follows, we make a whirlwind tour on UFO's distinctions reflected in OntoUML. For a full presentation on the latter including its grammar, formal semantics, applications, empirical support, and its formal connection to UFO, one should refer to [2,3].

Individuals in UFO can be either *events* (unfolding in time), *situations* (parts of the world that can be understood as whole) or *endurants*. Endurants (or continuants) can be *objects* or *aspects* (existentially dependent endurants). Aspects can be either existentially dependent on one single bearer (*qualities* and *modes*) or on multiple individuals (*relators*). Qualities are aspects that take their values in certain *quality structures* (conceptual spaces). Modes (e.g., dispositions) can be externally dependent on entities other than their bearers (extrinsic modes). Within the space of endurant types, we have *sortals* and *non-sortals*. Sortals are either *substance sortals* (i.e., the types that provide a uniform principle of identity and individuation for their instances) or specializations thereof (*subkinds*, *phases* and *(historical) roles*). Substance sortals are either called *kinds* simpliciter (when their instances are functional complexes – i.e., wholes mereologically structured into functional components), *quantity* (when their instances are amounts of matter), or *collectives* (when their instances are mereologically structured into a uniform membership). Non-sortals are types that collect instances of different substance sortals. They are either *categories*, *phase mixins*, *(historical) role mixins*, or *mixin* simpliciter. Substance sortals, subkinds and categories are rigid (they are necessarily instantiated by their instances); phases, (historical) roles (mixins), and phase mixins are anti-rigid (they are contingently instantiated by their instances); mixins are semi-rigid (necessarily instantiated by some of their instances but contingently by others). A historical role is a role played in the scope of an event by instances of a unique substance sortal (historical role mixin when these instances can come from different substance sortals). Higher-order types (including *powertypes*) are types whose instances are also types.

Figure 1 contains a fragment of an OntoUML model for a domain, which is used as a running example throughout this paper. For example, the class Person is stereotyped «kind». The classes Child, Teenager, and Adult are «phase»s a person goes through in life. These phases of Person are instance of the higher-order type PersonTypeByAge. Also, we have the role Employee that a person can play in the scope of a relationship (an instance of the «relator» class Employment Contract) with an Employer, and hence this class is marked with the stereotype «role». Employer is a role mixin whose instances can be of several kinds (e.g., Universities, Hospitals, Military Organizations). A University is a type of functional complex (hence, stereotyped as «kind») that has as components Departments. Departments are constituted by instances of Staff («collective»), which are

structured in sub-collectives (e.g., Senior Staff), both of which have members that are Employees. Qualities are typically represented in OntoUML via attributes, i.e., functions mapping instances of the associated types to Datatypes (representing quality structures).

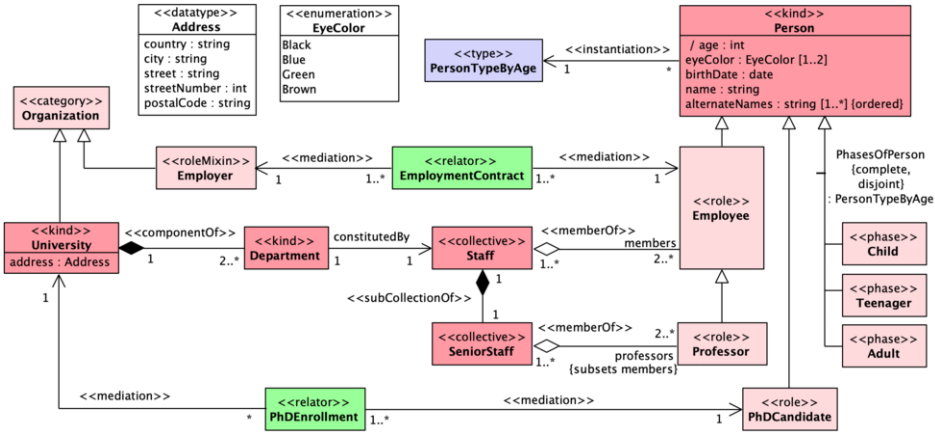

Figure 1. Diagram describing a University with Person phases and roles

3. Requirements

In the sequel, we list the requirements driving the design of Tonto:

R1 – It should cover UFO [3] (and corresponding OntoUML constructs), including the various types of classes with their corresponding metaproperites; it should support the various types of relations accounted for in UFO (such as inherence, mediation, whole-part relations); it should support the specification of the ontological natures of instances of classes (distinguishing classes of objects, of events, of aspects, etc.), as well as the specification of high-order types [14].

R2 – It should support the whole range of constructs that are usually offered in structural conceptual modeling languages (such as UML), including cardinalities, generalization sets, specializations, attributes, datatypes and enumerations.

R3 – It should embody rules that conduce to the production of sound models, in conformity with the axiomatization of UFO [3].

R4 – It should employ a textual syntax that is familiar to users of mainstream object-oriented programming languages (such as Java, Typescript and Swift) and UML.

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

R5 – The tool should assist the modeler in discovering model errors with (real-time) syntax verification based on OntoUML/UFO rules.

R6 – The tool should assist the modeler in producing syntactically correct Tonto models with features such as code snippets and auto-complete.

R7 – The tool should be interoperable with the OntoUML ecosystems by importing/exporting Tonto models to JSON in compliance with `ontouml-schema`[2].

R8 – The tool should enable transparent access to the ontouml-server [4] functionalities, leveraging these functionalities available for OntoUML models also for Tonto models (including generation of OWL implementations).

R9 – The tool should support the modularization of Tonto-based projects, leveraging dependency management that is widely adopted in software engineering (and embodied in automated build management tools such as Maven, Gradle, NPM, etc.)

These requirements for Tonto and its tooling support highlight the focus on creating a practical, user-friendly tool that addresses the limitations of existing ontology-driven modeling languages. By fulfilling these requirements, Tonto aims to enhance the productivity of modelers, improve model management, and foster a more collaborative and efficient modeling environment.

4. The Tonto Language

In this section, we present the various elements of the language[3] to address the aforementioned requirements, including Tonto code fragments corresponding to our running example. The elements are declared in Tonto specifications, i.e., textual files with the `.tonto` extension.

4.1. Package Declarations

The first declaration in a Tonto specification is equivalent to the definition of a *package* in UML/OntoUML, using the keyword `package` (see line 1 of Listing 1). It establishes that all the declarations within a file belong to the named package at the top of a file. Packages define naming spaces to cope with naming conflicts, and are the basis for modularity mechanisms in the Tonto package manager. 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. Auxiliary declarations include datatypes, enumerations, generalization sets and associations (when defined outside the body of class declarations).

4.2. Class declarations

Every *class declaration* 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 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 (in which case, a Tonto editor should offer a *warning* to the user).

[2] https://purl.org/ontouml-schema

[3] The Tonto EBNF specification can be found at https://w3id.org/tonto/ebnf.

Listing 1 shows a fragment of our example model in Tonto. It employs the keywords `kind`, `phase`, and `role` to define the model's classes. To capture specialization between classes, Tonto utilizes the `specializes` keyword following the identifiers of the superclass(es).

The listing also shows the syntax for attributes, using the builtin datatypes `number` and `string`. Cardinalities are indicated between square brackets, in this case exactly one for `name` and `age` (the default cardinality), and one or more for `alternateNames`. Specific lower and upper bounds can be established, e.g., `[2..4]`, `[0..*]`, with '`*`' denoting an unbound upper limit (and `[*]` as a shorthand for `[0..*]`). A constraint indicating that `alternateNames` is an `ordered` sequence is included between brackets.

Listing 1: Tonto model for person phases and roles based on a fragment of Figure 1.

```
package Persons

kind Person {
  name: string
  alternateNames: string [1..*] { ordered }
  age: number
}
phase Child specializes Person
phase Teenager specializes Person
phase Adult specializes Person

role UniversityStudent specializes Person

role Employee specializes Person
role UniversityProfessor specializes Employee
```

4.3. *Using multiple packages*

Tonto employs a multi-package system in which every `.tonto` file specifies one package, and imports can be used between packages. We exemplify this feature by adding a separate package called `PersonAndOrganizationDatatypes` in Listing 2. By default, Tonto supports the following datatypes: `number, string, boolean, date, time` and `datetime` following the existing datatypes on JSON and the most used datatypes on OntoUML catalog. Here we add custom datatypes for `int`, `Address` and `EyeColor`. Because numbers are broader than integers, we declare a custom datatype called `int`, which is a specialization of `number`. This definition of `int` is only nominal (i.e., 'opaque'). The `Address` datatype declared as a complex one formed by country, city, postal code, street, all of them with type `string`, and street number with type `int`. Lastly, there is the declaration of an *enumeration* (a particular type of Datatype in UML), which consist of unordered *literal* values. In this case, eye color is conceived of as a nominal quality structure with `Blue`, `Green`, `Brown`, and `Black` as possible values.

Listing 2: Tonto package of custom DataTypes.

```
package PersonAndOrganizationDatatypes

datatype int specializes number
datatype Address {
  country: string
  city: string
  postalCode: string
  street: string
  streetNumber: int
}
enum EyeColor { Blue, Green, Brown, Black }
```

The datatypes in the separate package can be used to revisit the attributes of `Person`. In this case, because elements are on different packages, we must first import the contents of the `PersonAndOrganizationDatatypes` package. If these are not imported, the package will not be able to make a reference to the other element. This decision prevents many suggestions from polluting the auto-completion functionality as projects grow.

Listing 3 shows the new version of the kind `Person` with full attributes. There are two options for using imported elements: with their short name (e.g., `EyeColor`) or with a qualified name (e.g., `PersonAndOrganizationDatatypes.EyeColor`) when the short name would cause ambiguities (due to its presence in more than one package).

There is also the possibility of specifying meta-attributes of attributes, enclosed by curly brackets. The `birthDate` attribute has a meta-property called `const` (i.e. immutable), `age` is `derived`, and `alternateNames` is `ordered`. Every meta-property in attributes needs to be defined inside brackets at the end of the attribute declaration. In classes, by default, every attribute is **not**: constant/immutable, ordered, or derived.

Listing 3: Revisiting the attributes of persons with imported custom datatypes.

```
import PersonAndOrganizationDatatypes

package PersonPhases

kind Person {
  birthDate: date { const }
  age: number { derived }
  name: string
  alternateNames: string [1..*] { ordered }
  eyeColor: EyeColor [1..2]
}
```

4.4. Generalization Sets

An important construct to shape taxonomies in Tonto is the *generalization set*, based on the homonymous construct in UML (and inspired in the corresponding syntax in the ML2 multi-level textual modeling language [15]). Generalization sets capture the relations

between a general class and a set of specializing classes. 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 requires instances of the general class to instantiate at least one of the subclasses. The absence of these keywords in a generalization set declaration amounts respectively to overlapping and incomplete generalization sets.

Listing 4 shows the short variant of the generalization set syntax in line 1, which includes the modifiers (in this case, `complete` and `disjoint`), the keyword (`genset`), a name for the set (`PhasesOfPerson`), the subclasses (after the `where` keyword) and the superclass (after `specializes`).

Lines 5 to 9 show the expanded variant of the same generalization set, which also includes the high-order type that is instantiated by each subclass (also known as powertype in the UML literature): `PersonTypeByAge` defined with the keyword `type` in line 3. The keywords `general` determine the specialized superclass, while the keyword `specifics` defines all subclasses, separated by a comma. Each phase previously declared is now specified to be instance of this higher-order type, marked as the `categorizer` of the generalization set. Categorizers are optional, but are instrumental in multi-level models.

Listing 4: A fragment of a Tonto specification to illustrate generalization set syntax.

```
1 disjoint complete genset PhasesOfPerson where Child, Teenager,
      Adult specializes Person
2
3 type PersonTypeByAge
4
5 disjoint complete genset PhasesOfPerson {
6   general Person
7   categorizer PersonTypeByAge
8   specifics Child, Teenager, Adult
9 }
```

4.5. Relations

Relations (corresponding to UML associations) can be declared in two different syntaxes in Tonto: in the first syntax (called ‘internal’), it is defined inside the body of the declaration of a class; in the second (‘external’), outside the body of class declarations.

Listing 5 shows the ‘internal’ variant, for the part-whole relation between a university and its departments, the constitution of a department by a staff collective with its members. The annotations `@componentOf`, `@memberOf` and `@subCollectionOf` corresponds to the homonymous stereotypes in OntoUML (a syntax inspired in Java annotations, and available to every association stereotype in OntoUML).

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. In the example, the first cardinality in line 4 represents the cardinality on the University end (`[1]`), and the second cardinality of `[2..*]` represents the Department end of the association.

The order of declaration in the first end is, after the relation stereotype: first end meta-attributes (`ordered`, `const` and `derived`, all optional); first end name between parentheses (also optional); first end cardinality (optional with default `[1]`). The association end is followed by the main relation keyword, which is: `‘<o>- -’` for a composition, i.e., non-shareable parthood; `‘- -’` (a line) for regular associations, and; `‘<>- -’` (a white diamond-adorned line) for an aggregation, i.e., shareable parthood. These keywords defines the ‘middle’ point of a relation. The relation name can be defined (optionally) after the keyword, in which case an extra keyword `‘- -’` is added after its name.

Listing 5: A fragment of the university example with relations in the internal syntax.

```
kind University specializes Organization {
  address: Address
  @componentOf
  [1] <o>-- [2..*] Department
}

kind Department {
  [1] -- constitutedBy -- [1] Staff
}

collective Staff {
  @memberOf
  [1..*] <>-- [2..*] Employee (members)
  @subCollectionOf
  [1] <o>-- [1] SeniorStaff
}
```

Listing 6 shows the definition of relations in the external syntax variant. 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 in the internal syntax.

Listing 6: Example of the external syntax variant for relations.

```
@memberOf
relation SeniorStaff [1..*] <>-- [2..*] Professor (professors
                                                   {subsets members})
```

Finally, listing 7 shows the use of mediation relations in the internal syntax with the relator pattern (the pattern is provided in the editor as a useful ‘code snippet’).

Listing 7: Example of relators added to the University package.

```
roleMixin Employer specializes Organization
role Employee specializes Person
relator EmploymentContract {
  @mediation [1..*] -- [1] Employee
  @mediation [1..*] -- [1] Employer
}
role PhDCandidate specializes Person
relator PhDEnrollment {
  @mediation [0..*] -- [1] University
  @mediation [1..*] -- [1] PhDCandidate
}
```

5. Implementation and Cross-Tool Evaluation

5.1. The Tonto Visual Studio Code Extension

We adopted Langium[4] to craft a language server for Tonto, leading to the Visual Studio Code extension depicted in Figure 2. This could be later extended to other IDEs supporting the Language Server Protocol. The Tonto Extension actively recognizes all `.tonto` files within a workspace as components of a unified project, treating each file as a distinct package.[5] A key functionality of this extension is its capability to conduct real-time UFO-based semantically-motivated syntactic verification during modeling, identifying and marking ontological inconsistencies as errors, akin to error handling in traditional programming environments. The extension implements verifications also performed in the OntoUML language, which include checking the taxonomic structures for incompatible metaproperties (non-sortals specializing sortals, rigid types specializing anti-rigid types, etc.) and for incompatible ontological natures (event types specializing endurant types and vice-versa, aspect types specializing object types and vice-versa, etc.)

The editor supports the transformation of Tonto projects to OntoUML (as JSON documents compliant with ontouml-schema) and vice-versa, which enables the integration with the functionality available in the OntoUML server. This opens us the possibility to transform Tonto specifications into gUFO-based OWL ontologies, bridging the gap from reference ontologies to operational ontologies.

In order to support the reuse of Tonto projects, we implemented a package manager. It is based on popular solutions for programming languages, for example, the Node Package Manager (NPM), the Swift Package Manager (SPM) and Cargo, the package managers for JavaScript, Swift, and Rust, respectively. The Tonto Package Manager (TPM) allows the definition of package dependencies on a *manifest file*, named `tonto.json`. An 'install' command can be used to retrieve all the necessary dependencies for a project from a distributed git repository.

[4] https://langium.org

[5] Simply adding a `.tonto` file prompts VS Code to suggest the installation of the extension, which is available in the marketplace https://w3id.org/tonto/extension. https://w3id.org/tonto documentation website was created to help the understanding of Tonto tools.

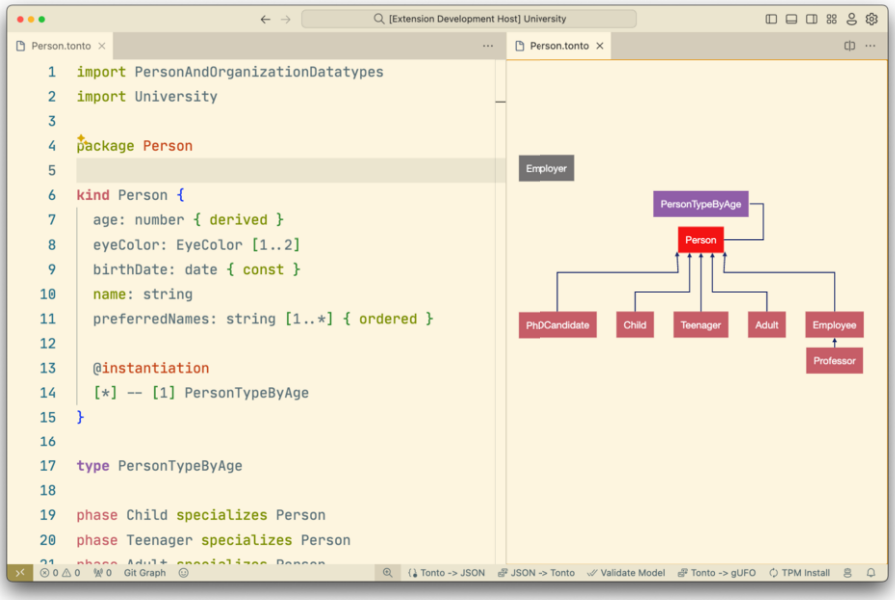

Figure 2. Example of Visual Studio Code running Tonto in the *Person* package

Listing 8: An example of a Tonto Manifest file.

```
{ "projectName": "UniversityModel",
  "version": "1.0.0",
  "publisher": "NEMO and SCS",
  "dependencies": {
    "PersonAndOrganizationDatatypes": {
      "url": "https://github.com/url-to-package",
      "directory": "path/to/package" }
},}
```

Listing 8 shows an example of a manifest file for the presented University project, showing the possibility of defining the datatypes used in this project as the equivalent of a library, allowing reuse in other projects. This is a feature that is currently indispensable in the management of software engineering projects, and which is not yet commonplace for ontology engineering.

Lastly, we implemented a prototype for automatic diagrammatic visualization of Tonto specifications showing the package being edited. This feature is shown in the right-hand side of Figure 2, and reduces the gap between a Tonto package and a corresponding OntoUML diagram.

5.2. Cross-Tool Evaluation: OntoUML and Tonto Ontologies Catalog

To evaluate the integration of Tonto with existing OntoUML tools, we employed the OntoUML Catalog to generate a new catalog[6] comprising equivalent versions of each model in Tonto. The Tonto catalog encompasses 168 models. We have implemented an 'import' command within Tonto's Command Line Interface (CLI)[7] to perform the automatic conversion of each model from its OntoUML JSON serialization to Tonto, instantiating every element using its corresponding counterpart within Tonto. Given the extensive number of models involved, an automated consistency check was executed to compare the generated Tonto model against its original JSON serialization. Analysis demonstrated an average equivalence of 98% between the models under the import command. This is due to the fact that Tonto does not contain every element from the UML language; these few elements are disregarded in the translation. Further, we have compared Tonto's syntax verification and the correspondent functionality in the ontouml-server API. A script executed concurrent syntax verification, logging error counts for each model. Results showed a 5% average error count discrepancy, with Tonto identifying more errors. Closer examination revealed that the majority of these discrepancies stemmed from pre-existing syntactic issues in the source OntoUML models or limitations in the JSON-to-Tonto transformation (e.g., anonymous generalization sets that requires naming in Tonto). This highlights the potential need for manual adjustments when importing existing models into Tonto. The observed increase in errors identified in Tonto is likely attributable to its more strict validation mechanisms, especially regarding element naming conventions.

Tonto prohibits the use of arbitrary strings, special characters, and numeric prefixes in element names. This restriction, intended to preserve integrity, can create naming conflicts during the import process, as some elements may need to be renamed to prevent data loss. These conflicts then generate errors specific to Tonto, relating to duplicate names, which would not be present in the original OntoUML JSON serialization. Furthermore, the import command may occasionally struggle to correctly determine the appropriate package, leading to additional errors. These issues are less likely to occur in OntoUML models contained within a single JSON project, for which the importing process is simpler. Despite these discrepancies, overall findings demonstrate a high-level of functional agreement between the original OntoUML server implementation and the syntactic verifications that were ported into the Tonto VS Code extension.

6. Related Work

We discuss here how Tonto compares with OWL Turtle and XML syntaxes, with the Ontological Modeling Language (OML),[8] and with OntoUML. This exercise highlights Tonto's unique contributions within the landscape of ontology notations.

Use of a foundational ontology. A key aspect of Tonto is that its design is tailored to represent the ontological distinctions and patterns defined by UFO. In contrast, the other textual syntaxes considered here are largely ontologically neutral. By incorporating

[6] https://w3id.org/tonto/catalog
[7] https://w3id.org/tonto/cli
[8] https://www.opencaesar.io/oml/

UFO distinctions, ontologies created in Tonto can benefit from extensive semantically-motivated syntactic verification, and clearer expression of ontological commitments.

Readability. Tonto adopts a human-readable syntax designed for intuitive authoring of UFO-based models, prioritizing accessibility over the complexities of machine-oriented formats like XML. Like Turtle and OML, Tonto offers a user-friendly textual syntax, though further research is needed to compare their relative readability. Notably, Turtle serializations of Tonto models exhibit increased verbosity. For example, one model could expand from 71 lines in Tonto format to 160 lines in OWL. This difference is primarily due to Turtle's syntactic requirements and the need to use of punning to express the instantiation of UFO's taxonomy of types within OWL.

OWL Compatibility. OWL is a de facto standard for ontology codification in the Semantic Web, and hence, the ability to transition from alternative languages to OWL is important. The other languages we have considered are either OWL serializations (Turtle and XML) or support conversions to it. OML (and likewise OWL) distinguishes itself by allowing for explicit axiom description, offering enhanced built-in datatype support, and accommodating individuals. These features position OML for superior OWL integration compared to Tonto, highlighting OML's nuanced capabilities in ontology modeling and interaction with OWL standards.

Code-based tools and Version Control. Code-based ontology development tools offer advantages like robust version control, crucial for collaborative projects. Notably, all text-based ontology languages in this study (including Tonto and OML) inherently support version control. However, their streamlined syntax simplifies change visualization using diff tools when compared to diagrammatic languages. While OntoUML integrates with the Visual Paradigm plugin and supports version control through its JSON schema, merging model changes presents significant challenges. These difficulties likely stem from complexities in the model serialization mechanism, where minor model changes can result in large serialization discrepancies. The same does not occur in Tonto, in which small changes only have local impact.

Dependency Management Support. While OWL includes an import directive, which is used in an ontology to gain access to the entities defined in other ontologies, there are no tools to support dependency management in a manner analogous to that offered to software engineering environments. This is even more critical in the case of OntoUML, as UML tools typically lack support for projects spanning several files, let alone distributed over several repositories. The Tonto package manager supports version declaration and checking, licensing declaration, and automated download from git repositories. Similar functionality is offered for OML as OML projects are Gradle[9] projects.

Logical Expressivity. We have not yet implemented support for the specification of invariants and derivation rules in Tonto, or other forms of axiomatization that are currently supported in OWL, OML and OntoUML, either directly or in the form of additional languages (such as SPARQL and SHACL, or OCL in the case of OntoUML). This is the first priority in the further development of the language and associated toolset.

Table 1 presents a summary of our analysis.

[9] https://gradle.org

Table 1. Comparing Tonto and other notations.

Characteristic	Tonto	Turtle	OML	XML	OntoUML
Based on a foundational ontology	x				x
Readability	x	x	x		x
Integration with OWL	x	x	x	x	x
Easy to use version control	x	x	x		
Direct application of code-based tools	x	x	x	x	
Dependency Management Support	x	x	x	x	
Logical expressivity (invariants, derivations)		x	x	x	x

Finally, while numerous textual languages exist for modeling purposes, such as Alloy [16], OCL [17], and MontiCore [18], they do not specifically target ontology specification. Tonto differentiates itself by focusing on UFO-based ontology modeling, addressing a niche that these languages do not. An analysis of these languages falls beyond the scope of this paper.

7. Conclusion

This article introduced Tonto, a textual language designed for UFO-based ontology modeling, addressing limitations of diagrammatic representations. Tonto enhances modeling efficiency through text-based syntax, real-time validation, and transformation capabilities, supported by a Visual Studio Code extension and a package manager for modularization. It offers a novel approach for the development of well-founded ontologies, leveraging text-based tools' advantages for version control, modularization, and integration with development environments.

We expect to continue the development of the language and toolset with a number of features, including: (i) support for rich axiomatization (invariants, derivation rules); (ii) specification of individuals and values for attributes; (iii) documentation generation (akin to Javadoc documentation); (iv) automated ontology testing and verification; and finally, (v) integration with Large Language Models (LLMs) to provide functionally currently offered in programming environments with tools such as Github copilot.

Concerning the latter, integrating Large Language Models (LLMs) with Tonto's text-based ontology representation can streamline ontology development, offering features like intelligent code suggestions and auto-completion. We believe LLMs can also support improved human-ontology interaction, enabling natural language interfaces for easier understanding, querying, and manipulation, making ontologies accessible to a wider audience. We also aim to design and perform experiments to empirically assess the performance of Tonto when contrasted with OntoUML in a number of tasks. In this context, we will also investigate the joint use of Tonto with automatically produced diagrammatic representations.

Acknowledgment

This study was supported in part by FAPES (1022/2022) and CNPq (443130/2023-0, 313412/2023-5).

References

[1] Guizzardi G, Fonseca CM, Almeida JPA, Sales TP, Benevides AB, Porello D. Types and Taxonomic Structures in Conceptual Modeling: A Novel Ontological Theory and Engineering Support. Data & Knowledge Engineering. 2021 Jul;134:101891.

[2] Guizzardi G. Ontological foundations for structural conceptual models. Enschede, The Netherlands: Centre for Telematics and Information Technology; 2005.

[3] Guizzardi G, Benevides AB, Fonseca CM, Porello D, Almeida JPA, Sales TP. UFO: Unified Foundational Ontology. Applied Ontology. 2022;17:167-210.

[4] Fonseca CM, Sales TP, Viola V, Fonseca LBR, Guizzardi G, Almeida JPA. Ontology-Driven Conceptual Modeling as a Service. In: JOWO 2021 - The Joint Ontology Workshops. CEUR-WS; 2021. p. 1-12.

[5] Benevides AB, Guizzardi G, Braga BFB, Almeida JPA. Validating Modal Aspects of OntoUML Conceptual Models Using Automatically Generated Visual World Structures. J UCS. 2010 2;16:2904-33.

[6] Guidoni G, Almeida JPA, Guizzardi G. Transformation of Ontology-Based Conceptual Models into Relational Schemas. In: Conceptual Modeling. ER 2020. Springer; 2020. p. 315-30.

[7] Sales TP, Guizzardi G. Ontological Anti-Patterns: Empirically Uncovered Error-Prone Structures in Ontology-Driven Conceptual Models. Data & Knowledge Engineering. 2015 2;99:72-104.

[8] Dori D. Words from pictures for dual-channel processing. Comm of the ACM. 2008;51(5):47-52.

[9] W3C. RDF 1.1 Turtle Terse RDF Triple Language; 2014. W3C Recommendation 25 February 2014. Available from: https://www.w3.org/TR/turtle/.

[10] W3C. OWL 2 Web Ontology Language XML Serialization (Second Edition); 2009. W3C Recommendation 11 December 2012. Available from: https://www.w3.org/TR/owl2-xml-serialization/.

[11] W3C. OWL 2 Web Ontology Language Document Overview (Second Edition); 2012. W3C Recommendation 11 December 2012. Available from: https://www.w3.org/TR/owl2-overview/.

[12] Wagner DA, Chodas M, Elaasar M, Jenkins JS, Rouquette N. Ontological Metamodeling and Analysis Using openCAESAR. In: Madni AM, Augustine N, Sievers M, editors. Handbook of Model-Based Systems Engineering; 2023. p. 925-54.

[13] Almeida JPA, Falbo RA, Guizzardi G, Sales TP. gUFO: A Lightweight Implementation of the Unified Foundational Ontology (UFO); 2019. Available from: http://purl.org/nemo/doc/gufo.

[14] Fonseca CM, Guizzardi G, Almeida JPA, Sales TP, Porello D. Incorporating Types of Types in Ontology-Driven Conceptual Modeling. In: Conceptual Modeling. ER 2022. vol. 13607; 2022. p. 18-34.

[15] Fonseca CM, Almeida JPA, Guizzardi G, Carvalho VA. Multi-level Conceptual Modeling: From a Formal Theory to a Well-Founded Language. In: Conceptual Modeling. ER 2018; 2018. p. 409-23.

[16] Jackson D. Software abstractions: Logic, Language, and Analysis. MIT Press; 2016.

[17] Cabot J, Gogolla M. Object Constraint Language (OCL): A Definitive Guide. vol. 7320; 2012. p. 58-90.

[18] Krahn H, Rumpe B, Völkel S. MontiCore: a framework for compositional development of domain specific languages. International Journal on Software Tools for Technology Transfer. 2010;12:353-72.