Ontology-Driven Conceptual Modeling as a Service

Claudenir M. Fonseca[1], Tiago Prince Sales[1], Victor Viola[2], Lucas B. R. da Fonseca[3], Giancarlo Guizzardi[1,4] and João Paulo A. Almeida[5]

[1]*Conceptual and Cognitive Modelling Research Group (CORE), Free University of Bozen-Bolzano, Bolzano - Italy*
[2]*Farfetch, Porto - Portugal*
[3]*X-Team International, Melbourne - Australia*
[4]*Services, Cybersecurity & Safety, University of Twente, Enschede - The Netherlands*
[5]*Ontology & Conceptual Modeling Research Group (NEMO), Federal University of Espírito Santo - Brazil*

Abstract

In the past decades, the Unified Foundational Ontology (UFO) has played an important role in supporting the development of ontologies in academic and business settings, being employed to represent widely diverse domains. In this period, a dedicated community of researchers has worked to support UFO and its representation language, OntoUML, by creating the OntoUML Lightweight Editor (OLED). Now that a new version of OntoUML is available, the need for up-to-date tool support has exposed the limitations of OLED, its development context, and the difficulties of bringing research contributions to the hands of modelers. To tackle these issues, this paper reflects upon the experiences of this community taking into consideration the goals of researchers (as developers) and modelers to devise a new microservice-oriented modeling infrastructure for OntoUML, called OntoUML as a Service (OaaS). This infrastructure supports future practical contributions to the language with a focus on lowering the entry barrier for the development new contributions and enabling an easier deployment to modelers. The paper also discusses the details of implementing OaaS through a number of projects that currently implement this infrastructure.

Keywords

ontology-driven conceptual modeling, Unified Foundational Ontology, OntoUML, service-architecture, applied ontology

1. Introduction

For a few decades now, ontologies have supported organizations in building precise representations of their problem domains. The inherent complexity of reality, however, places a great burden on modelers when developing ontologies as they need to consistently account for domains with very distinct characteristics. The usage of adequate formal languages in the representation of ontologies plays a significant role, in this sense, since these languages' constructs (e.g., predicates, classes, or relations) may serve as better cognitive and formal tools in the modeling task. OntoUML is an example of such a language, designed to enrich the Unified Modeling Language (UML) [1] with the concepts of the Unified Foundational Ontology (UFO)

FOMI 2021: 11th International Workshop on Formal Ontologies meet Industry
cmoraisfonseca@unibz.it (C. M. Fonseca); tiago.princesales@unibz.it (T. P. Sales); victorviola86@gmail.com (V. Viola); lucas.bassetti@x-team.com (L. B. R. d. Fonseca); giancarlo.guizzardi@unibz.it (G. Guizzardi); jpalmeida@ieee.org (J. P. A. Almeida)


[2]. By leveraging the several logical, philosophical, and psychological theories collected in UFO, OntoUML defines a set of constructs and *semantically-motivated syntactical constraints* [3] tailored for ontology-driven conceptual modeling (ODCM). In other words, OntoUML shifts the inherent complexity of reality towards the language's definition such that every syntactically valid model represents a sound ontology in terms of UFO.

One advantage of OntoUML being an extension of UML is that any UML CASE tool[1] that includes the profile mechanism (representation of constructs as stereotyped elements, such as classes and associations) could be used to build models. This advantage is limited, however, since these tools are unable to support basic OntoUML features, such as checking language constraints in users' models to prevent the development of unsound ontologies. To provide modelers with adequate support, a community of researchers and developers worked on the development of the OntoUML Lightweight Editor (OLED) [4], a CASE tool based on the Eclipse Modeling Framework (EMF) [5]. OLED, alongside a later spin-off tool, the Menthor Editor [6], presented the most accurate implementation of OntoUML and offered state-of-the-art model processing services, such as model verification, model simulation [7], anti-patterns detection [8], and multiple transformation options (e.g., OWL, SWRL, SBVR) [9, 10]. Additionally, OLED would include model editing services beyond basic diagramming that also explored OntoUML's definitions, such as inductive modeling [11]. We broadly refer to these model editing and model processing services as *model intelligence services*.

As the work towards a new version of OntoUML [9, 12, 13] – OntoUML 2 – began, the demand for updated tools and model-based services exposed several limitations of both OLED and Menthor, such as (i) its monolithic architecture that allowed for highly coupled components and that significantly hindered maintainability; (ii) its Ecore-based metamodel required changes that would deeply affect all features making updates challenging; (iii) its inability to adequately handle large-scale models; and (iv) the ever-growing technical-debt of the project, accumulated over the years by losing several contributors (e.g., graduating students, researchers moving to different institutions) who left pending issues on the services they developed. These limitations are not specific to OntoUML or any single modeling language in that regard, but the long-term consequences of the adopted software architecture and the research-centric environment in which the tools were developed.

In this paper, we propose a microservice-based [14] infrastructure for OntoUML, referred to as *OntoUML as a Service* (OaaS). This proposal is developed as an answer to a number of requirements, presented in section 2, emerging from previous experiences of the OLED and Menthor communities, and by the goals of involved stakeholders (i.e., developers, researchers, and modelers). OaaS is discussed in section 3, where each artifact of the infrastructure as well as their implementations are presented in detail. Currently available OaaS model intelligence services are presented in section 4. By the end, we position OaaS in terms of OLED, Menthor, and other related work in section 5, and present our final remarks in section 6.

[1] Computer-aided software engineering tool.

2. Stakeholders and Requirements

There are three main stakeholder types whose concerns constrain how we realized the new OntoUML modeling infrastructure, namely:

- **Service Developer**: a person who implements model intelligence services. This is usually a researcher with limited programming skills whose goal is to implement an algorithm that manipulates OntoUML models (e.g. a transformation from OntoUML to gUFO [15]), which she would like to make available to others via a modeling environment. Thus, a service developer needs a programming library that offers the possibility to create, edit, and query OntoUML models efficiently, while having a low learning curve. Since the kind of service to be implemented may varies significantly, as well as the developers skill set, the possibility to freely choose libraries, platforms, and even the programming languages can be extremely desirable.
- **Modeler**: a person who designs OntoUML models, be it in an academic, governmental, or industrial setting. Modelers want to develop high-quality models by leveraging model intelligence services, but also to have an enjoyable experience while using the modeling tool as a whole. Ideally, modelers prefer to stick with the tool they are accustomed to and just have OntoUML support incorporated into it. For them, it is fundamental that their tool is reliable and user-friendly. In industrial settings, it is also paramount that the tool can handle large-scale models and support collaboration.
- **Tool Developer**: a developer who curates model intelligence services and aggregates them in a tool to make them available to modelers. Tool developers are OntoUML experts who are also experienced programmers. They depend on service developers to create services, while service developers rely on them to make their services available to modelers. Tool developers care about the overall experience modelers will have, so they want to assure the quality of a service before incorporating it. Finally, tool developers seek standardization of model intelligence services through a flexible yet consistent interface, that can be easily maintained, and that amounts to low additional technical debt.

With the drivers and characteristics of the aforementioned stakeholders in mind, we derived the following core set of requirements for the new OntoUML modeling infrastructure:

R1 The technological requirements (e.g., frameworks, programming languages, platforms) imposed upon service and tool developers by the infrastructure should be minimal;
R2 The learning curve for service developers to use the infrastructure should be low;
R3 Tool developers should be shielded from the implementation details of model intelligence services, just as service developers should be shielded from the details of modeling tools;
R4 Infrastructure components should be highly decoupled;
R5 Infrastructure components should be easily maintainable;
R6 The infrastructure should be able to handle large-scale models;
R7 Model intelligence services should be easily integrable into (third party) modeling tools.

3. OntoUML as a Service

To realize the aforementioned requirements, we developed a modeling infrastructure based on a microservice architecture [14] and inspired by Model-Driven Engineering (MDE) [16] and the Language Server Protocol (LSP)[2]. This infrastructure, deemed *OntoUML as a Service* (OaaS), consists of:

- **ontouml-js**: an OntoUML API to create, edit, and query models, written in Javascript.
- **ontouml-schema**: a specification of how to serialize OntoUML models as JSON objects.
- **ontouml-server**: a reference implementation of an OntoUML server, which exposes a curated set of model intelligence services via a web API.
- **ontouml-vp-plugin**: a reference OntoUML modeling tool that consumes services from the ontouml-server. It is implemented in Java as a plugin for Visual Paradigm.

This infrastructure is designed to support OntoUML modeling environments in a microservice architecture, as depicted in Figure 1, which consists of modeling tools, OntoUML servers, and model intelligence services.

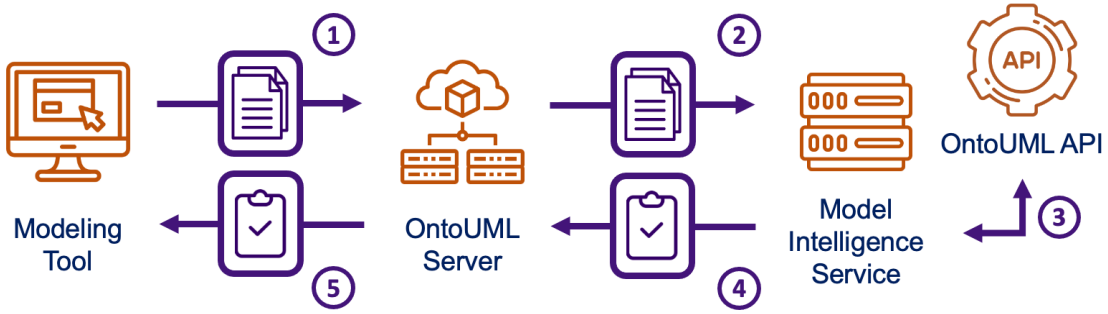

Figure 1: The OaaS infrastructure: (1) a modeling tool requests a service provided by an OntoUML server; (2) the OntoUML server executes the requested service passing the serialized model and additional arguments to the service; (3) the service process the received model employing a tailored OntoUML API; (4) the service returns the result of its model processing to the server; (5) finally, the server sends a response message to the modeling tool containing the service's output.

In this picture, a *modeling tool* is an application with which modelers directly interact, such as a desktop modeling tool (e.g. OLED, Menthor Editor, Visual Paradigm, MagicDraw, Astah) or a browser-based tool. Such a tool provides model editing features and is responsible for meeting modelers' expectations regarding ease of use, consistency, and reliability, and for providing them with an overall satisfying user experience. To provide model intelligence services, however, tools must include a layer capable of performing HTTP requests to OntoUML servers.

An *OntoUML server* is an application that exposes model intelligence services via a web API. It listens for requests from tools on dedicated endpoints, processes these requests, delegates them to the corresponding services, and sends the results back to the tools. The communication between a tool and a server is governed by OaaS' service protocol, which specifies how to send

[2]https://microsoft.github.io/language-server-protocol/

requests messages (e.g. which HTTP method to use, how to format the request body), as well as how these requests should be answered (e.g. which HTTP status code to use, how to format the response body). A crucial part of these messages is the OntoUML model being sent from the tool to the server, which is structured according to the `ontouml-schema` (see subsection 3.1).

Lastly, a *model intelligence service* is an application that operates on an OntoUML model, such as verifying its compliance with the language's syntax or transforming it into an OWL specification. It may be deployed as a library, in which case it can be directly executed from within an OntoUML server, or it may be deployed in its own web server. In the latter case, the OntoUML server will communicate with it using the same protocol that governs the tool-server communication. Naturally, model intelligence services need to deserialize, parse, and manipulate models they receive from requests. If the service is implemented in Javascript, this can be done via *ontouml-js*, our OntoUML API (see subsection 3.2).

3.1. ontouml-schema

OaaS foresees the implementation of services that receive, as input, models serialized as JSON objects. The shape of these objects is standardized by the ontouml-schema[3], which is written as a JSON document itself according to the JSON Schema[4] vocabulary. The purpose of this specification is two-fold. First, it informs developers about OntoUML metamodel's elements, such as classes, relations, and generalization sets, as well as their properties. For instance, an object describing a class has a boolean field named isAbstract. It also defines how to serialize diagrammatic information in a tooling independent way. Second, it can be used to automatically validate if a serialized model conforms to its rules, accusing any deviations. Listing 1 presents an excerpt of an OntoUML model[5] serialized according to this schema.

Listing 1: Excerpt of an OntoUML model serialized into ontouml-schema.

```
{
  "id": "sample_project",
  "type": "Project",
  "model": {
    "id": "root_package",
    "type": "Package",
    "contents": [
      {
        "id": "person",
        "type": "Class",
        "stereotype": "kind",
        "restrictedTo": ["functional-complex"]
      }
    ]
  },
  "diagrams": [...]
}
```

The schema specification acts as a shared OntoUML "metamodel" for services and tools, but there is no requirement for that to be their internal metamodels (in case they employ MDE).

[3]Source code at https://purl.org/krdb-core/ontouml-schema.

[4]https://json-schema.org/

[5]For the meaning of notions such as Kind, Subkind, Functional Complex, Material Relation, Relator, Category, which appear in the figures and listings throughout this article, the reader is referred to [2].

Internally, applications can simply maintain a compatibility layer that (de)serializes models as necessary.

3.2. ontouml-js

The OaaS infrastructure, with its service contract relying on platform independent technologies like JSON and HTTP, is designed to provide maximum flexibility to service and tool developers alike. However, this simplicity leaves a gap regarding programming APIs that can manipulate OntoUML models. To fill it, OaaS relies on programming language-specific OntoUML APIs, made available via package managers, such as NPM, for node.js, and Maven, for Java.

ontouml-js[6] is one such API. It is a TypeScript library[7] that provides a collection of classes corresponding to OntoUML metamodel's elements, and an extensive set of easy to use methods to create, modify, and query OntoUML models efficiently. It also includes out-of-the-box support for serializing and deserializing models compliant with the ontouml-schema, as well as a validation function for such models. Moreover, ontouml-js provides classes and interfaces for the implementation of model intelligence services, including service options and issues.

Listing 2: Using ontouml-js to create a model containing two classes and a relation.

```
let project = new Project({ name: 'My Project' });
let model = project.createModel({ name: 'Model a.k.a. Root Package' });
let person = model.createKind('Person');
let school = model.createKind('School');
let studiesAt = model.createMaterialRelation(person, school, 'studies at');
```

The ontouml-js API is used by all currently available OaaS services (which are all coded in Typescript), but a Java counterpart is already in the works, which is based on the (de)serialization mechanism implemented for the ontouml-vp-plugin project.

3.3. ontouml-server

The ontouml-server project[8] provides a reference implementation of an OntoUML server in the OaaS architecture, exposing a curated set of services. Each service route defined in the server answers to HTTP POST requests from client tools which must be sent with a JSON payload containing a serialized OntoUML model (in the "project" field) and an object containing all arguments necessary for the execution of the desired service (in the "options" field), as illustrated in Listing 3.

Listing 3: HTTP POST request sent to the ontouml-server requesting model transformation.

```
POST /transform/gufo
Content-Type: application/json

{
  "project": {
    "id" : "sample_project",
```

[6]Source code at https://purl.org/krdb-core/ontouml-js.
[7]TypeScript libraries can be used by both TypeScript and Javascript applications.
[8]Source code at https://purl.org/krdb-core/ontouml-server.

```
    "type" : "Project",
    "model" : {...},
    "diagrams": [...]
  },
  "options": {
    "format":"Turtle",
    "baseIri": "https://example.com"
  }
}
```

A service request is answered with a JSON object containing the output generated by it (in the "result" field) and an optional array of issues that may have emerged during processing (in the "issues" field), such as errors that prevented the service from executing correctly or warnings that are relevant to the requester. A successful response message uses the HTTP status 200 OK, while an error response message (sent for requests that could not be processed) uses a status code according to the HTTP conventions. For instance, 400 Bad Request would be used to indicate that a request was sent with some error, such as lacking the "project" field. Listing 4 presents an example of a successful response message to the prior request.

Listing 4: HTTP response (partial) received from the ontouml-server.

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "result": "@prefix : <https://example.com#>.\n@prefix gufo: <http://purl.org/nemo/gufo#>.\n...",
  "issues": null
}
```

The service contract defined by OaaS currently comprises the serialization of OntoUML models, the shape of request messages, and the shape of responses. The service endpoint corresponding to some service is defined by the specific server implementation, while the shape of the fields "options", "result", and "issues", are defined by the specific service implementation. A standardization for the "issues" array field is still under development.

All services currently available in ontouml-server, including model verification, model transformations into (gUFO-based [15]) OWL ontologies and relational schemas [10], and modularization [17], are executed inside the server itself. Nonetheless, OaaS allows for communication between servers using the presented service contract. In this way, dedicated servers can act as brokers, orchestrating and providing multiples services implemented in different programming languages, frameworks, or libraries, with the HTTP protocol serving as the common communication layer. Adopting this microservice architecture, OaaS can provide a higher-level decoupling between the different artifacts.

3.4. ontouml-vp-plugin

The ontouml-vp-plugin project[9] is a plugin for Visual Paradigm[10], a multi-platform CASE tool. It adds a fin extension to Visual Paradigm that enables the interaction with OntoUML servers. In order to integrate it with OaaS, the plugin (i) adds OntoUML's constructs (stereotypes for

[9] Available at https://purl.org/krdb-core/ontouml-plugin.

[10] https://www.visual-paradigm.com

classes, associations, and attributes), (ii) enable model (de)serialization in compliance with the ontouml-schema, and (iii) provide access to a deployed instance of the ontouml-server. In addition to these features, the ontouml-vp-plugin also adds to the Visual Paradigm context menus and event listeners that enhance the overall experience for OntoUML modelers, but the decoupling enabled by OaaS allows tool developers to rely on existing modeling tools to provide most of the model editing features necessary for modelers. Moreover, service developers must not concern themselves with the tool being used by modelers when developing a service to provide model processing features as they work on completely decoupled projects. Figure 2 presents an example OntoUML model being built on Visual Paradigm and part of the interface provided by the ontouml-vp-plugin.

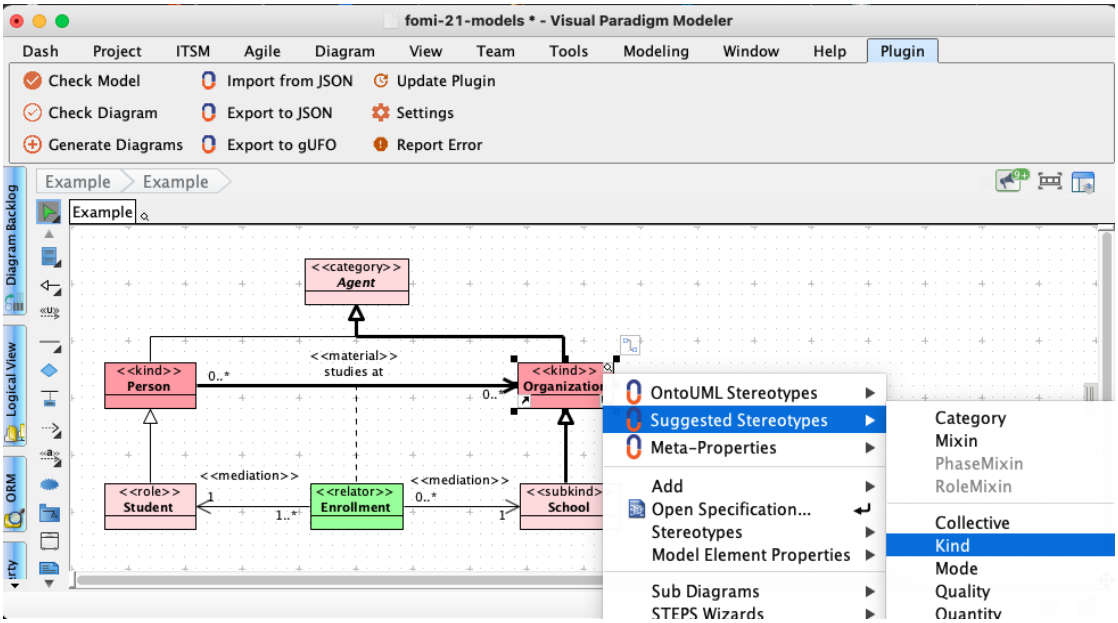

Figure 2: An OntoUML model being built in Visual Paradigm using the ontouml-vp-plugin.

Notice that OaaS can support the development of tools tailored for OntoUML, like OLED and the Menthor Editor, as well as extensions of commercial modeling tools, which may even share an OntoUML server. The latter is particularly interesting in industrial settings, as organizations tend to stick with the modeling tools they have already been chosen.

4. Model Intelligence Services

We propose the notion of *model intelligence services* as those services designed to support model engineering tasks (e.g., design, verification, validation, verbalization) by leveraging the language's metamodel. These services are intended to be entirely, or mostly, automated. Both model processing and model editing services can fit the notion of model intelligence services. Their implementation in microservice architectures, however, is not required, being this a

particular aspect of OaaS's implementation.

In OntoUML's case, its model intelligence services can leverage UFO's knowledge, which is embedded into the language's definition, to support more sophisticated features. At the moment, the set of services developed in the OaaS infrastructure is the following:

- **verification**: the verification service [9] checks if a model is compliant with OntoUML's syntax, returning a list of deviations. Each deviation has a severity level. An "error" indicates violations that generate logical or ontological inconsistencies in the model, while a "warning" indicates a lesser problem, such as an under-specified model.
- **ontouml2gufo**: this OWL transformation service generates ontologies based on the Lightweight Implementation of the Unified Foundational Ontology (gUFO) [15], allowing OntoUML users to take advantage of their ontologies in the OWL ecosystem.
- **ontouml2db**: the relational database transformation service generates relational schemas from OntoUML models [10]. More specifically, OntoUML's semantics informs the service's algorithm on suitable strategies to generate table declarations turning the knowledge present in the ontology into design decisions.
- **ontouml2alloy**: the Alloy transformation service is an update of the simulation service originally present in OLED [7]. The transformation generates the necessary files for model simulation through the Alloy Analyzer tool [18], now supporting OntoUML 2.
- **complexity management**: the complexity management service offers algorithms to handle large-scale models, such as pattern-based view extraction [19], modularization based on relational contexts [17], and model abstraction [20]. These algorithms help modelers to organize and communicate large models in a cognitively tractable manner.

5. Related Work

The requirements driving the proposal and implementation of OaaS (see section 4) are largely inspired by limitations observed in past implementations of software to support OntoUML modeling, particularly with the observations OLED [4] and Menthor [6]. Thus, a comparison between OaaS and these tools, in light of these requirements, can highlight the benefits of our approach.

In the case of OLED and Menthor, a developer seeking to implement a new service will face a large monolithic Java project. This architecture, which is similar to many other tools, imposes strict technological requirements on this developer, such as using Java and the ECore-based OntoUML API [21], which goes against **R1**. As consequence, they face a steep learning curve on both project-specific and platform-specific aspects (e.g., learning the EMF which supports these tools), which hurts **R2**. Not only that, but developers cannot adopt technologies they are more familiar with or that better suit their needs due to incompatibilities with the rest of the project. OaaS overcomes such issues for model processing services by hiding them behind HTTP APIs.

The use of an ECore-based API for manipulating models has shown, in practice, to be a serious shortcoming when querying large-scale models, thus hindering **R6**. This is particularly evinced when implementing model-driven transformations on such models and running queries like "list of all descendants of class X", which required parsing the whole model to answer. We

have resolved this issue on ontouml-js, our custom-made API, by adopting a combination of derived fields and dynamic caching, which resulted in a drastic increase in performance.

The services available on OLED and Menthor, for being developed inside monolithic architectures, cannot be easily reused by other applications, even if these share the same necessary dependencies. Even with the adoption of proper modularization of its code, many interaction points between services still remain, and the involvement of multiple developers in the history of the tool contributes to its lower coordination. In fact, that was a significant challenge in the development of Menthor, which had OLED's code as its starting point. The reuse of OLED's code led to laborious code refactoring to avoid re-implementing shared features. Designed with **R7** in mind, OaaS requires a separation between its artifacts, promoting the reuse of servers between tools, the reuse of services by different servers, as well as the reuse of tailored libraries implemented and shared by developer communities.

The interdependence of services within OLED and Menthor also hinders their maintainability, as it becomes harder to remove, replace, or update services (thus, hurting **R5**). This is also relevant as the original services' developers, oftentimes PhD candidates and researchers, no longer maintain their projects, which often forces tool developers to drop services due to a lack of capable developers to support it (see **R4**). OaaS offers a more reliable basis to prevent such issues, being easier to drop individual services in these ultimate scenarios. Moreover, the possibility of deploying services as containers can guarantee a consistent environment for running services in the long-term, regardless of changes in operational systems, programming languages, or order aspects that could make services' code deprecated otherwise, as was recently the case of Adobe Flash in web browsers.

Finally, OaaS's tools and services are completely hidden from one another behind the service contract, which means the implementation one is transparent to the other (see **R3**). OLED and Menthor may not require from developers similar degrees of transparency, which can enable certain features beyond the capabilities of OaaS's current scope, but the cost of higher complexity and dependency of developers adherence to internally defined interfaces and APIs.

Another approach related to OaaS is the Language Server Protocol (LSP). Not unlike OaaS, LSP defines a service contract between text editors and language servers, where text editors can request any kind of feature (including text editing and processing features) from a server dedicated to a target textual language. This not only frees the text editor to focus on the front-end that users interact with, but also promotes great reuse of language servers that can be easily added to editors and shared between editors. In this regard, LSP is much more comprehensive than OaaS comprising all sorts of use cases involving textual languages, but on the other hand, it requires editors that support the protocol in other to work. OaaS is tailored to a single language and facilitates the extension of existing tools rather them requiring the development of new ones. Moreover, visual languages like OntoUML have a model/view separation not present in textual languages that can be easily serialized and may rely on the position of tokens in text files to express semantics. Efforts to promote an extension of LSP to include visual languages exist, but are still gaining traction and face some of the same drawbacks like requiring compatible tools. Nonetheless, further investigation of LSP should inspire future developments of OaaS as it grows in scope.

6. Conclusions

This work presents the OaaS infrastructure, which is designed to support the implementation of future practical research in OntoUML. This infrastructure, based on a microservice architecture and materialized in a number connected projects (i.e., ontouml-schema, ontouml-server, ontouml-vp-plugin, and ontouml-js), seeks to mitigate a number of issues observed in the development of the OLED and Menthor Editor tools.

As result, the proposal enforces the decoupling of its components by design (**R4** and **R3**), exposing developers to a very low learning curve to integrate their services into OaaS (**R2**), as well as imposing almost no technological requirements on them (**R1**). We expect these aspects, along with the improved maintainability (**R5**) and performance (**R6**), to foster applied research in OntoUML. Moreover, we also expect these services to arrive sooner to modelers as they can be easily integrated into (third-party) tools these modelers are used to work with (**R7**).

As future work, we should broaden OaaS' scope to cover use cases where the current service contract offers limited support, particularly those involving model editing features and stateful communication. LSP should also be analyzed in more detail as to identify use cases also applicable to OaaS and investigate the possibility of employing OntoUML servers in tools supporting LSP. We also hope to mature the infrastructure by supporting additional development resources, such as new OntoUML libraries (particularly a library for Java based on the ontouml-vp-plugin project) and project templates for services.

Acknowledgments

We are grateful to the Zorginstituut Nederland supporting this initiative in the context of the ODIN and THOR projects. We thank them for testing and giving feedback on the first iterations of this modeling infrastructure. Claudenir M. Fonseca and Giancarlo Guizzardi are supported by the NeXON Project (Free University of Bozen-Bolzano). This study was financed in part by CAPES (grant 23038.028816/2016-41) and CNPq (grants 313687/2020-0 and 407235/2017-5).

References

[1] OMG® Unified Modeling Language®, OMG® Unified Modeling Language® (OMG UML®) Version 2.5.1, Technical Report, Object Management Group (OMG®), 2017. URL: http://www.omg.org/spec/UML/2.5.1.

[2] G. Guizzardi, Ontological foundations for structural conceptual models, Ph.D. thesis, University of Twente, 2005.

[3] V. A. Carvalho, J. P. A. Almeida, G. Guizzardi, Using reference domain ontologies to define the real-world semantics of domain-specific languages, in: M. Jarke, et al. (Eds.), Advanced Information Systems Engineering, Springer, Cham, 2014, pp. 488–502.

[4] J. Guerson, T. P. Sales, G. Guizzardi, J. P. A. Almeida, OntoUML Lightweight Editor: A model-based environment to build, evaluate and implement reference ontologies, in: IEEE 19th International Enterprise Distributed Object Computing Workshop, 2015, pp. 144–147.

[5] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Addison-Wesley, 2008.
[6] J. L. R. Moreira, T. P. Sales, J. Guerson, B. F. B. Braga, F. Brasileiro, V. Sobral, Menthor Editor: An ontology-driven conceptual modeling platform, in: O. Kutz, S. de Cesare (Eds.), 2nd Joint Ontology Workshops (JOWO), volume 1660, CEUR, 2016.
[7] A. B. Benevides, G. Guizzardi, B. F. B. Braga, J. P. A. Almeida, Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures, J. Univers. Comput. Sci. 16 (2010) 2904–2933.
[8] T. P. Sales, G. Guizzardi, Ontological anti-patterns: Empirically uncovered error-prone structures in ontology-driven conceptual models, Data Know. Eng. 99 (2015) 72–104.
[9] G. Guizzardi, C. M. Fonseca, J. P. A. Almeida, T. P. Sales, A. B. Benevides, D. Porello, Types and taxonomic structures in conceptual modeling: A novel ontological theory and engineering support, Data Know. Eng. 134 (2021) 101891.
[10] G. L. Guidoni, J. P. A. Almeida, G. Guizzardi, Transformation of ontology-based conceptual models into relational schemas, in: G. Dobbie, et al. (Eds.), Conceptual Modeling. ER 2020, Springer, Cham, 2020, pp. 315–330.
[11] G. Guizzardi, A. P. das Graças, R. S. S. Guizzardi, Design patterns and inductive modeling rules to support the construction of ontologically well-founded conceptual models in OntoUML, in: Advanced Information Systems Engineering Workshops, 2011, pp. 402–413.
[12] C. M. Fonseca, D. Porello, G. Guizzardi, J. P. A. Almeida, N. Guarino, Relations in ontology-driven conceptual modeling, in: International Conference on Conceptual Modeling, Springer, 2019, pp. 28–42.
[13] J. P. A. Almeida, R. A. Falbo, G. Guizzardi, Events as entities in ontology-driven conceptual modeling, in: International Conference on Conceptual Modeling, Springer, 2019, pp. 469–483.
[14] J. Lewis, M. Fowler, Microservices: a definition of this new architectural term, https://martinfowler.com/articles/microservices.html, 2014.
[15] J. P. A. Almeida, G. Guizzardi, R. A. Falbo, T. P. Sales, gUFO: A lightweight implementation of the Unified Foundational Ontology (UFO), http://purl.org/nemo/gufo, 2019.
[16] D. Schmidt, Guest editor's introduction: Model-driven engineering, Computer 39 (2006) 25–31.
[17] G. Guizzardi, T. P. Sales, J. P. A. Almeida, G. Poels, Relational contexts and conceptual model clustering, in: J. Grabis, D. Bork (Eds.), The Practice of Enterprise Modeling. PoEM 2020, volume 400, Springer, Cham, 2020, pp. 211–227.
[18] D. Jackson, Software Abstractions: logic, language, and analysis, MIT press, 2012.
[19] G. Figueiredo, A. Duchardt, M. M. Hedblom, G. Guizzardi, Breaking into pieces: An ontological approach to conceptual model complexity management, in: 12th International Conference on Research Challenges in Information Science (RCIS), IEEE, 2018, pp. 1–10.
[20] G. Guizzardi, G. Figueiredo, M. M. Hedblom, G. Poels, Ontology-based model abstraction, in: 13th International Conference on Research Challenges in Information Science (RCIS), IEEE, 2019, pp. 1–13.
[21] R. Carraretto, A Modeling Infrastructure for OntoUML, Technical Report, Ontology & Conceptual Modeling Research Group, Federal University of Espírito Santo, Brazil, 2010.