Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Ontological Model of Issue Tracking

by

Le Anh Viet Linh

Degree study program: Software Engineering and Technology

Prague, January 2019

Supervisor:

Ing. Petr Křemen, Ph.D.
Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Acknowledgements

I would like to thank my supervisor Ing. Petr Křemen, Ph.D. for supplying me with the necessary knowledge to tackle this work. Furthermore, my family, for being able to give me respite in times of stress and my friends, who help me keep my sanity together. Last but not least my counselor, where without his help I would not be where I am today.

Declaration

I hereby declare that I have compiled this dissertation using the listed literature and resources only, that my thesis has not been used to gain any other academic title and that I fully agree to my work being used for study and scientific purposes.

. . . .

Abstrakt

Mnoho issue tracking systému jsou od sebe navzájem odděleny, čímž je velké množství informací nespojeno. Role ontologie v počítačové vědě je vytvořit společné schéma, který lze sdílet mezi systémy, anotovat jejich data a spojit je tím dohromady ve snaze vytvořit velkou znalostní bázi. Tato práce se zabývá budováním takové ontologie koncepčním návrhem a vyhodnotí ji na malém datovém sadu. Výsledkem této práce je jednoduchý diagram navržený v OntoUML, který je schopen zachytit struktury issue tracking systému.

Klíčová slova: issue tracking, softwarový vývoj, konceptuální model, ontologie, ontologický model, Unified Foundational Ontology, OntoUML.

Překlad titulu: Ontologický model pro Issue Tracking

Abstract

Many issue tracking systems are disjointed from each other, rendering a great amount of information unconnected. The role of ontology in computer science is to create a common schema that can be shared among the systems, annotate their data and linking them all together in an effort to create a large knowledge base. This work will deal with employing such an ontology through conceptual design and evaluate it on a small data set of issues. The result of this work is a simple diagram designed in OntoUML that is able to capture the structures underlying issue tracking systems.

Keywords: issue tracking, software development, conceptual model, ontology, ontological model, Unified Foundational Ontology, OntoUML.

Contents

Chapter 1

Introduction

1.1 Background

Projects concerned with software development have become an activity involving a large group of people, ranging from analysts, developers, testers and possibly more. Analysts create specifications extracted through requirements from their clients, developers implement the features and quality assurance find and report potential defects in the implementations. All these steps are overseen by a project manager who ensures that the project team is focused and moving towards reaching the goals of the project. Along the way, issue tracking systems are used to organize and manage the tasks and defects that arise during the project.

Traditionally, it has been a tool used primarily by software development teams where issues involving defects have more than often been labeled and tracked as ‘bugs’. However, its prevalence has gone outside the bounds of the software community as IT projects have grown increasingly more complex. As a result, projects have become multi-faceted activities that include cross-cutting concerns beyond software related tasks. As such, many different issue tracking systems have been created covering different domains and the general understanding of an issue tracking system has evolved beyond than just being a ‘bug tracker’.

Issue tracking systems have become an integral tool in projects enabling teams to collect and focus on the tasks leading to the achievement of the project’s goal. The role of issue tracking systems has mainly been reporting, assigning, tracking, resolving and archiving issues thereby indirectly creating a knowledge base in the process. From the knowledge base, teams can query information using filters, keywords or other constraints.

1.2 Problem statement

Nowadays, countless issue tracking systems exist with a focus on software development, project management, product management and more. As many projects encompass a multitude of these aspects, the use of different issue tracking systems has become common within project teams.

Eventually, integration problems arise when the different systems need to be integrated with each other in order to create a common knowledge base. The underlying issue lies in disjointed model representation across the systems that conflict with each other rendering, for instance, cross-system querying impossible. Without a common model to rely on, the task of data integration continues to be an arduous endeavor in resolving this problem.

Take for instance the concept of 'issue'. Across many different systems its fields or properties will have varying terms: the field 'author' might not exist in another system because the same meaning is conveyed through the label 'reporter', one could make the comparison for other fields as well: 'dateCreated' vs. 'dateOfSubmission', 'label' vs. 'tag' and so on. Even the very term of 'issue' can be found to be called as 'task' in different systems. In addition to that, there are also differences in their data type signature for their fields.

Furthermore, querying upon data in an issue tracking system currently involves a set of filters for attributes of data fields or through searching through keywords within data documents. This prevents the user from a richer more complex data retrieval undermining information and relationships between data.

1.3 Goals

The goal of this work is to create and propose a model that is able to capture common structures across different types of issue tracking systems. The set of classes within the model will be enhanced with ontological conceptualization enabling their use across multiple issue tracking systems. Therefore, the resulting ontology can be reused by the system providers to decorate their existing models with a common definition, providing nigh seamless cross-system data integration and enhancing their data with semantics for a more meaningful and expressive querying.

1.4 Structure of work

The second chapter will familiarize the reader with the concepts which form the core of the solution proposed by this paper starting with an introduction to the semantic web followed by ontology. Afterwards, its application in software development will be discussed showing how it can benefit not only in the individual development process itself but also through sharing its resulting conceptual model in other software development projects. Furthermore, we will introduce an ontology called Unified foundational ontology (UFO) and OntoUML which is a version of UML that is designed to comply with the axioms laid out in UFO. The final part of this chapter will introduce tools that are used in the environment of semantic web and ontology, in particular, a CASE tool called Menthor which has been developed for creating conceptual models with accordance to OntoUML specifications, Protege for editing ontologies and GraphDB for querying semantically structured data.

The following chapter, we will describe issue tracking systems and present the concept of knowledge-based systems. Furthermore, the domains of software development and project management will be introduced as well as concrete issue tracking systems that specialize in these two domains.

The subsequent chapter will present related work regarding the problem statement and discuss their effectiveness in their resolution.

This is then followed by two main chapters of this paper, the first consisting of analysis. Here, we will draw similarities between the issue tracking systems showing that although they might have different terms and labels within their respective domains, their underlying structure is very similar. Taking advantage of this fact we will then propose a model created with OntoUML where we follow up by justifying our reasoning for the resulting model. Lastly, we will lay out the evaluation process of the model and discuss its results.

The second main chapter will deal with the evaluation of the model created in the previous chapter. It will explain the process of evaluating and integrating public data set with the model and its subsequent upload into a database where through various queries we will try to determine how well data is integrated across various systems under one model.

This paper will conclude with thoughts on the work conducted, its future prospect and potential further research for refining the solution.

Chapter 2

Core concepts and technologies

In this chapter, we introduce the core concepts and technologies used in this work. We will explore the environment and infrastructure of the semantic web with a particular focus on ontology since the fundamental part of this work deals with the application of ontological concepts.

In the end, we discuss how ontology can be incorporated during the process of software development through the use of a UML extended language called OntoUML and its associated CASE tool Menthor with further processing in a software called Protege. The resulting data set is then uploaded to a special type of database provided by GraphDB.

2.1 Semantic Web

The semantic web describes a vision for structuring and linking data on the web. Prior to the concept of the semantic web, there were already notions of links that connect different documents with each other through the use of hyperlinks. Examples of such linkages can be found on most websites but perhaps the simplest one to understand are Wikipedia pages, whose articles can contain multiple references not only to its own domain but also to external documents.

The idea behind the semantic web is to go beyond links between websites/documents and focus on linking their underlying data. The premise of this undertaking is to create structured data that can be put into a query-able data set. An example of such data set can be found in DBpedia, which has data extracted from Wikipedia infoboxes which are structured based on the labels and their values found inside the individual info boxes. The result of such structured data set provides a means for machines to process the data and even create inferences about implicit information within the data.

Another use of the semantic web is to be able to integrate different data sets from different sources together. For instance, within a company, many applications are used to manage their resources or information. More often than not are pieces of information duplicated across the different applications. For instance, an accounting system and a human resource system will have information about an employee in their respective systems causing duplicity. Therefore should information about the employee be updated, it would have to be done on two different systems.:

The rest of this section will introduce technologies of the semantic web that are used for this work: RDF, OWL, and SPARQL.

RDF

RDF, short for 'Resource Description Framework', is a framework used to describe the information about a *resource* [1]. A resource can be anything, ranging from people, things and even abstract concepts. In RDF, each resource has its own URI (Universal Resource Identifier) that can be uniquely referenced. URI is similar to URL in structure and retrieving a URI can yield additional information about the resource.

Another important notion in RDF are *statements* that have a structure of `<subject>` `<predicate>` `<object>`, called a *triple*. These triples are often visualized as graphs with both subject and object representing nodes and the predicate representing an edge that connects these two nodes. Figure 2.1 illustrates such a graph. Example triple statements, that are described by the graph are:

- `<Bob> <is a friend> <Alice>`
- `<Bob> <is interested in> <Mona Lisa>`
- `<Mona Lisa> <was created by> < Leonardo da Vinci>`

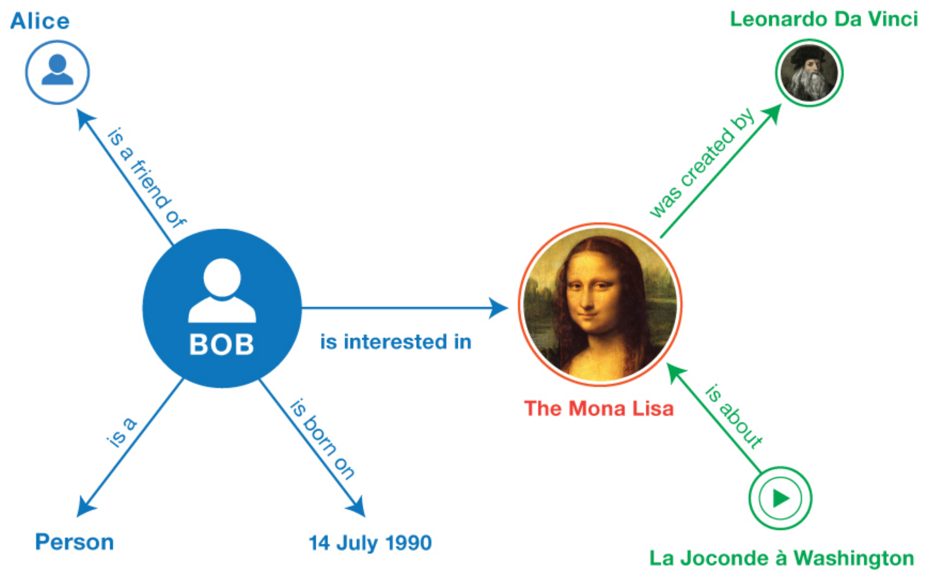


Figure 2.1: Example of triples visualized a graph taken from [1]

The triple structure can be represented in various ways, perhaps the most readable is in the format of *N-Triples* which is the exact representation as illustrated above as a sequence of subject, predicate and object. Other most common representations are in XML or in JSON-LD which is an extended version of JSON.

RDF also offers a schema called RDF Schema that provides definitions for the resources and its relationships between other resources. For instance, it can describe what domain and range a predicate can have for its subject and object respectively or whether a class of a resource is a subclass of a different class.

OWL

Similar to RDFS, OWL also contains definitions that can describe the underlying relationship between two resources but whereas RDFS offers simpler specification about the properties and relations of the resources, OWL contains further concepts for a richer expression of the resources. For example, in addition to the definitions offered by RDFS, OWL can describe data in set operations. Furthermore, OWL places more restrictions on the way data is modeled. This approach prevents the model from having ontological inconsistencies. All these benefits allow OWL modeled data to be run through a reasoner which can potentially uncover new triple statements in the data.

SPARQL

Once a data set is modeled using the aforementioned technologies, the data set can be queried through a query language called SPARQL. Similar to SQL, it is used to query data but whereas SQL is concerned with tables, SPARQL's queries are based on matching triples thus its syntax is based around the elements that comprise the triples. An example of a simple SPARQL query could look like

```
SELECT ?subject {
  ?subject foaf:knows <http://people.net/alice>
}
```

which will return records of resources that 'know' the resource identified by the URI `http://people.net/alice`. In this query, 'foaf' is a namespace for an ontology described in `http://xmlns.com/foaf/0.1/`. The concept of ontology will be described in the following section.

2.2 Ontology

The study of ontology is rooted in philosophy and is commonly described as a branch of metaphysics concerned with the nature and relations of being. This chapter will refrain from describing ontology from the philosophical standpoint and instead focus on its role in the domain of computer and information science.

In computer science, ontology is a formal representation of the knowledge through a set of concepts within a domain, the relationships between these concepts and a set of restrictions imposed upon the concepts and their relationships [2]. It is used to reason about the properties of that domain and may be used to describe the domain. Thus, in theory, ontology is often described as a "formal, explicit specification of a shared conceptualization".

A specific ontology provides a shared vocabulary of concepts for other systems, which can be used to model a domain, specifically the concepts defined within that domain, their properties and relationships along with a set of restrictions.

Ontologies are typically specified in first-order logic thus allowing them greater expressive power. For this reason, ontologies are said to be at the 'semantic' level as opposed 'logical' or 'physical' level in databases. Furthermore, their specifications allow for abstraction away from data models and implement other strategies [3].

Benefits of using ontologies

The main benefits that ontologies provide in computer-related fields are *explicit domain assumption*, *sharing of common domain understanding among people or software agents* and *reusability of domain knowledge* [4]:

- **Explicit domain assumption:** Concepts, the relationship between them and the restrictions imposed are clearly defined. This leaves little room for ambiguous interpretation of the underlying model and enables the sharing and reuse of the ontology.
- **Sharing of common domain understanding:** Sharing the knowledge about a common domain facilitates integration between systems that use the same ontology. Computer agents will be able to extract and aggregate data from the different systems connecting multiple knowledge bases together.
- **Reusability of domain knowledge:** Reusing existing domain knowledge prevents different systems from 're-inventing the wheel' and introduces common standards that allow for interoperability between systems.

Types of ontologies

Ontologies exist in many forms and can be divided in many ways. Here we will distinguish between *domain ontology* and *upper ontology.*

Domain ontology

A domain ontology represents concepts that belong to a specific part of reality such as medicine or finance. They represent a particular vocabulary of concepts that apply to their given domain. An issue with domain ontologies is that often times different people write ontologies with differing representations of concepts for the same domain. This is can arise due to different languages, different intended uses of the ontologies, and different perceptions of the domain. Merging these ontologies requires hand-tuning and/or software merging which is a time-consuming and thusly expensive endeavor. On the other hand, different domain ontologies that share the same upper ontology have can be merged with less effort.

Upper ontology

Upper ontologies aim to provide conceptualizations of very general terms, such as time, objects, events etc. that are applicable across multiple domain ontologies. They provide domain ontologies with a set of basic elements specifying the meanings of their concepts. As a result, upper ontology provides two main benefits: to bring about *semantic interoperability* and make the *ontological commitment* of a vocabulary explicit [5].

- **Semantic interoperability:** The ability for software agents to exchange data with unambiguous, shared meaning.
- **Ontological commitment:** An agreement to use the shared vocabulary in a coherent and consistent manner within a specific context.

2.3 Usage of ontology in software development

The research on ontology in computer and information related fields date back to the late 60s however only in recent years did the topic gain considerable traction in the field of software development. This is mainly due to systems becoming more data-driven and interconnected thus a greater focus of software development has shifted towards reusability and integration [6]. As we have shown, ontology promises to enable sharing and reuse of domain models thereby allowing different applications to define their models under the same structure. As a result, different applications following the same ontological structure and relationships have the potential to integrate systems seamlessly. Therefore, the importance of ontology has grown because its application facilitates these tasks by providing a shared blueprint for developing common systems. Recent software engineering paradigms such as model-driven development complement the resurgence of ontology since it provides with a rigid conceptual basis for the creation of domain models.

Stages in software development

Software development projects typically follow a set of connected steps beginning with requirements gathering and formulation followed by analysis and design. The artifacts created by these activities commonly consist of conceptual models designed by UML. These are transferred to the implementation part of the development process which is then concluded by testing and deployment.

Conceptual constraints in developing software systems

Building large software application, particularly enterprise-level systems like CRM, PMS etc., is a complex affair since the knowledge comes from different sources which are further spread across varying platforms and infrastructures. In many cases, these sources have to be integrated but that is often hindered by the different terms and labels that are annotated by the various systems. In addition, designers of software systems have different conceptual perspectives on how a system should be modeled, further causing segmentation between the resulting systems. These problems can be overcome during the analysis process of software development by reusing a shared existing ontological model that can standardize over the varying systems of similar purpose. Moreover, an additional benefit of reusing ontological models instead of creating their own during the software development is cutting down a great amount of time spent on analysis.

Furthermore, the quality of the input artifacts determines the success of the given development step [7]. In other words, improper or inaccurate artifacts will contribute to a flawed implementation of the task. In the early eighties, Barry Boehm published statistics showing that uncovering errors in later stages of the development increases the cost of resolving it. Therefore, from the standpoint of software project management, having a reliable conceptual model can prevent potential cost from emerging during the development process.

Application of ontology during the analysis phase

For these reasons, we focus on the analysis part which deals with creating a conceptual model capturing concepts used to help people know, understand, or simulate a subject the model represents. The analysis phase defines the framework and scope of the software system on the basis of the requirements gathered in the previous development process. The resulting requirements document includes UML diagrams that describe the objects in the system, their relationship with each other and actions that can be done on these objects. In general, the specifications are technology independent, as they essentially describe the ontology of the underlying system. During the design phase, these objects are mapped onto software specific items, such as lists, stacks, trees, graphs, algorithms, and data structures.

Drawbacks of using UML

Although UML is the most commonly used language for representing conceptual models, it lacks certain constructs/stereotypes due to its absence of semantic restrictions. This problem results in diminishing expressiveness when modeling concepts as many of them are simply modeled using associative relationships where the origin of the concept remains either unclear or left in ambiguity. Therefore, the resulting conceptual model described in UML can cause poor and ambiguous representation of the reality that the system is supposed to be modeled after. As such, that particular conceptual model neither captures the reality fully and faithfully nor is it reusable since its lack of semantic restrictions causes different perspectives on the implementation of the underlying model.

As conceptual models are considered important tools to achieve a common understanding of a domain, the need for a clear and unambiguous representation becomes crucial, especially for later phases of software development. Beyond the development process itself, the resulting conceptual model may also be shared for use in other systems that are concerned with similar domains facilitating the integration between these. Therefore, it is necessary to add notions and concepts of ontology when creating conceptual models as using UML by on its own falls into the risk of undermining its expressiveness.

2.4 Unified foundational ontology

Giancarlo Guizzardi created in his Ph.D. thesis 'Ontological Foundations for Structural Conceptual Models' a structure for unifying several existing foundational ontologies named Unified Foundational Ontology or UFO for short. The development and creation of UFO drew from a number of theories such as formal ontology in philosophy, cognitive science, linguistics, and philosophical logic. UFO is an upper ontology that provides specifications for domain ontologies. Furthermore, it aims to provide a common foundation for conceptual modeling based on certain heuristics that follow philosophically well-founded principles as well as capturing the ontological distinctions underlying human cognition and common sense. UFO is divided into three main categories:

- UFO-A, ontology of endurants
- UFO-B, ontology of perdurants
- UFO-C, ontology of social concepts

We focus on UFO-A, as its definitions have been completely formally characterized in the thesis and because it has been applied in various fields. Endurants are entities that are perceived as a wholly complete concept independent of snapshot or time. In contrast, perdurants are entities for which only a part of the entity can be observed given a snapshot or time. In other words, any entities that would we describe as 'processes' are perdurants. Thus, UFO-A deals with aspects of structural modeling such as types and taxonomic structures, roles, part-whole relations, particularized relational properties, and relations. These are entities that retain their substance throughout snapshots or times.

Universals, individuals and instantiations

UFO builds upon the distinction between the *universals* and the *individuals*. Individuals are described as real entities that we perceive in the world. The perception does not need to be concrete, as this also includes abstract entities. Thus in this context, 'real' is meant to be understood in the sense that they are the existential representation of an idea, a universal.

Universals feature patterns and characteristics that can be realized by multiple individuals. Thus, in contrast to individuals, universals can be thought of as an idea that individuals manifest. Universals are defined by *principle of application* and identity which will be discussed further below. Principle of application is a principle for which we can judge whether an individual is an instance of that universal [8].

Individuals can instantiate multiple universals. In this sense, they behave like multiple blueprints for the individuals to instantiate and take on. The relation between an individual and its universal(s) is called *instantiation.*

Consider an individual *John. John* can have multiple instantiations of universals such as *Person*, *Man*, and *Husband.* In this respect, a universal can also be instantiated by multiple individuals, for example, a universal *city* can be instantiated by individuals *Prague*, *Berlin* and *London.*

Identity

Another important aspect of UFO is the notion of *Identity.* The property of identity enables an individual to be exactly the one concrete individual that is different other individuals. It can be defined as the fact of being who or what an individual is [9].

Through comparing identities, we are able to determine whether two individuals are the same. For this, we need a principle on which basis we can judge whether two instances are the same. This principle is unsurprisingly called *principle of identity.* It is defined as a method that that compares through an identifier the uniqueness of an individual. An example of an identifier for a *Person* could be his name or citizen ID number.

The choice of an identifier is often a subject of debate of whether it really is able to uniquely identify an individual. Take for instance the above example with name as an identifier. Here, we run into the problem of multiple individuals having the same name.

Universals that provide an identity principle are called *Sortals* and *Non-Sortals*, commonly also referred to as *Mixins.* Examples of sortal universals are *Person*, *City* or *Student.* In contrast, a non-sortal universal is for example *Customer*, since it can be instantiated by instances of universal *Person* or by instances of universal *Company.*

Identities are important in determining the existence of an individual given its instantiations and for defining the different types of universals. For instance, *John* can move in and out of being an *Employee* without losing his identity. However, he cannot do the same with *Person*, if he ceases his instantiation with *Person*, he himself also ceases his identity.

2.5 OntoUML

UFO has been used for development and redesigning conceptual models in domain ontologies from different areas, for instance in Enterprise Modeling, Software Engineering, Service Science, Petroleum and Gas, Telecommunications, and Bioinformatics. Nonetheless, out of all the applications of UFO, the one that concerns this work the most is the design and development of an ontology-based modeling language known as OntoUML.

OntoUML is an extension to UML based on the ontology outlined in UFO-A. The classes offered by OntoUML are representations of UFO-A constructs which are then represented using UML stereotypes. These constructs carry a set of formal definitions and restrictions that encompass the given class. Therefore, it includes a system of rules and constraints capable of describing most concepts. Its language can capture a wide variety of stereotypes: types and taxonomic structures (including roles), part-whole relations, events, formal and material relations, dependent (weak) entities, attributes and attribute value and measurement spaces (roughly datatypes). As such, OntoUML offers greater expressive power than UML.

The advantage of using OntoUML over UML is that its stereotypes rooted in ontological axioms laid out by UFO. Thus in contrast to UML, OntoUML applies notions of *universals*, commonly referred to as *types*, and *individuals*. It employs basic notations found in UML, such as classes, associations, and generalizations with various ontologically defined stereotypes that in result in richer descriptions and a more defined nature of the model. Furthermore, it omits UML's simple part-whole relationships. Instead, OntoUML provides its own relationships that are able to more precisely describe the nature of part-whole relations between objects.

There are a total of twelve types that define OntoUML's *class stereotypes*. In the following sections, we will first describe the notion of *rigidity* and afterwards present the types, that have been used in this work for building an ontology for issue tracking.

Rigidity

The design of UFO and subsequently OntoUML borrows heavily from the theories of modal logic. In a metaphysical sense, UFO and OntoUML deal with the notion of *worlds* with modal operands *necessity* and *possibility*. The idea of *rigidity* stems from these operands and thus OntoUML distinguishes between two types of rigidity: *rigid and anti-rigid*:

- Rigid types: These types employ the notion of *necessity* whose instances are applied essential characteristics *necessarily*. Thus, Instances of a rigid type cannot cease to be its instance without ceasing to exist [11]. For example, we have the type *Person* and individual *John*. Throughout *John's* existence, he never ceases to be an instance of *Person*.

- Anti-rigid types: These types, on the other hand, apply the notion of *possibility*. If an individual instantiates an anti-rigid type, that particular instance can cease to exist without the individual losing existence. An example of this situation is the type *student* where an individual can stop being a *student* once he finishes school.

In essence, rigidity is a property of OntoUML's types that determines whether a characteristic of an individual can change between time periods.

Kind and Subkind

Kind is a rigid type that supplies its identity to its instances. Through the rigidity, their instances receive its identity and keep it for the duration of their existence. As such, they build the foundation for models created with OntoUML.

Subkind is a specialization of Kind that extends its ancestor by its own features. They do not define their own identity and thus they must necessarily inherit from ancestors of rigid type.

Role and Phase

Role and *Phase* are both anti-rigid types. Similar to Subkind, they do not provide identity and thus must also necessarily inherit from other types. These types are used to model their ancestors' attributes and features at some given time.

Role is used to model certain properties of individuals when it is in relation with other individuals. These two individuals are necessarily connected through an association, as the role is dependent on the individual at the other end. Consider the role *student*, one cannot become a *student* without enrolling in a *school* first. These two individuals are connected together through the type *Relator* as illustrated in Figure 2.2.

Phase is used to represent different states of its ancestor. Though the ancestor can move in and out between phases, unlike with roles, he can be only in one certain phase from a *disjoint* and *complete* generalization set. For instance, an individual cannot be in phase *sick* and phase *healthy* (modeled in 2.2) at the same time, nor in phases *young* and *old*. However, he can be both in phases *ill* and *young* if both of these phases are in different generalization sets.

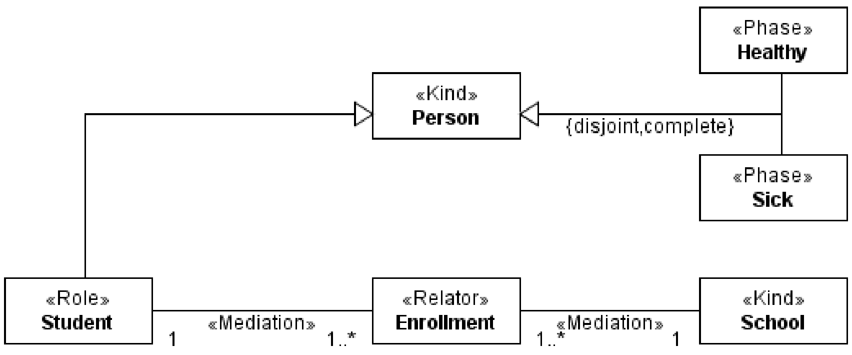


Figure 2.2: Example of an OntoUML model.

Relator

A *Relator* is a type that is used for relating two individuals together, essentially mediating between these two. Relators can be concrete such as a membership contract between a studio and a member or it can also be abstract such as a kiss or a call. Relators can connect multiple individuals. An example of such a relation is a purchase of an item by a person in a shop where purchase would be of type *Relator*.

In the section about on Role, it was mentioned that it has to be connected through and association. Relators are the association that glues the Role with another individual. In the case of *Student* and *School*, the resulting Relator could, for instance, be *Enrollment*.

2.6 Tools

In conjunction with the development of OntoUML, CASE tools for using the modeling language have also been developed. The first such tool was called OLED, standing for OntoUML lightweight editor which was then succeeded by another tool called Menthor Editor (ME) that added a few new additional features.

The process of designing diagrams is similar to what one would expect from any other tool dealing with the design of class diagrams. However, ME offers further features that complement ontology-driven conceptual modeling. At its core, it offers class stereotypes with OntoUML specifications.

Since the field of ontology deals with multiple specifications for designing ontologies, ME provides multiple code generation options. The designed diagram with OntoUML specifications can be generated in other implementations of ontology such as RDF or OWL. Furthermore, it is also able to convert the diagram to UML.

Crucially for ontology design, ME further provides validation toolkits for validating the model. Since OntoUML is rooted in the ontology of UFO, it has clearly defined axioms that are verifiable by a computer. Therefore, ME offers a syntax verification making sure that the model we are building is correct throughout the process.

Another validation toolkit is the detection of ontological anti-patterns. Though the model may be syntactically correct, it may be vulnerable to domain-related ontological misrepresentations [12]. ME is able to detect these anti-patterns and through a step-by-step process. It is worth noting that, although these anti-patterns emerge, they do not strictly need to be corrected, rather it is seen as a suggestion to take the mode 'under microscope'.

Lastly, an important validation toolkit is a visual simulation of the model. The simulation tool is able to generate multiple kinds of worlds on the basis of the created model. From the simulation, we can check the correctness of the model and identify any anomalies we have created within the model.

In addition to Menthor, this work also made use of Protege which is an open-source application for managing and editing ontologies. It was used for inspecting the created ontology and for applying the ontology to script-processed open data sets. The resulting combination was then uploaded to a graph database called GraphDB which stores triples that can be queried using SPARQL syntax.

Chapter 3

Issue tracking systems

As mentioned in the first chapter, issue tracking systems are an integral part of software development projects as they have become an undertaking involving a large group of people with various activities and sub-activities. An issue tracking system aides in keeping track of the current activities in one location that is visible to people involved in the development process. As a result, the information created on an issue tracking system is shared across the participants and thus provides an overview of the current state of the project.

At its core, an issue tracking system is a software that manages and maintains a list of issues. An issue can have several aspects to it depending on the type of issue and the domain it revolves around. In most issue tracking systems, the details of the issue entail a reporter, the date of submission, the phase the issue is in, a description of the issue, an assignee and the date of resolution.

By collecting issues throughout the software development process, the issue tracker system indirectly creates a database containing information on each participant, resolutions to common problems and other data and information. By applying an ontology upon on an issue tracking system, it enhances its functionality providing it with benefits and functions of a knowledge-based system.

3.1 Issue tracking for software development

The most common type of issue tracking system used in software development is a bug tracking system. A software bug is an error or a flaw found within a software that renders the program or system to behave in unintended ways or directly produce incorrect or unintended results. Virtually no software is bug-free, it is therefore vital to track the emerging software bugs during the course of software development in order to manage their resolution and record the solution for future references. A bug report typically contains a unique ID, severity, priority, reporter, date of submission, an assignee, date of resolution and its current status. Creating a database of software bug reports can aid in future development processes for instance by quickly resolving similar bugs by querying the database of existing software bugs.

Beyond bug tracking, issue tracking systems within software development are also used to track work activities of people involved in the development. Such software task report contains the creator of the task, date of creation, a task type/label priority, an assignee, date of completion, component part/parent task that the task is dependent on and the current status of the task. Similar to storing bug reports, having a database of activity reports also brings additional benefits, like having an overview on the progress of the project, by tracking what parts of the project are being worked on.

This work has analyzed three different issue tracking systems that are predominantly marketed towards software development:

GitLab

GitLab is a web-based Git-repository manager with an issue tracking system. GitLab boasts a clean and minimalistic UI that avoids cluttering the display with many fields and information. On the other hand, its issue tracking system is the weakest out of the three analyzed in the work, lacking customization options beyond adding user-created labels to issues.

Jira

Jira is a proprietary issue tracking system developed by Atlassian. It is highly ranked in popularity amongst issue tracking systems. Jira is offered in three different packages. The one that was analyzed for this work is Jira Software, which concerns itself with planning and tracking issues in software development. Out of the three analyzed systems, Jira is the most powerful tool, allowing its users to create virtually unlimited custom fields for their issues and customize their workflow, which also can be visualized, and other customization options. However, the powerful features come at a cost of a higher learning curve when using Jira as it is well-known from being complex.

Redmine

Redmine is an open-source web-based issue tracking system. Despite being an open-sourced application, it boasts features similar to Jira albeit with fewer customization options. Being an open-source application, the source code is public, meaning its user can directly modify the program changing code to fit their needs and requirements. However, the application needs to be installed and hosted, making the setup in comparison to other issue tracking systems more cumbersome. Its downsides are the dated user interface design that at some places can evoke an unintuitive feeling when navigating around the application.

3.2 Issue tracking for project management

Project management software typically have more general structure than issue tracking systems for the software development domain since within a project, there are multiple teams besides software. Therefore, project management systems have to account for this and provide features that are highly customizable. From large collections such as creating workspaces to smaller items like customizable and user-defined fields for various issues or tasks. Due to high degree of customization, project management software is able to track issues from multiple fields, ranging from budget management, resource allocation to team communication. However, due to the greater freedom project management system provide, their issues typically lack a standard structure which can make querying them an unruly task.

This work has analyzed two different issue tracking systems that are predominantly marketed towards project management:

Asana

Asana is a software-as-a-service designed to improve team collaboration and work management. It helps teams manage projects and tasks in one tool. Teams can create projects, assign work to teammates, specify deadlines, and communicate about tasks directly in Asana. It also includes reporting tools, file attachments, calendars, and more.

Wrike

Wrike is a project management system whose main UI consists of features that are divided into two categories: project management and team collaboration. Project management interface consists of an interactive Gantt chart, a workload view and a data table that can be customized to store project data. These features are supposed to help teams track dates and manage their assignments. Team collaboration interface contains features that are designed to facilitate and aid team communication and group decision making. As such, it includes live editor and an assignment detail UI view with discussion threads on the respective assignment that can be edited.

3.3 Knowledge-based systems

The last topic this chapter will touch on are knowledge-based systems. A knowledge-based system is a computer program that reasons and uses a knowledge base to either solve complex problems or query upon the data and information within the knowledge base. The term is broad and is used to refer to many different kinds of systems. Nevertheless, a commonality among knowledge-based systems is an attempt to represent knowledge explicitly through the use of ontologies and its rules rather than implicitly via code the way a conventional computer program does, like in a relational database. A knowledge-based system has two main sub-systems: a *knowledge base* and an *inference engine*.

- **Knowledge base:** A knowledge base is a technology similar to a database that is used to store complex structured and unstructured information used by a computer system. Such a system was then coined ’knowledge-base’ in order to distinguish its form of data storage from the more commonly known database.

In contrast to databases, a knowledge base requires structured data as opposed to tables with numbers and strings. These data would have pointers to other objects that in turn have additional pointers essentially creating a graph of interconnected information within the knowledge base.

Thus the ideal representation for a knowledge base is an ontology since it provides data with well-defined structure and relationships.

- **Inference engine:** An inference engine is a component of the system that applies logical rules to the knowledge base to deduce new information.

  The knowledge base stores facts about the world upon which the inference engine applies logical rules to the knowledge base and deduces new knowledge. This process would repeat every time there is a new fact discoveries in the knowledge base, as the new insight could trigger additional rules in the inference engine.

  Inference engines work primarily in one of two modes depending on the input of the inference: *forward chaining* and *backward chaining*. Forward chaining starts with the known facts and asserts new facts. Backward chaining starts with goals and works backward to determine what facts must be asserted so that the goals can be achieved.

Chapter 4

Existing solutions

The main problem that is being discussed in this work is the lack of a shared ontology that could be used for integrating disjoint and fragmented data from issue tracking systems over different platforms and infrastructures. However, there are little to no existing solutions that deal with specifically that problem. Instead, most of the existing solutions on the given problem deal more with leveraging the advantage of utilizing technologies in ontology for a single system rather than focus on delivering a shared ontology for common use.

For instance, [13] does describe how an issue tracking system can utilize technologies of ontology to enhance the system for automating issue tracking. There, they describe the architecture and implementation details for building such a system. Although they do present the way they have mapped their model onto the objects, it does not cover a meaningful amount of the classes that are found within an issue tracking system. Though this work is a great example of how ontology can benefit the productivity be providing ontological meaning to data, it does not propose a model whose task is to create a shared ontology for issue tracking.

Another solution that is slightly related to this problem is [14], where the main goal is to show how aggregating data from various sources benefit the issue resolving process. It conducts a mixture of qualitative analysis by conducting interviews with developers, as well as quantitative research whereby they try to find correlations between various variables such as *the number of comments in a bug report* with and without a *StackOverflow link in the bug description*. This paper ultimately shows, that using tags and links in a bug report to other sites leads to improvement of the productivity during development. Ultimately, while this paper does demonstrate the virtue of a connected knowledge base, it does not propose any solutions to get closer to that stage.

Lastly, there are have been a great increase in ontologies published online. Unfortunately, only a few target the domain of issue tracking since their scope is either too big or too small. For instance, in the work [15], an enterprise ontology is proposed and while it can have a certain use for the domain of issue tracking, the domain of enterprise is much broader which would be too excessive for employing it on a smaller domain. On the other hand, this online resource `http://oscaf.sourceforge.net/tmo.html` offers ontology only in the domain of tasks without regarding the broader infrastructure of an issue tracking system such as the notion of projects and comments that discuss the issues.

Thusly, this work is largely written without relying on possible existing solutions since there have not been any research that directly deals with employing benefits of ontological technology with regards to issue tracking systems.

Chapter 5

Ontological model for issue tracking systems

This chapter deals with the approach of creating the ontological model for issue tracking. The first part describes the process of analyzing various issue tracking systems, collecting labels and subsequently building a set of terms that will comprise the resulting vocabulary. The second part will show the mapping between the found terms and their respective OntoUML stereotypes and the resulting model diagram created in Menthor.

5.1 Analysis of issue tracking systems

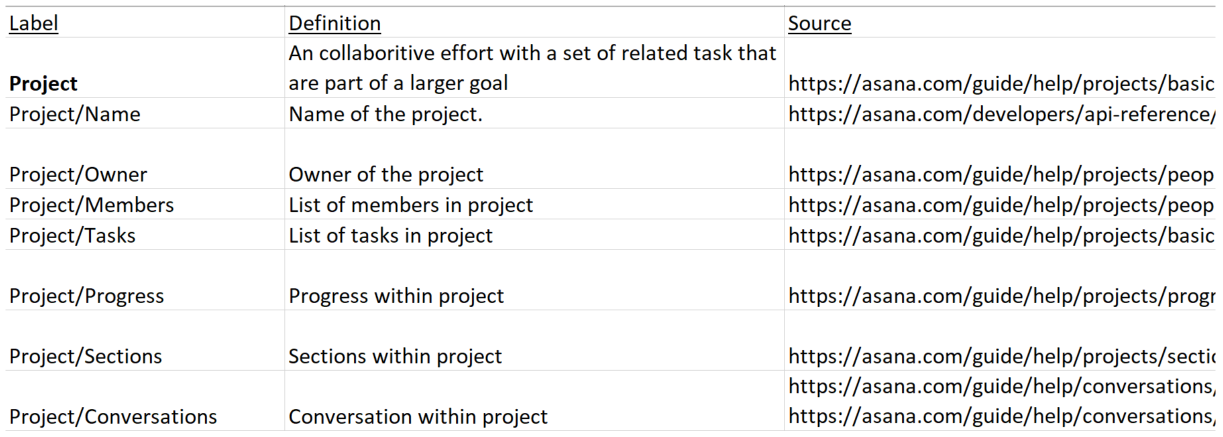
Many kinds of issue tracking domains exist, but as it was indicated in the previous chapters this work will limit itself to analyzing two domains:

- Software development
- Project management

The above domains cover a large majority of existing issue tracking systems and as such the limited scope of the domains analyzed offers a compromise between the system's market coverage and time for research. In this paper, a total of five issue tracking systems will be analyzed: three from software development domain and two from project management domain.

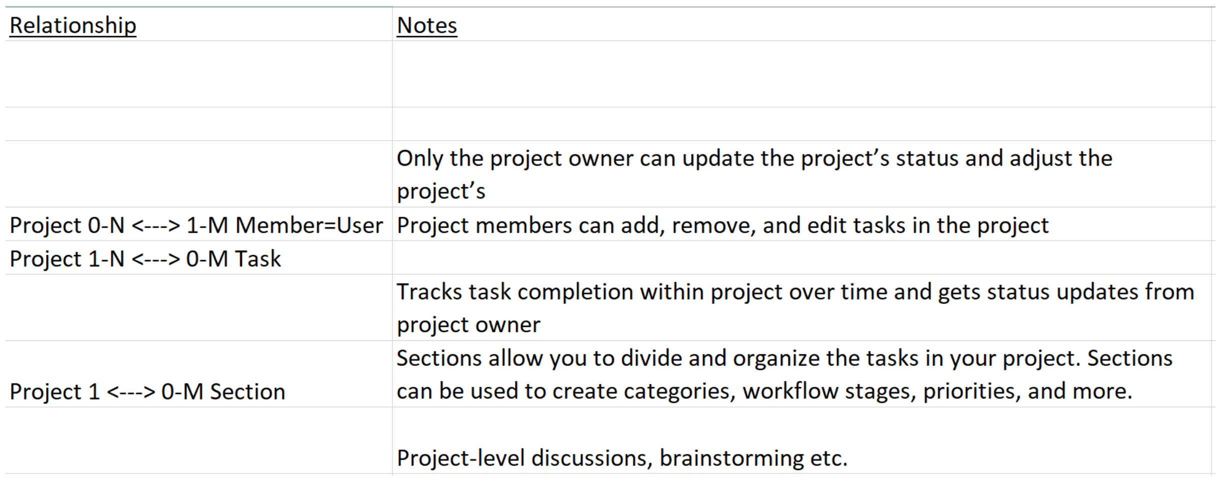
The analysis involves collecting the terms of their respective systems and model relationships between these. Subsequently, an ontological class diagram will be constructed for the different domains which will be evaluated using public data from various issue related data sets. During the evaluation process, data sets will be put through a script that will extend them with semantic meaning which will then be uploaded to a database for creating queries the resulting data.

As mentioned during the earlier parts of this work, we have analyzed five issue tracking systems from two different domains. The first part of the analysis was to research the chosen issue tracking systems and enumerate possible important concepts for the ontology of issue tracking. This was simply done by using the systems for some time, exploring and navigating through their user interface. During that process, concepts concerning issue tracking within the system were recorded in a spreadsheet document. These documents had multiple sheets each representing a **concept**. These sheets had tables where each row represented an **attribute** of the respective class and its columns held data about the attributes' *label*, *definition*, *source*, *relationship*, and *notes*.



Label	Definition	Source
Project	An collaboritive effort with a set of related task that are part of a larger goal	https://asana.com/guide/help/projects/basic
Project/Name	Name of the project.	https://asana.com/developers/api-reference/
Project/Owner	Owner of the project	https://asana.com/guide/help/projects/peop
Project/Members	List of members in project	https://asana.com/guide/help/projects/peop
Project/Tasks	List of tasks in project	https://asana.com/guide/help/projects/basic
Project/Progress	Progress within project	https://asana.com/guide/help/projects/progr
Project/Sections	Sections within project	https://asana.com/guide/help/projects/secti
Project/Conversations	Conversation within project	https://asana.com/guide/help/conversations, https://asana.com/guide/help/conversations,

Figure 5.1: Spreadsheet for 'Asana', with records for the term 'Project' (label, definition, source).



Relationship	Notes
	Only the project owner can update the project's status and adjust the project's
Project 0-N <---> 1-M Member=User	Project members can add, remove, and edit tasks in the project
Project 1-N <---> 0-M Task	
	Tracks task completion within project over time and gets status updates from project owner
Project 1 <---> 0-M Section	Sections allow you to divide and organize the tasks in your project. Sections can be used to create categories, workflow stages, priorities, and more.
	Project-level discussions, brainstorming etc.

Figure 5.2: Spreadsheet for 'Asana', with records for the term 'Project' (relationship, notes).

- **Label:** The name for the attribute. In the document, it is prefixed by the term followed by a slash. For instance, if the concept was 'Project' and the attribute was 'Name', then the resulting label would be 'Project/Name'.
- **Definition:** The definition of the label that is either defined by a source document of its respective issue tracking system or by common presumption.
- **Source:** The source from where the definition came from. In most cases during this analysis, the sources came from user manuals or documentation of its respective issue tracking system. Nonetheless, there were cases where definitions could not have been found.
- **Relationship:** If a given attribute offers multiplicity with regards to its concept, by which is mean that multiple instances of the attribute class can be contained within the concept, it will be defined under this column.
- **Notes:** Possible notes that were taken when the recording the attribute in the spreadsheet document.

Storing the records in this structured way helps keeping a reference on the concepts of the issue tracking systems that were analyzed as well as provide a way to compare the concepts and their attribute between each other. During the analysis, the following concepts were extracted from the various issue tracking systems that will form the basis for creating an ontology for issue tracking:

- **Asana:** Comment, Project, Task, User
- **Gitlab:** Comment, Issue, Participant, Project, Milestone, User
- **Jira:** Comment, Issue, Project, User, Watcher
- **Redmine:** Group, Issue, Parent task, Project, User, Watcher,
- **Wrike:** Comment, Project, Subtask, Task, User
- **User roles:** Assignee, Commentator, Reporter,

As we can see, most of these systems share the same label for their concepts. In fact, there are even concepts with differing labels that share the same semantic meaning. These occurrences can be identified by looking for concepts with similar definitions. Take the concept of 'Issue' and 'Task' for instance. If we break down their definitions taken from the sources of their respective systems, we will uncover that their concepts have a roughly similar meaning:

- Jira, Issue: *Pieces of work that must be completed.* [1]
- Wrike, Issue: *Action items that need to be completed* [2]

[1]https://www.atlassian.com/agile/tutorials/issues
[2]https://help.wrike.com/hc/en-us/articles/210322705-Building-Blocks-Tasks-Folders-and-Projects

Another way how to identify concepts with similar semantic meaning is by examining the functionalities of the concepts and their interaction with other concepts during the use of their respective issue tracking systems. Sticking with the example of 'Issue' and 'Task', in both cases submitting an issue or a creating a task was about an activity that needed to be completed. The life cycle of these particular concepts roughly follow the same simple pattern where it first begins with submitting one to the system, then assigning a person to handle the issue which eventually gets resolved.

Finally, the terms/concepts that will be used for building the model are:

Concept	**Definition**
Assignee	*User who has been assigned to resolve an issue.*
Commentator	*User who has commented on an issue.*
Parent issue	*An issue who is reliant on the resolution of other issues.*
Sub-issue	*An issue which is being reliant on by a higher, commonly more general issue.*
Subscriber	*Stakeholders of the issue that have an active interest in its completion.*
Comment	*Additional remarks or information discussing the issue.*
Issue	*A situation/activity impacting a matter (within a scope) needed to be addressed.*
Scope	*The extent of the area or subject matter that something deals with or to which it is relevant.*
User	*User of issue tracking system.*

Rationale of the selection

Absence of project class

Throughout the research on the vocabulary and terms used in issue tracking systems, the concept of a project-like structure appears often across most of these systems. These are however not included in the modelling process. The reason for it is twofold:

Firstly, the focus of this task was on modeling around the concept of 'Issue'. Creating an additional class for 'Project' would introduce unnecessary complexity of the model, such as creating further relationships with members and owners, groups and subgroups and other attributes within the project. These details do not contribute to the knowledge base of issue tracking. The concept of 'Project' serves as a way to categorize the issues within a frame of subject matter. However, that description is in conflict with the wide-spread definition of the term 'project'.

Thus secondly, the very definition of the term 'Project' has an already deeply embedded meaning for many a large audience. According to the Cambridge Dictionary, it is described as:

'A piece of planned work or an activity that is finished over a period of time and intended to achieve a particular purpose'.

Nonetheless, the issue tracking systems use the term 'Project' in a different manner than through the definition described above. For example, not all projects in the issue tracking systems deal with a temporal aspect. Furthermore, different issue tracking systems also use different terms and labels that describe the same usage of the word, for instance, Wrike additionally uses another term called 'Folder' that differs from 'Project' in that it has additional attributes to work with.

Analyzing the actual semantic meaning of these varying definitions, we suggest the term 'Scope' as a common denominator, as according to the Oxford English Dictionary, 'scope' is defined as:

'The extent of the area or subject matter that something deals with or to which it is relevant.

Presence of comments

Comments provide the issue with additional information and insight throughout its resolution process that are not described in the issue itself. Therefore it is important to include these in the modeling process since they can potentially extend the knowledge of the issue.

Sub-issues and sub-tasks

In the analyzed issue tracking systems, there were issues and tasks that were part of a greater whole. Asana, Jira, and Wrike have explicit sub-issues and subtasks, but in the other systems, the labels were rather misleading: in GitLab the greater whole was called 'Milestone', whereas in Redmine they were labeled as 'Parent task'. These labels are not explicitly suggesting a sub-issue or a subtask. However, by delving into the definition of these labels, it turns out that these systems do indeed support a hierarchical relation with issues and tasks.

This suggests that there is a common hierarchical relationship between issues/tasks among the issue trackers and therefore the relationship between issues/tasks and their smaller versions was included in the diagram.

5.2 Building relationships between terms

The next step in the modeling process was to map the terms found during the previous step onto OntoUML class stereotypes. For this part, a table was created with its columns representing the concept, its corresponding stereotype, and its definition.

Concepts that derive stand on their own, meaning they didn't derive from other concepts were stereotyped as `<<Kind>>`, therefore 'Comment', 'Issue', 'Scope' and 'User' are `<<Kind>>`.

On the other hand, concepts that share a common concept could be stereotyped in three ways: `<<Phase>>`, `<<Role>>` or `<<Subkind>>`. We can rule out `<<Subkind>>` since it is a rigid stereotype. Individuals of the remaining concepts can instantiate the other types in a particular time and can cease them to exist while the individual can still exist. Therefore, the remaining concepts have antirigid rigidity. Furthermore, since we established, that a `<<Kind>>` can only take one `<<Phase>>` at a time, that stereotype is also ruled out since 'User' can be 'Assignee', 'Commentator', 'Reporter' and 'Subscriber' all at the same time. The same goes for 'Issue' as it can both be a 'Parent issue' and 'Sub issue' at the same time. Thus, the only applicable stereotype for the rest is `<<Role>>`.

- **Kinds:** Comment, Issue, Scope, User
- **Roles:** Assignee, Commentator, Parent issue, Reporter, Sub-issue, Subscriber

Because of the existence of Role stereotypes, OntoUML requires an external entity, a *truthmaker* to make them valid. In OntoUML, that entity is defined by the type `<<Relator>>`. As mentioned in the introduction to OntoUML, Relator is a type that mediates between two other types. Thus for the model, additional concepts not found during the initial analysis process need to be added.

- **Relators:** Assignment, Commenting, Dependency, Discovery, Subscription

Concept	**Definition**
Assignment	*The process of assigning a user to an issue.*
Commenting	*The act of creating a comment.*
Dependency	*The process that produces an issue.*
Discovery	*An issue which is being reliant on by a higher, commonly more general issue.*
Subscription	*A process by which a user becomes updated on changes of a given issue.*

OntoUML Diagram

From the tables that were filled in the previous sections, we can then construct an ontological class diagram using the Menthor CASE tool. Firstly, we create class stereotypes as listed in the table and afterwards through the guidance of the ontological axioms, we fill in the relationships between these classes. The result is the model shown in Figure 5.3.

The material relation derives from the Relator to specify the name of the relationship. The derivation is indicated in the diagram by the dashed lines that go from a `<<Relator>>` to the material relation.

In common UML diagrams, the relationship between 'Parent issue' and 'Sub-issue' would be described through reflexive associations. However, OntoUML offers tools to express the distinction between what is a parent issue, sub-issue and the relationship between them by using the `<<Role>>` stereotype and describe their relationship with `<<Relator>>` stereotype. It is important to put 'Parent issue' and 'Sub-issue' under a disjoint generalization set, otherwise the same instance of issue can be its own parent issue and sub-issue at the same time.

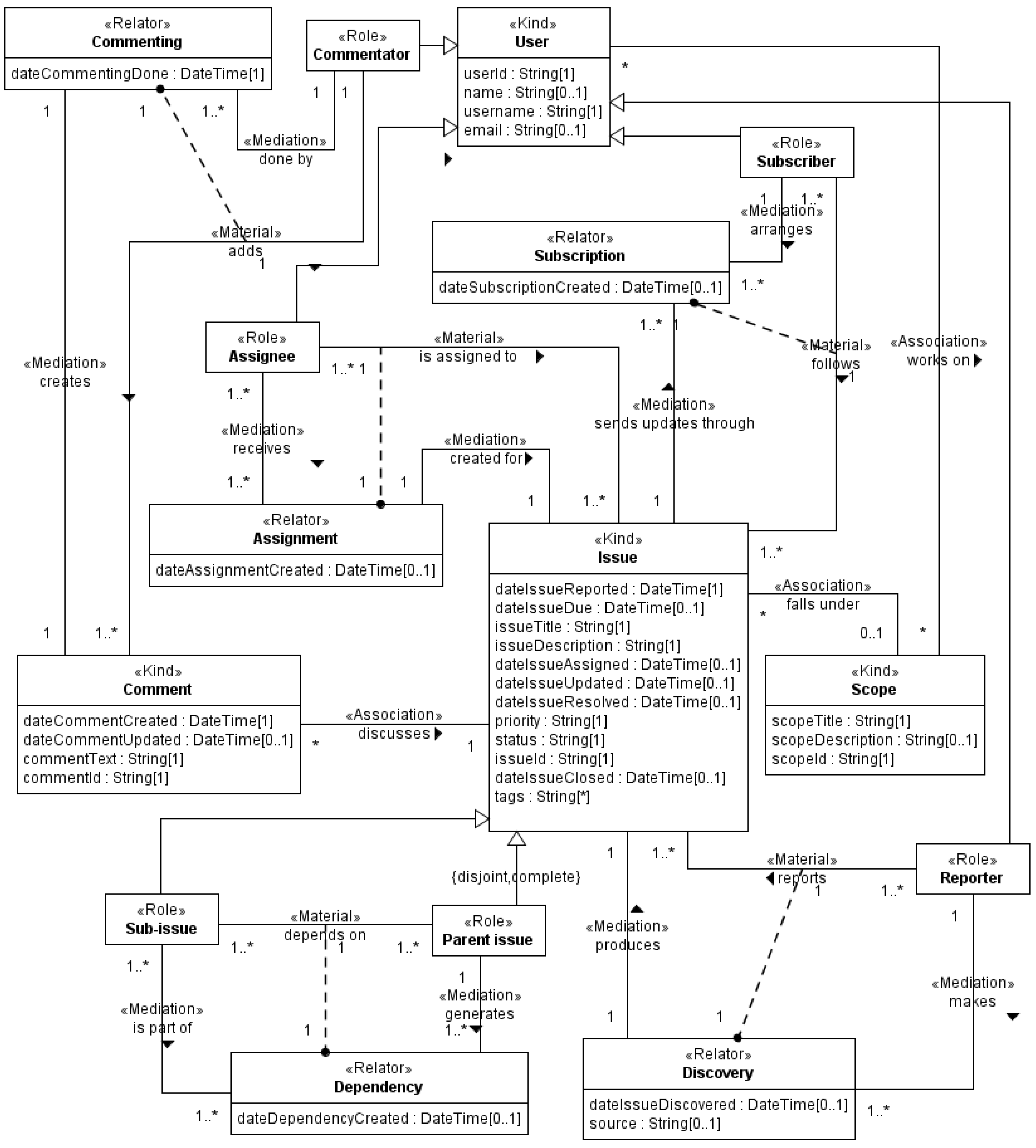


Figure 5.3: OntoUML diagram for issue tracking systems.

Chapter 6

Model evaluation

In the evaluation part of this work, we take our model through a series of validation test and applying the resulting ontology over a small dataset of issues gathered from some of the analyzed issue tracking systems. The goal of this part is to uncover what benefits such a model can bring with it and what possible deficiencies it might still have. This will give us an insight into further motivations and resolutions for future work of similar scope.

6.1 Correctness and consistency

With the built-in validation toolkit provided by Menthor Editor, the OntoUML model was analyzed for syntactical correctness and consistency. Throughout the process of designing the model, the syntax validation was used upon every change such as adding a new class stereotype are establishing relationships between these. Whenever an OntoUML rule was broken, it was fixed before we went to the next parts. As such, this proposed model almost manages to be free of any syntactical or ontological errors. The only errors found concern the multiplicity of attributes of the kinds since OntoUML doesn't allow for attributes to have multiplicity less than one, meaning we cannot have optional attributes under its rules.

Subsequently, when running the model through the anti-pattern validation enabling the detection for all types of anti-patterns, a total of 22 potential threats have been identified. When attempting to correct these errors through its step-by-step process, it did not improve the situation and instead made it conceptually harder to follow the relations in the model. Therefore for the sake of better comprehension, the anti-patterns were left unchecked.

Lastly, we let the model run through the Alloy visual simulation tool that generates multiple possible worlds constrained through the rules laid out by the underlying UFO ontology. With this component, we can detect whether desired instances behave in the manner we expect them to. In the case of this work, it detected a strange behaviour, where issues can be their own parent and sub-issues or where the sub-issue of a parent issue can be its parent issue, illustrated in 6.1. As mentioned previously, it was important to add a disjoint generalization set for parent and sub-issue to resolve this problem. An example of a consistent world can be found in Figure A.1 in Appendix A.1.

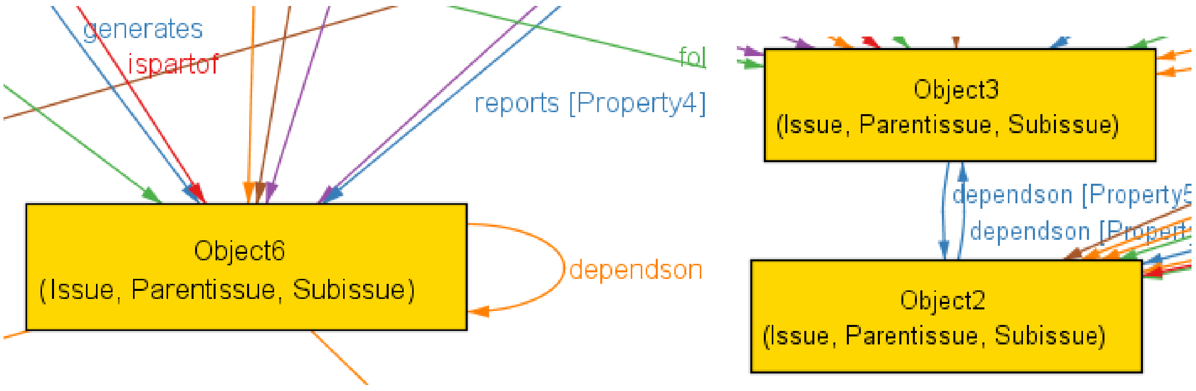


Figure 6.1: Forbidden behaviour detected by Alloy

6.2 Integration

Data set sources

For evaluating the level of integration, we fetched issues from three different sources through REST API. One of them is from Redmine, an issue tracking system for software development that was discussed in previous chapters. The two other sources come from issue tracking systems that were not analyzed for the purpose of creating the model, one of these comes from GitHub and the other, rather unconventionally, comes from StackOverflow which is a website concerning with questions and answers relating to software development. As such, although not specifically regarded as an issue tracking system, it displays certain functionalities and characteristics found in issue tracking systems. This approach of testing the model on several data sources which were not part of the analysis process shows how generalizable the model can be.

Additionally, these platforms were also chosen because they provided a public REST API for their data whereas other platforms had private projects and repositories whose data was not possible to fetch through REST API without running into authentication and authorization issues.

These are the following URL for calling the source's respective REST APIs all of which return data in the JSON format:

GitHub
URL: https://api.github.com/repos/reduxjs/redux/issues
Parameters: state=all&assignee=*
URL: https://api.github.com/repos/reduxjs/redux/issues/2244/comments

Redmine
URL: http://www.redmine.org/issues.json
Parameters: offset=60&limit=15

StackOverflow
URL: https://api.stackexchange.com/2.2/users/458193/questions
URL: https://api.stackexchange.com/2.2/users/1376441/questions
Parameters: site=stackoverflow&pagesize=7&filter=withbody

GitHub comments had to be fetched separately as they were not part of the returned data when fetching issues. It is only one set of comments for a specific issue that was fetched from the above REST API.

For StackOverflow, we collected questions from two specific users that were present in the issues in GitHub. This way we want to create a connection between the data set from two different platforms and leverage the benefits of linked data.

Transformation script

The data collected from these sources were then run through a transformation script in order enhance their JSON structure with special syntax to essentially convert them to *JSON-LD* format.

JSON-LD is one of the serialization formats for RDF that provides syntax that offers annotations for mapping JSON properties and values onto RDF concepts, like URI or triple statements. It was chosen as it was the easiest way to semantically enhance an existing JSON. The key annotations that were used in the transformation scripts were '@context', '@id', and '@type'.

- **@context:** Used to map the properties of a JSON node to a URI
- **@id:** Used to uniquely describe the data encoded as JSON node, essentially by providing a URI
- **@type:** Used to define the type of the JSON node

The role of the transformation script was to decorate the existing JSON data with these annotations and also add further properties and object that were not present in the source data set. Those were typically object properties that describe a predicate between things and *Relators*.

Additionally, the transformation script generated a schema specific to the various platforms that retained their respective labels. The schema is created via the annotation of '@context'. Appendix A.1 contains a JSON object for a Redmine issue and Appendix A.2 shows the '@context' mapping of the object properties to URI.

The way our ontology was integrated into the data sets was to add subclass to the transformation to be able to tell that a platform specific concept, created through the schema defined in their respective '@context' annotation, is a subclass of a concept in our model. For instance, a question in StackOverflow corresponds to 'Issue' in our model, we would model this relationship as `<so:Question> <owl:subClassOf> <ito:Issue>`. This will prove to be useful for querying data where for instance we either want to retrieve all issues or only issues from a certain platform.

After the data has been decorated with the annotations as well as added with supplementary objects and properties, it then was converted to an RDF format called N-QUAD using jsonld library from https://github.com/digitalbazaar/jsonld.js. This step was needed in order to open the resulting file in Protege since it cannot open JSON-LD formats.

Editing data source in Protege

The resulting transformation was then inspected in Protege for validity and correctness of the transformation. Initially, the transformation mapped all the properties of the JSON-LD objects that were not '@id' or '@type' as owl:AnnotationProperty. This had to be changed since our ontology differentiates between what is a owl:DatatypeProperty and what is a owl:ObjectProperty. Thus the transformation script had to be revised based on these findings.

When all issues regarding the transformation have been resolved, our ontology was then imported to the active ontology of the transformed data set. This process will unify the URI of our ontology stated in the transformed data set with our own ontology to provide it with further information such as the range and domain of the properties as well as additional information about the hierarchy of the types.

This combined ontology was then subsequently exported into a RDF/XML format in order to upload to GraphDB. The process from transforming the data set to uploading to GraphDB was done multiple times for each data set retrieved.

Querying in GraphDB

The last part of the evaluation process was querying upon the data set created through the previous steps. Here we want to see whether the data sets behave according to our ontology and how powerful the ontology is in querying data across different platforms. The result set of the queries can be found in Appendix C.

All reporters and their username

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ito: <http://onto.fel.cvut.cz/ontologies/issue_tracking_ontology#>

SELECT DISTINCT ?reporter ?username {
  ?reporter rdf:type ito:Reporter.
  ?reporter ito:username ?username.
}
```

This query returns user URIs and their associated username who are of type 'Reporter'. Since the prefix uses our ontology, it is able to retrieve all 'Reporter' users. If we want to retrieve 'Reporter' from specific platforms, we need to specify a different prefix that refers to the ontology of the specific platform. For instance, if we want retrieve 'Reporter' from

StackOverflow, we add `PREFIX so: <http://onto.fel.cvut.cz/issue_tracking/stackoverflow#>` and change the query to ?users rdf:type so:Owner since in StackOverflow, 'Owner' corresponds to 'Reporter' in our ontology with so:Owner being its subclass.

All issues reported by a user

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX ito: <http://onto.fel.cvut.cz/ontologies/issue_tracking_ontology#>

SELECT DISTINCT ?issue {
  ?issue ito:isreportedby ?assignee.
  ?assignee
  (owl:sameAs|^owl:sameAs)*
  <http://onto.fel.cvut.cz/issue_tracking/stackoverflow/user/458193>.
}
```

This query returns URIs of all issues that are reported by a user that is identified by URI `<http://onto.fel.cvut.cz/issue_tracking/stackoverflow/user/458193>`. In addition to that, it also returns all issues that have been made by him but under a different URI. Therefore if that user had an instance in a different platform that reported an issue, it will be shown the result set.

RM and SO 'issues' reported between 01.01.2014 and 28.10.2018 and their reporter's username

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ito: <http://onto.fel.cvut.cz/ontologies/issue_tracking_ontology#>
PREFIX rm: <http://onto.fel.cvut.cz/issue_tracking/redmine#>
PREFIX so: <http://onto.fel.cvut.cz/issue_tracking/stackoverflow#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?issue ?dateIssueReported ?username {
  ?issue rdf:type ?type.
  ?issue ito:dateIssueReported ?dateIssueReported.
  ?issue ito:isreportedby ?reporter.
  ?reporter ito:username ?username.
  FILTER(
    (?dateIssueReported < "2018-10-28"^^xsd:dateTime) &&
    (?dateIssueReported > "2014-01-01"^^xsd:dateTime)
  ).
  VALUES (?type) { (rm:Issue) (so:Question) }.
}
```

This query is meant to illustrate the degree of complexity for retrieving specific records from a triple database. It essentially returns URIs all Redmine issues that have been reported before November 2018. Through the VALUES keyword from SPARQL, we can extend the query to return issues from other platforms. Of course, if we desire to query all the available issues instead of the platform specific one, we specify the object of rdf:type to be ito:Issue which is the superclass for rm:Issue and so:Question.

6.3 Assessment

The quality of the data set depends on the richness of expression the underlying ontology can provide. The simplicity of the ontology lends itself to cover many data sets concerning issues. It succeeds in annotating data in conventional issue tracking systems but beyond that, through this evaluation process we were also able to partially cover the platform StackOverflow that although not regarded as an issue tracking system, shares similar data structure to them.

However, because of its simplicity, compromises had to be done. This ontology lacks a vocabulary for issue priority and status types, therefore it is not able to gather different terms for status or priority types under a common, unifying term. For instance, it will describe an issue status type as 'Fixed' or 'Done' differently even if they semantically could mean the same. This is also an issue for creating a common term for the multitude of tags that can be put on issues. Nevertheless, the problem with creating all-encompassing term for status and priority types as well as tags is with issue tracking systems whose administrators can create custom labels and have their semantic meaning that is not documented publicly.

Another such compromise was the omission of the notion of comment threads with replies or sub-comments which can produce less information about discussions regarding an issue. For common issue tracking systems, this would not be a large problem, at most impacting the flow of comments. However, the lack of such concepts creates a structural issue when modeling StackOverflow data since its 'answers' to a 'Question' can be interpreted as comments. However, these 'answers' have comments on their own. Therefore this ontology would encounter difficulties when adding the 'answers' portion to the StackOverflow data set.

Furthermore, the testing playground was also very limited due to inaccessibility to more different platforms to test the ontology. During the analysis of issue tracking systems, some of their data displayed interesting structure that would perhaps challenge the implementation of the transformation script and thus indirectly the ontology itself.

Chapter 7

Conclusion

The field of ontology has a promising future in front of it. As the society becomes more data-driven, the need for structured, machine-readable and machine understood data is increasing evermore. By creating an ontological model for issue tracking, we are able to connect the fragmented data all over the net together and construct a knowledge base whose information and knowledge hidden in the web of data left to be uncovered.

This work examined the current situation status of disjointed systems over various platforms and described how applying an ontological blueprint could remedy this problem. We elaborated upon the significance of ontology in software development and how it can create connections between data spread all over the world. We then proceeded onto describing the ways with which we can build such a blueprint. The environment used to create the model provides a great conceptualization framework and toolkits upon which we can model. With this knowledge at hand, the result of this work is a simple ontological model for issue tracking that is able to capture a decent amount of information through a small data set.

Future work on the topic would be refining the OntoUML model to capture more similarities between different structures whilst trying to avoid eventual sacrifices in generalization. As discussed in the assessment of the model, the drawback of the created model is its simplicity. Whilst it is able to annotate a data from a wide variety of differing platforms, it is not able to crate more complex and perhaps interesting links and inferences about the underlying data set. The set of all possible information on issue tracking remains yet to be fully uncovered. By making this ontology accessible to the public, people from different backgrounds can collaborate, offer inputs and share their knowledge for refining the ontology.

Finally, the generated OWL ontology from the model needs to be uploaded onto a public domain, so that potential domain designers can start annotate their data with the ontology, a step towards a data-connected infrastructure.

Bibliography

[1] https://www.w3.org/TR/rdf11-primer/

[2] Diana Man. Ontologies in Computer Science. 2013.

[3] Tom Gruber. Encyclopedia of Database Systems. 2009.

[4] Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. Stanford, 2001.

[5] Robert Hoehndorf. What is an upper level ontology? 2010.

[6] Wolfgang Hesse, Ontologies in the Software Engineering process. 2005.

[7] R. Pergl, D. Buchtela, Z. Rybola, I Ryant. OntoUML a UFO-A pro softwarové inženýrství. 2012.

[8] Giancarlo Guizzardi, Gerd Wagner. Using the Unified Foundational Ontology (UFO) as a Foundation for General Conceptual Modeling Languages. 2010.

[9] U.S. Department of Defense. Data Modeling Guide (DMG) for an Enterprise Logical Data Model (ELDM). 2011.

[10] Giancarlo Guizzardi, Gerd Wagner. A Unified Foundational Ontology and some Applications of it in Business Modeling. 2004. Engineering by Combining Ontology Patterns. 2015.

[11] Fabiano B. Ruy, Cássio C. Reginato, Victor A. Santos, Ricardo A. Falbo, Giancarlo Guizzardi. Ontology

[12] Tiago Prince Sales, Giancarlo Guizzardi. Anti-patterns in Ontology-driven Conceptual Modeling: The Case of Role Modeling in OntoUML. 2016.

[13] Habes Alkhraisat. Issue Tracking System based on Ontology and Semantic Similarity Computation. AL salt, Jordan, 2016

[14] Denzil Correa, Ashish Sureka. Integrating Issue Tracking Systems with Community-Based Question and Answering Websites. New Delhi, India, 2013.

[15] Mike Uschold, Martin King, Stuart Moralee, Yannis Zorgios, The Enterprise Ontology. 1997.

[16] Giancarlo Guizzardi. Ontological Foundations for Structural Conceptual Models. University of Twente, 2005.

Appendix **A**

Alloy visual simulation

A.1 Consistent world generated by Alloy

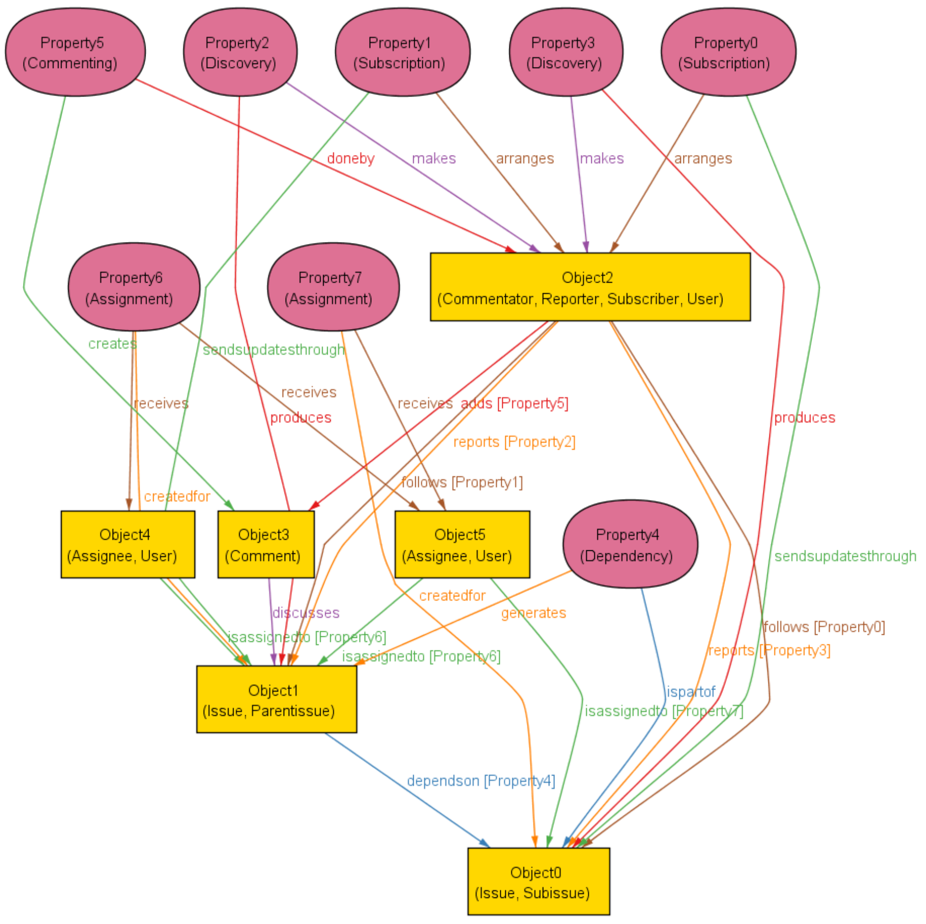


Figure A.1: Consistent world generated by Alloy.

Appendix B

Example JSON-LD mapping

B.1 Redmine issue in JSON

```
{
  "id": 29889,
  "project": { "id": 1, "name": "Redmine" },
  "tracker": { "id": 2, "name": "Feature" },
  "status": { "id": 1, "name": "New" },
  "priority": { "id": 4, "name": "Normal" },
  "author": { "id": 107353, "name": "Marius BALTEANU" },
  "assigned_to": { "id": 107353, "name": "Marius BALTEANU" },
  "category": { "id": 10, "name": "UI" },
  "subject": ...,
  "description": ..,,
  "created_on": "2018-10-31T21:28:33Z",
  "updated_on": "2018-11-05T05:14:22Z"
}
```

B.2 JSON-LD @context mapping for Redmine issue

```
{
  "@context": {
    "id": "<base URI>/redmine/issue#id",
    "subject": "<base URI>/redmine/issue#subject",
    "category": "<base URI>/redmine/issue#category",
    "status": "<base URI>/redmine/issue#status",
    "priority": "<base URI>/redmine/issue#priority",
    "created_on": {
      "@id": "<base URI>/redmine/issue#created_on",
      "@type": "<base URI>/XMLSchema#dateTime"
    },
    "updated_on": {
      "@id": "<base URI>/redmine/issue#updated_on",
      "@type": "http://www.w3.org/2001/XMLSchema#dateTime"
    },
    "description": "<base URI>/redmine/issue#description",
    "Issue": "<base URI>/redmine#Issue",
    "coveredby": "<base URI>/redmine/issue#coveredby",
    "createdthrough": "<base URI>/redmine/issue#createdthrough
       ",
    "fallsunder": "<base URI>/redmine/issue#fallsunder",
    "isreportedby": "<base URI>/redmine/issue#isreportedby",
    "producedby": "<base URI>/redmine/issue#producedby"
  }
}
```

The last six JSON properties had to be added to the original JSON data and since some of these properties rely on the existence of Relators, these also had to be created to generate a valid JSON-LD format that reflects the created ontology.

The structure for the URI differ depending on the whether the JSON property encodes a type of a thing or whether it encodes its properties. Type here means whether the JSON object is an 'Issue', 'User' etc., essentially representing the Kinds and to a certain extent the Roles of our model.

- **Types:** `<base URI>/<platform>#<Type>`
- **Property:** `<base URI>/<platform>/<type>#<property>`
- **Instances:** `<base URI>/<platform>/<type>/<ID value>`

Appendix C

Result set of SPARQL queries

These are the results sets of the SPARQL queries shown in chapter 6. The dataset and further queries can be found in the attached CD.

Namespaces[1]**:**

- gh_instance: <http://onto.fel.cvut.cz/issue_tracking/github/>
- rm_instance: <http://onto.fel.cvut.cz/issue_tracking/redmine/>
- so_instance: <http://onto.fel.cvut.cz/issue_tracking/stackoverflow/>

C.1 All reporters and their username

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ito: <http://onto.fel.cvut.cz/ontologies/issue_tracking_ontology#>

SELECT ?reporter ?username {
  ?reporter rdf:type ito:Reporter.
  ?reporter ito:username ?username.
}
```

	assignee	**username**
1	gh_instance:user/1128784	markerikson
2	gh_instance:user/810438	gaearon
...		
9	rm_instance:user/107353	Marius BALTEANU
10	rm_instance:user/352642	Cuong Nguyen
...		
21	so_instance:user/458193	Dan Abramov
22	so_instance:user/1376441	Johannes Lumpe

[1] *the namespacing is not proper* PREFIX *syntax of SPARQL, as it cannot resolve paths. It is used here for shortening the URI of the result sets.*

C.2 All issues reported by a user

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX ito: <http://onto.fel.cvut.cz/ontologies/issue_tracking_ontology#>

SELECT ?issue {
  ?issue ito:isreportedby ?reporter.
  ?reporter (owl:sameAs|^owl:sameAs)* <so_instance:user/458193>.
}
```

	issue
1	gh_instance:issue/208087614
2	gh_instance:issue/84239257
3	gh_instance:issue/85098222
4	gh_instance:issue/88862407
5	gh_instance:issue/89385026
6	gh_instance:issue/95917263
7	so_instance:question/12939321
8	so_instance:question/22501428
9	so_instance:question/24581873
10	so_instance:question/26566317
11	so_instance:question/6331075
12	so_instance:question/7718684
13	so_instance:question/9245030

C.3 RM and SO 'issues' reported between 01.01.2014 and 28.10.2018 and their reporter's username

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ito: <http://onto.fel.cvut.cz/ontologies/issue_tracking_ontology#>
PREFIX rm: <http://onto.fel.cvut.cz/issue_tracking/redmine#>
PREFIX so: <http://onto.fel.cvut.cz/issue_tracking/stackoverflow#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT ?issue ?dateIssueReported ?username {
  ?issue rdf:type ?type.
  ?issue ito:dateIssueReported ?dateIssueReported.
  ?issue ito:isreportedby ?reporter.
  ?reporter ito:username ?username.
  FILTER(
    (?dateIssueReported < "2018-10-28"^^xsd:dateTime) &&
    (?dateIssueReported > "2014-01-01"^^xsd:dateTime)
  ).
  VALUES (?type) { (rm:Issue) (so:Question) }.
}
```

	issue	**dateIssueReported**	**username**
1	`rm_instance:issue/29855`	"2018-10-27T06:46:28Z"	Yutaka Hara
2	`rm_instance:issue/29849`	"2018-10-26T02:00:45Z"	Cuong Nguyen
3	`rm_instance:issue/29853`	"2018-10-26T18:47:11Z"	Vitaly Krivenko
4	`so_instance:question/22501428`	"2014-03-19T09:32:09.000Z"	Dan Abramov
5	`so_instance:question/24581873`	"2014-07-05T00:13:56.000Z"	Dan Abramov
6	`so_instance:question/26566317`	"2014-10-25T19:22:49.000Z"	Dan Abramov

Appendix D

Attachments

D.1 CD

The attached CD contains files that were created and used during this work. The files are mainly different representations of the ontologies that were generated during the course of this work. These files can be opened in an ontology editor such as Protege. Additionally RDF files can be uploaded to a triple store where they can be queried upon using SPARQL. Sample SPARQL queries can also be found in the CD.