UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

CENTRO TECNOLÓGICO

COLEGIADO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO

Fernando Amaral Musso

An OntoUML 2.0 to Alloy transformation for the OntoUML Server

Vitória, ES

2021

Fernando Amaral Musso

An OntoUML 2.0 to Alloy transformation for the OntoUML Server

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

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Colegiado do Curso de Ciência da Computação

Supervisor: João Paulo Andrade Almeida

Vitória, ES

2021

Fernando Amaral Musso
An OntoUML 2.0 to Alloy transformation for the OntoUML Server/ Fernando Amaral Musso. – Vitória, ES, 2021-
56 p. : il. (algumas color.) ; 30 cm.

Supervisor: João Paulo Andrade Almeida

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Colegiado do Curso de Ciência da Computação, 2021.

1. Modelagem Conceitual Baseada em Ontologias. I. Musso, Fernando Amaral. II. Universidade Federal do Espírito Santo. III. An OntoUML 2.0 to Alloy transformation for the OntoUML Server

CDU 02:141:005.7

Fernando Amaral Musso

An OntoUML 2.0 to Alloy transformation for the OntoUML Server

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

Trabalho aprovado. Vitória, ES, 20 de maio de 2021:

João Paulo Andrade Almeida
Orientador

Vítor Estêvão Silva Souza
Universidade Federal do Espírito Santo

Tiago Prince Sales
Free University of Bozen-Bolzano, Itália

Vitória, ES
2021

Acknowledgements

First, I would like to thank God for making all of this possible and giving me the strength and motivation not to give up, despite all the hardships.

Next, I would like to thank professor João Paulo for his patience and tranquility with me for the entire duration of this project and previous Scientific Initiation works. I definitely learned a lot being so close to you and your research.

I would also like to thank my parents Aparecida and Guilherme for always giving me the best education and believing in me. You are my role models and I love you very much.

I would also like to thank my brother Alexandre for being always supportive and being a friend I know I can always rely on.

I would also like to thank my entire family for always supporting me and being so united. I thank especially my grandparents João (*in memoriam*) and Elízia, Fernando and Ilse, my godmother and tennis partner Cristina. Also Arthur, for all the inspiration I get from you.

I would also like to thank my dog Vida. Of course I could not leave her out, given all the emotional support and company she has given me throughout my entire graduation and life as a whole.

I would also like to thank my fellow UFES friends. Eric (and Laís), Dhiego, Mariana, Pedro, Lucas Tassis, Lucas Ribeiro, Eduardo, Breno, Ivo and Leo. Without you, this academic journey would have been much tougher and boring. I am grateful of everything we learned and accomplished together, and will miss our hours-long Discord calls the night before important tests.

I would also like to thank all my online friends who have made my life more joyful. Kara, Colin, Stephanie, Steven, Jasmine, Shannon, Ailea, Tim, Ryan and “Catnip”. Thank you for all the routine breaks, game nights, movie nights and often just random chatter and memes.

I would also like to thank my other friends Júlia, Matheus and Renato. Despite not being with me at university, you have all been inspirations for me and life is definitely much better being your friend.

Lastly, I thank all the friends I have made along the way. Those who helped me become who I am today. Those who might not be as close anymore, but our memories together will stay with me forever. Vanessa and Talib, Gustavo and Guilherme, Paulo, Henrique, Antonio, Guti and Valentina, Rodrigo and Glória.

Abstract

We navigate the world through concepts that relate to each other and form a complex network of relationships. The study of a domain includes studying the concepts that belong to it and formalizing a conceptualization to be shared with others that may benefit from it. The artifact that results from this effort is called an “ontology”, a formal, explicit specification of a shared conceptualization.

With ontology-driven conceptual modeling, modelers are able to design formal models to represent conceptualizations that aid the comprehension of a domain and enable a common understanding of it by different stakeholders. With this intention, the Unified Foundational Ontology (UFO) was developed, along with OntoUML, an ontologically well-founded version of UML 2.0 class diagrams.

UFO and OntoUML have gone through several updates over the years and OntoUML 2.0 emerged as the latest version of the language. Thereby, frameworks and other tools designed for OntoUML became outdated. New projects developed to work with OntoUML 2.0 models, such as the OntoUML Server, are eager to be able to utilize such tools, but require up-to-date functionalities.

With this in mind, this work presents an updated model transformation of OntoUML 2.0 to Alloy, to be provided as a service offered by the OntoUML Server parent project. This service will allow model verification and model validation for OntoUML 2.0 models with the intention of aiding modelers in the creation of higher-quality models.

Keywords: Ontology-Driven Conceptual Modeling. Transformation. UFO. OntoUML. Alloy.

Resumo

Navegamos pelo mundo por meio de conceitos que se relacionam entre si e formam uma rede complexa de relacionamentos. O estudo de um domínio inclui o estudo dos conceitos que pertencem a ele e a formalização de uma conceituação, a ser compartilhada com outros que podem se beneficiar dela. O artefato que resulta desse esforço é chamado de "ontologia", uma especificação explícita e formal de uma conceituação compartilhada.

Com a modelagem conceitual baseada em ontologias, modeladores são capazes de projetar modelos formais para representar conceituações que auxiliam a compreensão de um domínio e permitem um entendimento mútuo dele por diferentes partes interessadas. Com essa intenção, a Unified Foundational Ontology (UFO) foi desenvolvida, juntamente com OntoUML, uma versão ontologicamente bem-fundamentada de diagramas de classe da UML 2.0.

UFO e OntoUML passaram por diversas atualizações ao longo dos anos e OntoUML 2.0 surgiu como a versão mais recente da linguagem. Com isso, *frameworks* e outras ferramentas desenvolvidas para OntoUML se tornaram desatualizadas. Novos projetos desenvolvidos para trabalhar com modelos OntoUML 2.0, como o OntoUML Server, estão ansiosos para utilizar essas ferramentas, mas requerem funcionalidades atualizadas.

Com isso em mente, este trabalho apresenta uma transformação atualizada de modelos OntoUML 2.0 para Alloy, a ser disponibilizada como um serviço oferecido pelo projeto pai OntoUML Server. Esse serviço permitirá a verificação de modelos e a validação de modelos OntoUML 2.0, com o objetivo de assistir modeladores na criação de modelos de maior qualidade.

Palavras-chave: Modelagem Conceitual Baseada em Ontologias. Transformação. UFO. OntoUML. Alloy.

List of Figures

List of Tables

List of abbreviations and acronyms

NEMO	Núcleo de Estudos em Modelagem Conceitual e Ontologias
ML2	Multi-Level Modeling Language
OCL	Object Constraint Language
UML	Unified Modeling Language
UFO	Unified Foundational Ontology

Contents

1 Introduction

In general terms, conceptual modeling has been characterized as "the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication" (MYLOPOULOS, 1992). Under this view, conceptual models are fundamental artifacts to the software development process and are useful to establish a common knowledge of the domain between different stakeholders. Furthermore, a good comprehension of the domain, accompanied by a systematic formalization, grants great support for problem solving in various fields, such as software engineering and project management.

There has been a growing interest in the use of foundational ontologies to design and evaluate conceptual models, in a branch named ontology-driven conceptual modeling (GUIZZARDI, 2005). This approach improves traditional techniques by taking into consideration ontological properties inherited from a foundational ontology, such as rigidity, identity and dependence. In this context, the scientific community has contributed for more than a decade now to the development of the Unified Foundational Ontology (UFO), and the associated OntoUML language (GUIZZARDI, 2005). OntoUML is a UFO-based profile for UML 2.0 class diagrams (GUIZZARDI, 2005) that makes it possible to enhance models by adding stereotypes and meta-attributes to model elements such as classes and associations, capturing ontological properties from UFO. OntoUML is regularly updated and currently a 2.0 version was released (GUIZZARDI et al., 2018).

The construction of high-quality models is challenging, however. The modeler must be capable of expressing the desired conceptualization with precision, to avoid ambiguity and inconsistencies. More specifically, the model must allow the description of all admissible scenarios and inhibit the inadmissible scenarios, according to the original conceptualization (GUIZZARDI, 2005). For this purpose, model verification and model validation are powerful techniques that can assist the modeler in acquiring confidence concerning the quality of produced models. Model verification can be achieved by providing model checking tools and model validation can be done through model simulation. The results of the application of these techniques can be interpreted by the modeler in search for unwanted behavior in their designed models.

1.1 Motivation

This research project was thought out as an important service to be included in the OntoUML Server[1], a web API designed to expose functionalities for OntoUML models (e.g. au-

[1] Available at: <https://github.com/OntoUML/ontouml-server>

tomatic syntax verification and model transformation). The development team of the OntoUML Server consists of a number of former NEMO members, currently at the Free University of Bozen-Bolzano, in Italy.

The need of a model validation service to the OntoUML Server, through the development of a model transformation, gave birth to this project. The target language chosen for this task is Alloy (JACKSON, 2012), a logic-based language that allows the creation of simulations for model instances. With its powerful tool, *Alloy Analyzer*, it is possible to analyze instances of the model in order to verify behavior and assess aspects of the model, such as underconstraining and overconstraining.

There are many works regarding OntoUML to Alloy transformations, such as (BRAGA et al., 2010), (BENEVIDES et al., 2011) and (SALES, 2014). All of these, however, are currently outdated when considering the new OntoUML 2.0 elements and ontological properties. This work builds on these previous efforts, updating what is required. The design of the transformation is particularly influenced by Sales's M.Sc. Thesis (SALES, 2014).

Moreover, two other works were done in this same context of model transformation and validation using Alloy, prior to this project. Both works were Scientific Initiation projects concerning ML2, a multi-level conceptual modeling language (FONSECA, 2017). In the first, a transformation of ML2 to Alloy was developed. And subsequently, an extension to the language using OCL (WARMER; KLEPPE, 2003) was proposed and included to the transformation. Therefore, this graduation project is also a means of consolidating all academic work done during the undergraduate course.

1.2 Goals

The main goal of this research project is to develop a model transformation service to be included in the OntoUML Server parent project. This transformation should offer modelers a validation functionality that aids them in the creation of higher quality OntoUML 2.0 models, without requiring any additional learning of technologies.

The plan to achieve the main goal consists of the following specific goals:

- Revisit pre-existing transformation approaches for OntoUML to gather ideas and potentially reuse useful patterns;
- Define a set of updated transformation rules;
- Develop the transformation service in a programming language;
- Demonstrate the applicability of the transformation service through an example with simulation scenarios.

1.3 Research Methodology

The approach used to achieve the goal of this project consists in the following activities (grouped in three parts):

- Part 1: Literature Review.
  - Activity 1.1: Study of the ontologically well-founded language OntoUML 2.0 (FONSECA et al., 2019) (GUIZZARDI et al., 2021).
  - Activity 1.2: Study of the formal logic-based language Alloy (JACKSON, 2012).
  - Activity 1.3: Study and review of pre-existing transformation approaches for OntoUML, such as (BRAGA et al., 2010), (BENEVIDES et al., 2011) and (SALES, 2014).
- Part 2: Model Transformation Project.
  - Activity 2.1: Specification of the OntoUML 2.0 to Alloy transformation rules.
  - Activity 2.2: Implementation of the transformation using the Typescript programming language.
- Part 3: Testing and Results Evaluation.
  - Activity 3.1: Evaluation of the correctness and the feasibility of the approach and tools used.
  - Activity 3.2: Inclusion of the transformation as a new service offered by the OntoUML Server.

1.4 Monograph Structure

The remainder of this monograph is organized as follows:

- Chapter 2 presents the ontological theory used in the course of this work and the target language of the transformation;
- Chapter 3 presents the transformation rules and design decisions used in its implementation;
- Chapter 4 presents the transformation results and model simulation scenarios;
- Chapter 5 discusses some aspects of the implementation;
- Chapter 6 presents the conclusions and final considerations of this project.

2 Theoretical Foundations

2.1 The Unified Foundational Ontology (UFO)

The Unified Foundational Ontology (UFO) (GUIZZARDI, 2005) is constituted by three main parts: UFO-A, an ontology of objects (endurants) (GUIZZARDI, 2005); UFO-B, an ontology of events (perdurants) (GUIZZARDI et al., 2013); and UFO-C, an ontology of social entities built on the top of UFO-A and UFO-B (GUIZZARDI; FALBO; GUIZZARDI, 2008).

UFO defines a fundamental distinction of elements between universals and individuals. Universals are abstract patterns of features that describe a group of different individuals. Individuals are instances that exist in the real world, extend in time and possess a unique identity. To illustrate, Organization is an example of a universal. Instances of Organization, such as The Coca-Cola Company and Pepsico, are individuals. Note that both share the same characteristics of being an Organization, but they also have their own particularities, such as different names, date of foundation, owner, employees, and so on.

Endurants represent object-like entities, and are divided in two groups: substantials and moments. Substantials are existentially independent endurants, such as Person, Car and Organization. Moments are existentially dependent endurants, such as a person's weight and a car's color. In other words, moment individuals can only exist in other endurants.

Substantials are further divided into two subgroups: sortals and non-sortals. Sortal substantials are those whose instances carry a uniform principle of identity, such as Person, Car and Organization. Non-sortals are those whose instances can obey different principles of identity, such as Furniture and Works of Art.

Furthermore, universals can be classified according to a rigidity ontological property. Rigid universals are those whose instances cannot cease to be without ceasing to exist. An Organization, for instance, cannot stop being an Organization without ceasing to exist entirely. Anti-rigid universals are those whose instances can move in and out of their extension. To illustrate, a Person can become an Employee, and later on cease being an Employee, without ceasing to exist. Other examples are: Child, Adult, Student, Husband and Wife. Finally, semi-rigid universals apply rigidly to some instances and anti-rigidly to others.

In summary, UFO-A's taxonomy is composed of all these endurant distinctions, as shown in Figure 1. The leaves of this hierarchy will be discussed in detail in the next section, as they are the basis for the modeling constructs of OntoUML.

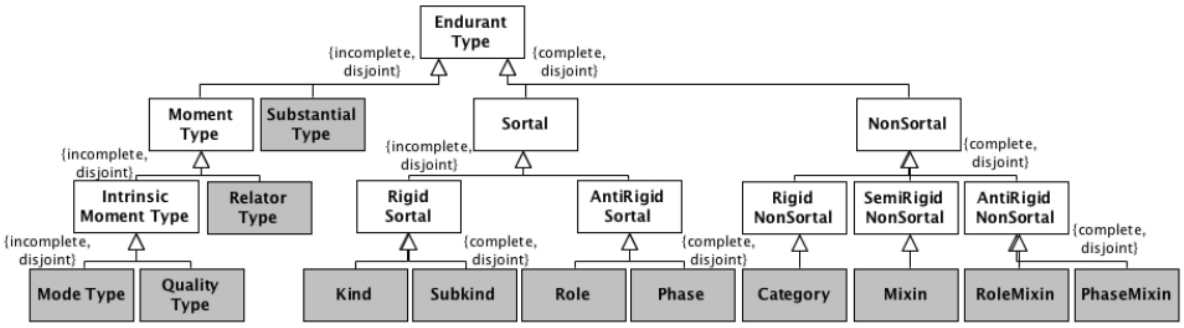

Figure 1 – UFO-A's hierarchy of endurants.

2.2 OntoUML 2.0

OntoUML is a UFO-based profile for UML 2.0 class diagrams (GUIZZARDI, 2005). As a modeling language, it provides to the modeler a number of stereotypes that can be used to decorate classes and relations, as well as a set of meta-properties to enhance the semantics of model elements.

OntoUML has gone through several extensions and updates since its conception, giving support to new stereotypes and deprecating others. Moreover, it was also extended to support the classification of moment universals by rigidity and sortality (to be discussed later in this section) and to include the taxonomy of events from UFO-B. The consolidation of this evolution is called OntoUML 2.0 (GUIZZARDI et al., 2018). This research project will focus on the changes in stereotypes and other ontological properties concerning the structural aspects corresponding to UFO-A, leaving the inclusion of events from UFO-B to future work.

This section will present the various modeling options of OntoUML 2.0, the syntax and how the language captures the ontological properties of UFO. To illustrate some of the patterns, a running example is provided in Figure 2.

To begin with, every OntoUML 2.0 class defines a group of ontological natures for its instances, a notion borrowed from UFO. They are grouped into substantial and moment natures. The substantial ontological natures are: functional complex, collective and quantity. The moment ontological natures are: intrinsic mode, extrinsic mode, quality and relator. The ontological natures of a class are determined via the meta-attribute *restrictedTo*, composing a set of possible natures. Note that for some non-sortals, for example, this meta-attribute may contain both substantial and moment natures. This is the case of the category Social Entity of Figure 2, whose *restrictedTo* meta-attribute contains the functional complex and relator ontological natures.

A set of ultimate sortals was defined with ontological natures in mind (GUIZZARDI et al., 2021), with the rule that every endurant must instantiate a unique ultimate sortal. These ultimate sortals represent the top of the hierarchy and each provide a stereotype. Substantial

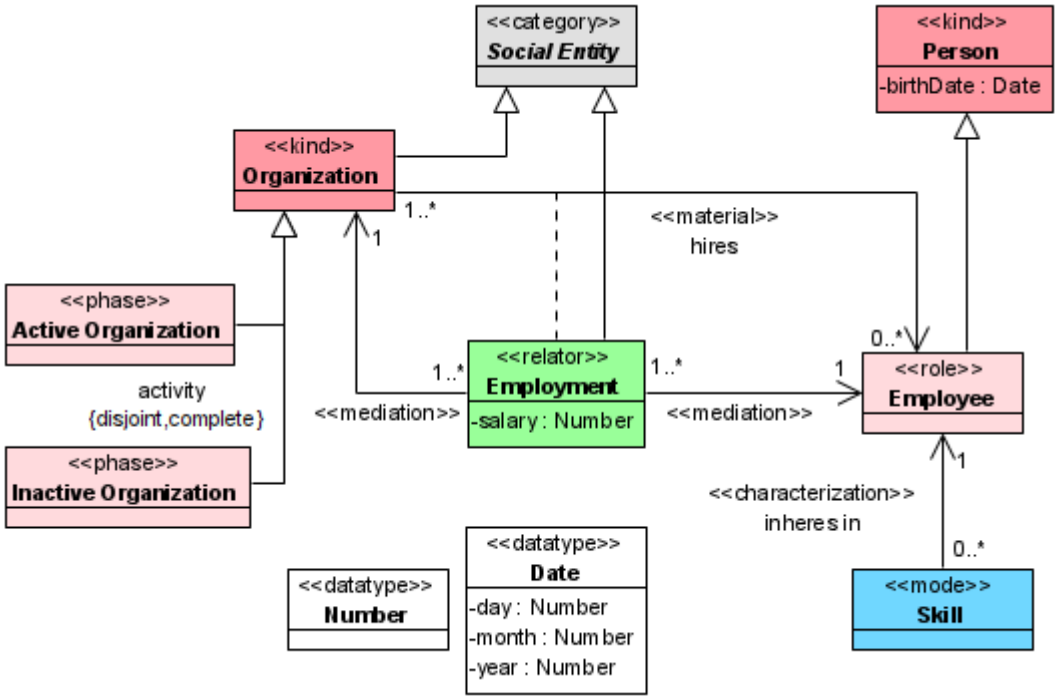

Figure 2 – Running example of OntoUML 2.0 elements.

ultimate sortals are kinds, collectives and quantities. Moment ultimate sortals are relators, qualities and modes. They are detailed below:

- The «kind» stereotype decorates classes that represent substantial ultimate sortals whose instances are regular objects, such as Person and Organization.
- The «collective» stereotype decorates classes that represent substantial ultimate sortals whose instances are groups of objects, such as Soccer Team.
- The «quantity» stereotype decorates classes that represent substantial ultimate sortals whose instances are quantities, typically referred by general mass terms, such as Water and Gold.
- The «relator» stereotype decorates classes that represent moment ultimate sortals whose instances are relators, truthmakers of a material relation, such as Marriage and Employment.
- The «quality» stereotype decorates classes that represent moment ultimate sortals whose instances are qualities, such as Weight, Height and Geometry.
- The «mode» stereotype decorates classes that represent moment ultimate sortals whose instances are modes, intrinsic properties of an individual, such as Goal, Symptom and Skill.

The following stereotypes are used to represent sortals that specialize one of the ultimate sortals and, therefore, cannot exist on their own:

- The «subkind» stereotype decorates classes that represent rigid sortals, such as Non-Profit Organization and Commercial Organization.
- The «phase» stereotype decorates classes that represent externally independent anti-rigid sortals, such as Child, Adult, Active Organization and Inactive Organization.
- The «role» stereotype decorates classes that represent externally dependent anti-rigid sortals, such as Spouse, Employee and Student.

Lastly, the sterotypes for non-sortals are:

- The «category» stereotype decorates classes that represent rigid non-sortals, such as Physical Object (as a superclass of kinds Person and Car) and Social Entity.
- The «phaseMixin» stereotype decorates classes that represent externally independent anti-rigid non-sortals, such as Living Animal and Dead Animal.
- The «roleMixin» stereotype decorates classes that represent externally dependent anti-rigid non-sortals, such as Provider and Customer.
- The «mixin» stereotype decorates classes that represent semi-rigid non-sortals, such as Legally Recognized Civil Partnership.

Furthermore, elements from an OntoUML model can be related with generalizations and stereotyped associations (FONSECA et al., 2019). The «material» and «mediation» stereotypes are often used in the Relator Pattern, in which a material relation is derived from two or more mediations, with a relator as the truthmaker. In Figure 2, this pattern can be found between Organization, Employment and Employee. Other stereotypes, such as «characterization» and «externalDependence» are often used with the Mode Pattern, not included in the running example. Part-whole relations, not represented in the running example, also come with a set of possible stereotypes. They are: «componentOf», «memberOf», «subCollectionOf» and «subQuantityOf», whose semantics trace back to Guizzardi's Ph.D. thesis (GUIZZARDI, 2005).

Additionally, the UML notion of generalization set is employed to represent a group of generalizations of a common superclass, capturing a common criterion of specialization used for its subclasses. The UML meta-attributes *isComplete* and *isDisjoint* are used to capture some extra behavior. A complete generalization set requires that every instance of the common superclass must be an instance of at least one of subclasses from the set. A disjoint generalization set signals that there is no instance of the common superclass that is an instance of multiple

subclasses in the set. In other words, there is no overlapping between the subclasses. A complete disjoint generalization set is a special kind of generalization set in which both meta-attributes are True. To exemplify, an Organization represented in Figure 2 is either Active or Inactive, and never both at the same time.

In summary, in the example of Figure 2, a Person can assume the role of an Employee, if they are hired by an Organization. Employment is a relator that acts as the truthmaker of the material relation *hires*, mediating both the Organization and the Employee. Social Entity is an abstract category whose instances can be substantials (like Organizations) or moments (like Employments). An Organization has two phases, which describe if it is an Active or an Inactive Organization. Finally, the Skill mode characterizes an intrinsic property that inheres in an employee.

2.3 Alloy

Alloy is a logic-based language used to simulate instances of a defined model (JACKSON, 2012). It was chosen as the target of the transformation, given its ease of use and the very powerful *Alloy Analyzer* tool.

Alloy's logic is based on atoms and relations. A relation consists in a set of tuples, each tuple being a sequence of atoms. The arity of the relation is determined by the number of atoms inside a tuple. For instance, a relation that maps names to addresses can be represented as a binary relation, such as `{(N1,A1),(N2,A2),(N3,A3),...}`. Additionally, a function is a binary relation that maps each atom to at most one other atom.

Atoms can be grouped in sets. In Alloy, these sets are called *signatures*. In short, sets are to atoms what classes are to instances in an object-oriented programming language. It is possible to create subsets and simulate generalization sets, just like inheritance in a language like Java. Signatures can also have field declarations, that end up representing a relation between the signature they are in and other atoms (such as others signatures).

Moreover, Alloy models can be constrained by establishing *facts*. These are logical statements that must be true for all instances of the model. The language presents a number of mathematical operators, set operators, relational operators and logical operators to be used in its expressions. Their use allows a number of operations, as simple as additions and unions, but also as complex as transitive closures.

In addition, Alloy allows model verification with the inclusion of *assertions* to verify if a certain expression holds true for every instance. It also provides model validation through the execution of predicates with the *Alloy Analyzer* to generate possible instances of the model. For this, it is necessary to use `run` statements. These statements are executed for a defined scope that constrains the number of atoms used in the enumeration process of Alloy's simulation.

Also, due to the enumeration phase of the process, Alloy is very limited to scope sizes. Because of this, even if no undesired instances are identified for a small scope, it is not possible to affirm that the designed model is free from imperfections. However, in his book (JACKSON, 2012), Jackson discusses the *small scope hypothesis*, a premise he adopts due to the fact that "most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or small instance makes no difference". He completes the hypothesis by stating that "if the analysis considers all small instances, most flaws will be revealed". With this in mind, it is recommended to run models for various instances and scopes (respecting Alloy's limitations), to achieve a better understanding of its structure and to make conclusions about model validation based on sound evidence. Thereby, even if larger scopes are not considered, the final analysis will be supported by the *small scope hypothesis*.

When running predicates, the Alloy Analyzer generates an instance for the model that follows all defined constraints. A pop-up window is opened to show a visualization of the instance, with the options of displaying it as a diagram, as a tree, as a table or as text. Since OntoUML is a diagrammatic language, the first option will be the most useful for the context of this project. With the simulation visualization as a diagram, atoms of Alloy's signatures are displayed on the screen as nodes, with arcs connecting them. These arcs represent the relations defined in the model specification. By having access to instances of the models like this, the modeler can intuitively analyze if their models are underconstrained, overconstrained, incomplete or otherwise poorly designed.

The next chapter will present how OntoUML 2.0 elements can be transformed to an Alloy specification, to take advantage of its powerful model validation and analysis tool. Later, Chapter 4 will show simulation scenarios for the example of Figure 2 leveraging the Alloy Analyzer.

3 The OntoUML 2.0 to Alloy Transformation

This chapter presents the transformation rules from source models in OntoUML 2.0 to target specifications in Alloy. For every model transformation, three modules are generated: the world structure module, which allows adding temporal behavior to the models in Alloy (and is the same for any source model); the ontological properties module, which contains useful predicates that represent UFO constraints (and again is the same for any source model); and the main module, which contains the signatures and all generated fields, facts and functions according to the OntoUML 2.0 elements being transformed. This structure and most transformation rules presented in this chapter were reimplemented and extended from the OntoUML to Alloy transformation proposed in (SALES, 2014).

3.1 World Structure Module

This module describes the world structure proposed in (BENEVIDES et al., 2011). A `World` captures instances of the model in a given moment, along with their possible relations. It functions as a snapshot of the model in a moment in time. Worlds relate to one another via the `next` field declaration, which allows the representation of a possible sequence of worlds (refer to Listing 3.1). An important constraint to avoid temporal cycles is also added, as in `this not in this.^(@next)` (line 6).

In this specification, the `CurrentWorld` is fixed and must exist. It is used as a reference to the other worlds in the simulation (lines 10–27). A world in the `next` attribute of `CurrentWorld` is considered a `FutureWorld`, while a world from which it is possible to reach `CurrentWorld` by a finite number of steps of `next` is considered a `PastWorld`. Lastly, a special world labeled `CounterfactualWorld` is used to represent alternative futures to a `PastWorld`. This is relevant to simulate and check modal properties of an OntoUML model.

Individuals of the model exist in a specific world (or worlds). To restrict their existence and avoid inconsistencies, some constraints are added. First, the `continuous_existence` predicate (line 30) states that an individual cannot cease to exist and then come back to life in a future world. Additionally, the `elements_existence` predicate (line 35) ensures that all individuals must exist in at least one world.

Finally, additional predicates are included to be used as simulation parameters (lines 40–52). The `singleWorld` predicate is used for simulating a single world (Current). Next, the `linearWorlds` predicate is used for simulating a linear world structure, with Past, Current and Future. At last, the `multipleWorlds` predicate includes a Counterfactual to the simulation.

Listing 3.1 – World structure module in Alloy.

```
module world_structure[World]

some abstract sig TemporalWorld extends World{
  next: set TemporalWorld -- Immediate next moments.
} {
  this not in this.^(@next) -- There are no temporal cycles.
  lone ((@next).this) -- A world can be the immediate next momment of at most one
      world.
}

one sig CurrentWorld extends TemporalWorld {} {
  next in FutureWorld
}

sig PastWorld extends TemporalWorld {} {
  next in (PastWorld + CounterfactualWorld + CurrentWorld)
  CurrentWorld in this.^@next -- All past worlds can reach the current moment.
}

sig FutureWorld extends TemporalWorld {} {
  next in FutureWorld
  this in CurrentWorld.^@next -- All future worlds can be reached by the current
      moment.
}

sig CounterfactualWorld extends TemporalWorld {} {
  next in CounterfactualWorld
  this in PastWorld.^@next -- All past worlds can reach the counterfactual moment.
}

-- Elements cannot die and come to life later
pred continuous_existence [exists: World->univ] {
  all w : World, x: (@next.w).exists | (x not in w.exists) => (x not in (( w. ^next)
      .exists))
}

-- All elements must exists in at least one world
pred elements_existence [elements: univ, exists: World->univ] {
  all x: elements | some w: World | x in w.exists
}

-- Run predicate for a single World
pred singleWorld {
        #World=1
}

-- Run predicate for linear Worlds (Past, Current, Future)
pred linearWorlds {
        #World=3 and #PastWorld=1 and #FutureWorld=1
}

-- Run predicate for multiple Worlds (Past, Counterfactual, Current, Future)
pred multipleWorlds {
        #World=4 and #PastWorld=1 and #CounterfactualWorld=1 and #FutureWorld=1
}
```

3.2 Ontological Properties Module

This module defines some ontological properties of UFO to be used during the transformation, such as rigidity and immutability (refer to Listing 3.2).

The `rigidity` and `antirigidity` predicates (lines 4 and 10) are used to constrain classes according to their rigidity ontological property, taking into consideration their ontological nature. In addition, the `immutable_source` and `immutable_target` predicates (lines 16 and 22) are used for relation ends with the *isReadOnly* meta-attribute equal to True.

Listing 3.2 – Ontological properties module in Alloy.

```
1  module ontological_properties[World]
2
3  -- This predicate states that a class is rigid
4  pred rigidity [Class: univ->univ, Nature: univ, exists: univ->univ] {
5    all w1: World, p: univ | p in w1.exists and p in w1.Class implies
6      all w2: World | w1!=w2 and p in w2.exists implies p in w2.Class
7  }
8
9  -- This predicate states that a class is anti-rigid
10 pred antirigidity [Class: set univ->univ, Nature: univ, exists: univ->univ] {
11   all x: Nature | #World>=2 implies (some disj w1,w2: World |
12     x in w1.exists and x in w1.Class and x in w2.exists and x not in w2.Class)
13 }
14
15 -- This predicate makes the source relation end immutable
16 pred immutable_source [Target: World->univ, rel: univ->univ->univ] {
17   all w1: World, x: univ | x in w1.Target implies
18     all w2: World | x in w2.Target implies (w1.rel).x=(w2.rel).x
19 }
20
21 -- This predicate makes the target relation end immutable
22 pred immutable_target [Source: World->univ, rel: univ->univ->univ] {
23   all w1: World, x: univ | x in w1.Source implies
24     all w2: World | x in w2.Source implies x.(w1.rel)=x.(w2.rel)
25 }
```

3.3 Main Module

This module is the main module of the transformation and is where all signatures, fields, facts and functions will be added. It consists of a skeleton specification that is present in every model transformation (refer to Listing 3.3).

First, all required modules are imported, including the world structure module and the ontological properties module discussed previously (lines 3–7). Some native Alloy utility models are also used, such as `util/relation` and `util/ternary` for relation mapping and `util/sequniv` for ordering.

Three main signatures are defined to group all the different classes in the model. Those

are: `Endurant`, `Object` and `Aspect` (lines 9, 10 and 11). With this definition, `Endurant` is an abstract signature, with every `Endurant` being either an `Object` or an `Aspect`. These signatures will be discussed more in depth in Section 3.3.1. Additionally, the `Datatype` signature (line 13) will be used in datatype mapping, discussed in Section 3.3.5.

The core signature of this module and the home of most of the OntoUML 2.0 element definitions is the abstract signature `World` (lines 15–17). Classes and relations will be mapped as fields of this signature, to capture the behavior that individuals and their relations exist in a particular `World`. It is also worth to notice that this signature is used as a parameter to the world structure and ontological properties modules.

Moreover, a fact block is included to invoke the `continuous_existence` predicate and the `elements_existence` predicate (lines 19–22), discussed in the previous section. This guides the simulation to show every element of the model in at least one `World` and avoid the reappearance of individuals throughout the timeline.

In addition, a function named *visible* (line 24) is used to improve the model visualization when running simulations. This function has no effect in the transformation, acting solely as a workaround kept from (BRAGA et al., 2010) and (SALES, 2014). The `exists` field is added to its body in every model transformation, so that all things that exist in the worlds appear as nodes in the simulation. Additional terms are appended with a + if they involve datatypes, such as class attributes. This is necessary due to the atemporality property of datatypes and will be further discussed in Chapter 4.

Finally, six run statements are proposed to guide the modeler in different simulation scenarios (lines 29–34). All involve the `singleWorld`, `linearWorlds` and `multipleWorlds` predicates described in Section 3.1, with the first three being used for a small scaled simulation and the last three being using for a larger scaled simulation. Note that these are optional and the modeler can come up with their own statements if they wish.

Listing 3.3 – Skeleton specification of the main module in Alloy.

```
1  module main
2
3  open world_structure[World]
4  open ontological_properties[World]
5  open util/relation
6  open util/sequniv
7  open util/ternary
8
9  abstract sig Endurant {}
10
11 sig Object extends Endurant {}
12
13 sig Aspect extends Endurant {}
14
15 sig Datatype {}
16
```

```
17 abstract sig World {
18     exists: some Endurant,
19 }
20
21 fact additionalFacts {
22     continuous_existence[exists]
23     elements_existence[Endurant, exists]
24 }
25
26 fun visible : World->univ {
27     exists
28 }
29
30 -- Suggested run predicates
31 run singleWorld for 10 but 1 World, 7 Int
32 run linearWorlds for 10 but 3 World, 7 Int
33 run multipleWorlds for 10 but 4 World, 7 Int
34 run singleWorld for 20 but 1 World, 7 Int
35 run linearWorlds for 20 but 3 World, 7 Int
36 run multipleWorlds for 20 but 4 World, 7 Int
```

3.3.1 Mapping of Classes

All classes, with the exception of those stereotyped as «dataype» and «enumeration», are mapped into a field declaration in the `World` signature, taking into consideration the ontological natures within their *restrictedTo* meta-attribute (refer to Table 1). Thereby, class fields represent binary relations that tie every `World` to a subset of individuals that exist in that `World`. The expression `World.Class` refers to every individual that instantiates the class in any `World`, whilst the expression `w.Class` (`w` being a particular `World`) refers to the individuals that instantiate `Class` in `w`. Keep in mind that any red-colored keywords of this chapter refer to variables used in the transformation rules. For instance, `Class` refers to the name of the class transformed to Alloy.

If the transformed class is restricted only to substantial ontological natures (functional complex, collective or quantity), the field's type is the projection of the `Object` signature, as in `ObjectClass: set exists:>Object`. For clarification, a projection of a relation `r` over a set `s`, represented by `r:>s`, contains all the tuples of `r` that end with an element in `s`. Therefore, ObjectClass can only contain atoms that exist in a certain world as Objects. Analogously, if the transformed class is restricted only to moment ontological natures (intrinsic mode, extrinsic mode, quality or relator), the field's type is the projection of the `Aspect` signature, as in `AspectClass: set exists:>Aspect`. If, however, the *restrictedTo* meta-attribute contains both substantial and moment ontological natures, the field's type is the projection of the `Endurant` signature, as in `EndurantClass: set exists:>Endurant`. The keyword `set` is used to make it optional for a world to contain an instance of a class.

In addition, for top-level classes in the model hierarchy, the rigidity ontological property of the class is mapped into a fact block. This approach seems to rule out cases where anti-rigid

Table 1 – Class mapping to Alloy.

OntoUML 2.0	Alloy

ObjectClass, AspectClass, EndurantClass	`abstract sig World {`
	`  ObjectClass: set exists:>Object,`
	`  AspectClass: set exists:>Aspect,`
	`  EndurantClass: set exists:>Endurant`
	`}`

RigidClass	`fact {`
	`  rigidity[RigidClass,Nature,exists]`
	`}`

AntirigidClass	`fact {`
	`  antirigidity[AntirigidClass,Nature,exists]`
	`}`

isAbstract	`fact abstractClass {`
	`  all w: World | w.AbstractClass =`
	`  w.Subclass_1+...+w.Subclass_N`
	`}`

classes specialize rigid classes. However, by default, anti-rigid behavior can be verified but is not required to show up in the model instances. That said, for rigid classes, i.e. stereotyped as «kind», «quantity», «collective», «mode», «quality», «relator», «subkind» or «category», the rigidity predicate is called within the fact block, as in `rigidity[RigidClass,Nature,exists]`. For anti-rigid classes, stereotyped as «role», «roleMixin», «historicalRole», «historicalRoleMixin», «phase» or «phaseMixin», the antirigidity predicate is called within the fact block, as in `antirigidity[AntirigidClass,Nature,exists]`. It is worth to notice that, in this context, `Nature` refers to `Object`, `Aspect` or `Endurant`. Finally, semi-rigid classes do not require any further constraining.

Furthermore, if the class's *isAbstract* meta-attribute is True, an additional fact is provided to constraint the class's extension to the union of all of its subclasses, as in `AbstractClass = Subclass_1+...+Subclass_N`.

The mapping of datatypes and enumerations will be discussed in Section 3.3.5.

To illustrate the mapping described in this section, consider the classes of Figure 3, extracted from the running example of the previous section. The transformation of these classes to Alloy is specified in Listing 3.4. Social Entity is a category whose *restrictedTo* meta-attribute contains the functional complex and relator ontological natures. Therefore, it is mapped as an Endurant (line 3). It is also a top-level rigid class (lines 8–10) and abstract class (lines 12–14). Organization and Employment are not top-level classes. Thus, they are only mapped to field declarations of the `World` signature according to their Object and Aspect natures, respectively (lines 4 and 5).

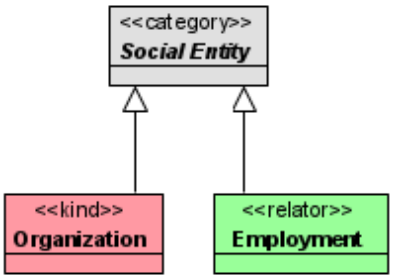

Figure 3 – Fragment of the running example showcasing classes.

Listing 3.4 – Fragment of the Alloy specification of the running example for classes.

```
abstract sig World {
    exists: some Endurant,
    SocialEntity: set exists:>Endurant,
    Organization: set exists:>Object,
    Employment: set exists:>Aspect
} {}

fact rigid {
    rigidity[SocialEntity,Endurant,exists]
}

fact abstractClass {
    all w: World | w.SocialEntity = w.Organization+w.Employment
}
```

3.3.2 Mapping of Relations

Like classes, relations are mapped into a field declaration in the `World` signature. The only exceptions are derivations, which are mapped into a fact (refer to Table 2). Relation fields represent ternary (or 4-ary, in the case of ordered associations) relations that tie every `World` to a subset of related individuals that exist in that `World`. The ternary relation is justified by the fact that an association between `Class1` and `Class2` happens in the extension of a `World`, the world being the third dimension of the relation. In this work, only binary associations between classes will be considered.

As a general rule, relations are mapped as a ternary field that relates their `Source` and `Target` classes, as in `GeneralRelation: set Source -> Target`. If any of the relation ends is tagged with the *isOrdered* meta-attribute, the mapping changes to a 4-ary relation, that uses the built-in Int signature as a fourth dimension to create the behavior of ordering, as in `OrderedRelation: set Source -> Int -> Target`. However, the inclusion of Int in the relation only provides the "position" of the relation, deeming necessary the inclusion of an additional ordering fact block to ensure the proper ordering behavior.

Relations that take part in the Relator Pattern have special mapping rules. The material

Table 2 – Relation mapping to Alloy.

OntoUML 2.0	Alloy

GeneralRelation	`abstract sig World {`
	`  GeneralRelation: set Source -> Target`
	`}`

MaterialRelation (connected to a DerivationRelation)	`abstract sig World {`
	`  MaterialRelation: set Source -> Relator -> Target`
	`}`

isOrdered	`abstract sig World {`
	`  OrderedRelation: set Source -> Int -> Target`
	`}`
	
	`fact ordering {`
	`  all w: World, x: w.SourceClass |`
	`  isSeq[x.(w.OrderedRelation)]`
	`  all w: World, x: w.SourceClass, y: w.OrderedClass |`
	`  lone x.((w.OrderedRelation).y)`
	`}`

DerivationRelation	`fact derivation {`
	`  all w: World, x: w.Source, y: w.Target,`
	`  r: w.Relator |`
	`  x -> r -> y in w.MaterialRelation iff`
	`  x in r.(w.MediationRelation_1) and`
	`  y in r.(w.MediationRelation_2)`
	`}`

isComposite (PartWholeRelation)	`fact composition {`
	`  all w: World, x : w.PartClass |`
	`  lone WholeRelationEnd[x,w]`
	`  all w: World, x : w.PartClass |`
	`  some WholeRelationEnd[x,w] implies no`
	`  (OtherWholeRelationEnd_1[x,w]+...+`
	`  OtherWholeRelationEnd_N[x,w])`
	`}`

MediationRelation, PartWholeRelation	`fact acyclic {`
	`  all w: World |`
	`  acyclic[w.MediationRelation,w.Source]`
	`  all w: World |`
	`  acyclic[w.PartWholeRelation,w.Source]`
	`}`

relation, whose truthmaker is a relator, is mapped into a 4-ary relation, as in `MaterialRelation : set Source -> Relator -> Target`. The derivation relation that ties the relator to the material relation is represented in a fact block, constraining the derived material relation to the mediation relations present in the pattern. Also, it is important to notice that relations with the «material» stereotype which are not part of a Relator Pattern in the model are treated as a general relation.

Furthermore, part-whole relations and relations stereotyped as «componentOf», «memberOf», «subCollectionOf» or «subQuantityOf» are mapped as general relations, with the addition of a fact block to capture the semantics of a composite aggregation, if that is the case. Two constraints are taken into consideration: the first states that a part can compose at most one whole; the second forbids the composition of wholes of any other type. In Section 3.3.3, constraints on association ends with the *isReadOnly* meta-attribute help with the immutability

semantics of these relations as well. In addition, Section 3.3.6 will take into consideration the weak supplementation axiom, an important constraint for part-whole relations. It is also worth to notice that the main focus of this project was put into classes and regular associations, so not everything from part-whole relations is covered.

Lastly, part-whole relations and mediations must not compose a cycle in the model. They must also be anti-reflexive and anti-symmetric. To prevent this behavior, an acyclic constraint is introduced for every relation of this kind. Since acyclic also implies anti-reflexivity and anti-symmetry, no further constraining is required.

The mapping of relations between datatypes will be discussed in Section 3.3.5.

As an example, consider the fragment of Figure 4, extracted from the running example. The transformation of the relations of this fragment to Alloy is specified in Listing 3.5. This example showcases the Relator Pattern, where the material relation `hires` is derived from the relator Employement, which mediates classes Organization and Employee. The material relation is mapped to a 4-ary relation (line 2), with the relator, acting as the truthmaker of the relation, being represented as the fourth dimension. The unnamed mediations are mapped as general relations (lines 3 and 4). Moreover, acyclic constraints are created for each one of them (lines 7–9 and 11–13). Finally, the derivation is not mapped into a field declaration in the `World` signature, but as a fact instead (lines 15–18).

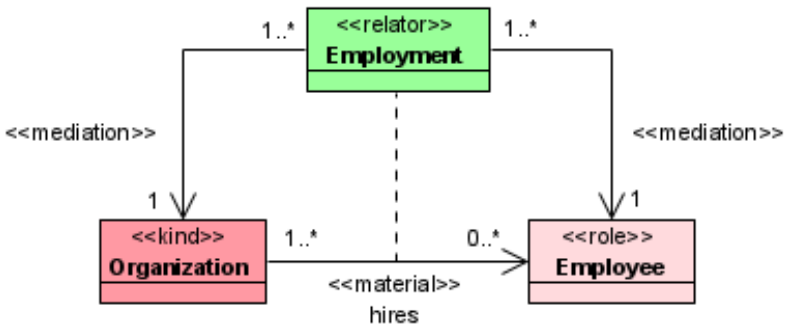

Figure 4 – Fragment of the running example showcasing relations.

Listing 3.5 – Fragment of the Alloy specification of the running example for relations.

```
1  abstract sig World {
2      hires: set Organization -> Employment -> Employee,
3      relation1: set Employment -> one Organization,
4      relation2: set Employment -> one Employee
5  } {}
6
7  fact acyclic {
8      all w: World | acyclic[w.relation1,w.Employment]
9  }
10
11 fact acyclic {
12     all w: World | acyclic[w.relation2,w.Employment]
```

```
13 }
14
15 fact derivation {
16     all w: World, x: w.Organization, y: w.Employee, r: w.Employment |
17             x -> r -> y in w.hires iff x in r.(w.relation1) and y in r.(w.relation2)
18 }
```

3.3.3 Mapping of Properties: Attributes and Relation Ends

Attributes and relation ends are grouped as properties in OntoUML 2.0. Since they contain many similarities, this section discusses the mapping of both. Attributes essentially represent relations between a class and a datatype and are mapped in a similar way as relations were mapped in the previous section: into a field declaration of the `World` signature. Both attribute ends and relation ends are mapped to functions in Alloy, which are later used in a number of facts as "aliases" (refer to Table 3).

Source ends are mapped to `(w.Relation).x` or `(select13[w.Relation].x`, depending on the arity of their relations. Target ends are mapped to `x.(w.Relation)` or `x.(select13[w.Relation])`. Note that the `select13[]` predicate is borrowed from the `ternary` utility module and returns a binary relation of the first and last atoms of the ternary relation. In other words, if `a->b->c` is a ternary relation, `select13[a->b->c]` will return the binary relation `a->c`. This is used to ignore the relator class present in material relations. Lastly, attribute ends are like target ends in their class to datatype relation. Thus, they share the same mapping as target ends.

In addition, the multiplicity of attributes and relation ends is usually mapped into the field declaration of their respective relations. That is the case for default multiplicities, such as 0..1 (treated with Alloy's `lone` keyword), 1..1 (`one`), 1..* (`some`) and 0..* (`set`). For custom multiplicities, however, an additional fact is used to constraint the relation end to the lower bound and the upper bound values. For association ends whose container is a material relation (connected to a derivation), the multiplicity is also mapped to facts, regardless of it being custom or not.

Finally, for immutable attributes and relation ends with the *isReadOnly* meta-property, an additional fact block is added, including the respective `immutable_target` predicate or `immutable_source` predicate from the ontological properties module.

The mapping of properties of datatypes will be discussed in Section 3.3.5.

For instance, consider the fragment of Figure 5, extracted from the running example. The transformation of the attribute and relation ends of this fragment to Alloy is specified in Listing 3.6. The attribute `birthDate` is mapped to a field declaration of the `World` signature (line 2), much like a relation (as seen in the previous section). The two unnamed association ends of the `inheresin` characterization relation are mapped to functions (lines 9–11 and 13–15).

Table 3 – Property mapping to Alloy.

OntoUML 2.0	Alloy

Attribute	`abstract sig World {`
	`  Attribute: set OwnerClass set -> AttributeType,`
	`  OrderedAttribute: set OwnerClass set -> set Int set -> set AttributeType`
	`}`
	
	`fun AttributeEnd [x: World.OwnerClass,w: World] : set AttributeType {`
	`  x.(w.Attribute)`
	`}`

SourceEnd,
TargetEnd
	`fun SourceEnd [x: World.Target,w: World] : set World.Source {`
	`  (w.Relation).x`
	`}`
	
	`fun TargetEnd [x: World.Source,w: World] : set World.Target {`
	`  x.(w.Relation)`
	`}`
	
	`-- Material (connected to derivation) or Ordered`
	`fun SourceEnd [x: World.Target,w: World] : set World.Source {`
	`  (select13[w.Relation]).x`
	`}`
	
	`-- Material (connected to derivation) or Ordered`
	`fun TargetEnd [x: World.Source,w: World] : set World.Target {`
	`  x.(select13[w.Relation])`
	`}`

Default Multiplicity	`-- 0..1 (lone); 1..1 (one); 1..* (some); 0..* (set)`
	`abstract sig World {`
	`  Attribute: set OwnerClass set -> some AttributeType`
	`  Relation: set Source lone -> one Target`
	`}`

Custom Multiplicity	`-- Attribute`
	`fact multiplicity {`
	`  all w: World, x: w.OwnerClass |`
	`  #AttributeEnd[x,w]>=LowerBound and`
	`  #AttributeEnd[x,w]<=UpperBound`
	`}`
	
	`-- Source End`
	`fact multiplicity {`
	`  all w: World, x: w.Target |`
	`  #SourceEnd[x,w]>=LowerBound and`
	`  #SourceEnd[x,w]<=UpperBound`
	`}`
	
	`-- Target End`
	`fact multiplicity {`
	`  all w: World, x: w.Source |`
	`  #TargetEnd[x,w]>=LowerBound and`
	`  #TargetEnd[x,w]<=UpperBound`
	`}`

isReadOnly	`fact associationProperties {`
	`  immutable_target[OwnerClass,Attribute] -- Attribute`
	`  immutable_target[Source,Relation] -- Source End`
	`  immutable_source[Target,Relation] -- Target End`
	`}`

The names Skill1 and Employee3 are aliases used for when the association ends do not have an explicit name. They refer to the class connected to the end and receive the class's name, followed by a number to differentiate other ends connected to the same class. Lastly, since the relation `inheresin` is a characterization, the *isReadOnly* meta-attribute of the target end is true. Thus, the `immutable_target` predicate is added to a fact block (line 6).

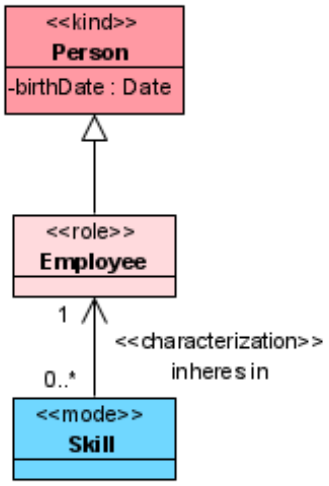

Figure 5 – Fragment of the running example showcasing properties.

Listing 3.6 – Fragment of the Alloy specification of the running example for properties.

```
1  abstract sig World {
2      birthDate: set Person set -> one Date
3  } {}
4
5  fact relationProperties {
6      immutable_target[Skill,inheresin]
7  }
8
9  fun Skill1 [x: World.Employee, w: World] : set World.Skill {
10     (w.inheresin).x
11 }
12
13 fun Employee3 [x: World.Skill, w: World] : set World.Employee {
14     x.(w.inheresin)
15 }
```

3.3.4 Mapping of Generalizations and Generalization Sets

The mapping of generalizations is very straightforward, since Alloy has the `in` construct used to subset signatures. The inheritance between a subclass and a superclass can be seen as subsetting, since the extension of the subclass is said to be included in the extension of the superclass. Therefore, adding the expression `Subclass in Superclass` to a fact block is enough to generate this behavior (refer to Table 4).

Table 4 – Generalization and generalization set mapping to Alloy.

OntoUML 2.0	Alloy
Generalization	`fact generalization { Subclass in Superclass }`
Generalization Set (*isDisjoint*, *isComplete*)	`fact generalizationSet { disjoint[Subclass_1,...,Subclass_N] -- Disjoint Superclass = Subclass_1+...+Subclass_N -- Complete }`

Additionally, the *isDisjoint* and *isComplete* meta-attributes of generalization sets are mapped into a fact block to capture their behavior, with the expressions `disjoint[Subclass_1,...,Subclass_N]` (if disjoint) and `Superclass = Subclass_1+...+Subclass_N` (if complete).

To illustrate, consider the fragment of Figure 6, extracted from the running example. The transformation of the generalizations and generalization set of this fragment to Alloy is specified in Listing 3.7. The Active Organization and Inactive Organization phases are subclasses of the common superclass Organization. The specializations are mapped into simple fact blocks (lines 1–3 and 5–7). The generalization set `activity`, formed by these two phases, is disjoint and complete. Therefore, the mapping into the fact block also specifies the disjointness (line 10) and the completeness (line 11).

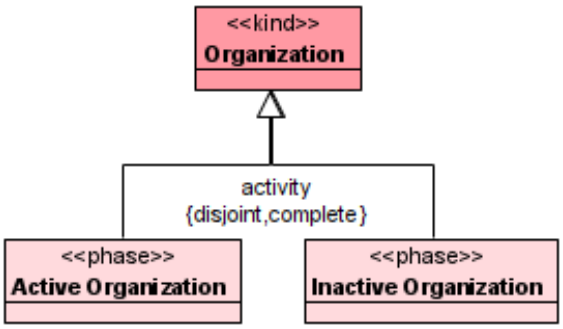

Figure 6 – Fragment of the running example showcasing generalizations.

Listing 3.7 – Fragment of the Alloy specification of the running example for generalizations.

```
1  fact generalization {
2      ActiveOrganization in Organization
3  }
4
5  fact generalization {
6      InactiveOrganization in Organization
7  }
8
9  fact generalizationSet {
10     disjoint[ActiveOrganization,InactiveOrganization]
11     Organization = ActiveOrganization+InactiveOrganization
12 }
```

3.3.5 Mapping of Datatypes and Enumerations

Because datatypes are atemporal, they cannot be mapped into a field declaration of the `World` signature like other classes. Thus, they are mapped into individual signatures that subset the general `Datatype` signature. Enumerations are also atemporal, but are mapped to the Alloy built-in `enum` construct (refer to Table 5).

Table 5 – Datatype and enumeration mapping to Alloy.

OntoUML 2.0	Alloy
DatatypeClass	`sig DatatypeClass in Datatype {}`

Attribute (within a DatatypeClass)	`sig DatatypeClass in Datatype {`
	`  Attribute: AttributeType -- lone, one, some, set`
	`}`
	
	`-- Custom multiplicity`
	`fact multiplicity {`
	`  all x: DatatypeClass | #x.Attribute >=LowerBound`
	`  and #x.Attribute <=UpperBound`
	`}`

Relation (between DatatypeClasses)	`sig SourceDatatypeClass in Datatype {`
	`  Relation: TargetDatatypeClass`
	`}`
	
	`fact multiplicity {`
	`  all x: SourceDatatypeClass |`
	`  #x.Relation >=TargetLowerBound`
	`  and #x.Relation <=TargetUpperBound`
	`  all x: TargetDatatypeClass |`
	`  #Relation.x>=SourceLowerBound`
	`  and #Relation.x<=SourceUpperBound`
	`}`

EnumerationClass	`enum EnumerationClass {`
	`  Literal_1, ..., Literal_N`
	`}`

Attributes and relations between datatypes are mapped into field declarations in the source datatype signature. Any stereotype associated to relations is ignored. The multiplicity of attributes is mapped the same way as to how it is done for regular classes. For relations, however, the multiplicity is always defined in a fact block, taking into consideration the lower bound and the upper bound of both the source and target ends of the relation. If the lower bound or the upper bound are undefined, the part of expression concerning them is ommited.

No additional mapping is required for generalizations and generalization sets involving datatypes. Also, enumerations do not support attributes, specializations or relations between themselves.

For instance, consider the fragment of Figure 7, extracted from the running example. The transformation of the datatypes of this fragment to Alloy is specified in Listing 3.8. The Number and Date datatypes are mapped into individual signatures, independent of `World`, that subset the `Datatype` signature (lines 1 and 3). Additionally, the attributes of Date are mapped into field declarations of its own signature (lines 4–6).

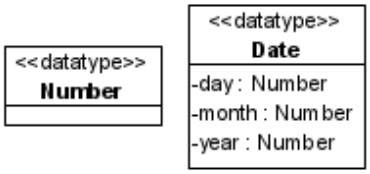

Figure 7 – Fragment of the running example showcasing datatypes.

Listing 3.8 – Fragment of the Alloy specification of the running example for datatypes.

```
1 sig Number in Datatype {}
2
3 sig Date in Datatype {
4     day: one Number,
5     month: one Number,
6     year: one Number
7 }
```

3.3.6 Additional Facts

After transforming the OntoUML 2.0 elements, some additional constraints should be considered to enforce some other ontological properties of UFO and guide the Alloy simulation.

First, to constraint the existence of individuals in a `World` only to those defined in the `World` signature, the projection of the `Object` signature must be in the union of all top-level object classes. Analogously, the projection of the `Aspect` signature must be in the union of all top-level aspect classes, and the projection of the `Endurant` signature must be in the union of all top-level endurant classes (refer to Table 6).

Next, a similar constraint is defined for datatypes. The extension of the Datatype signature must be equal to the union of all its datatype subtypes, as in `Datatype = DatatypeClass_1+...+DatatypeClass_N`. Also, all top-level datatypes must be disjoint. This is enforced by adding the expression `disjoint[DatatypeClass_1+...+DatatypeClass_N]`.

Additionally, the identity principle requires that ultimate sortals must be disjoint. To enforce this rule and also prevent top-level classes with different ontological natures to overlap in the Alloy specification, the expression `disjoint[Class,(DifferentNaturedClass_1+...+DifferentNaturedClass_N)]` is included within a fact field in the `World` signature.

Moreover, classes with the «relator» stereotype must obey a relator rule that requires every instance to mediate at least two disjoint entities. To enforce the desired ontological property, a fact block is specified taking into consideration the relation ends of every mediation relation in which the relator takes part.

Furthermore, classes with an intrinsic moment ontological nature (intrinsic mode or quality) must not indirectly characterize themselves. To prevent this, an acyclic characterization

Table 6 – Additional mapping to Alloy.

OntoUML 2.0	Alloy

Existence Constraint	`abstract sig World {} {`
	`  exists:>Object in ObjectClass_1 +...+ ObjectClass_N`
	`  exists:>Aspect in AspectClass_1 +...+ AspectClass_N`
	`  exists:>Endurant in EndurantClass_1 +...+`
	`  EndurantClass_N`
	`}`

Datatypes Constraint	`fact additionalDatatypeFacts {`
	`  Datatype = DatatypeClass_1 +...+ DatatypeClass_N`
	`  disjoint[DatatypeClass_1 +...+ DatatypeClass_N]`
	`}`

Identity Constraint	`abstract sig World {} {`
	`  disjoint[Class,(DifferentNaturedClass_1`
	`  +...+ DifferentNaturedClass_N)]`
	`}`

Relator Constraint	`fact relatorConstraint {`
	`  all w: World, x: w.RelatorClass |`
	`  #(MediationRelationEnd_1[x,w]`
	`  +...+ MediationRelationEnd_N[x,w])>=2`
	`}`

Acyclic Characterizations Constraint	`fact acyclicCharacterizations {` ` all w: World	acyclic[` ` (w.CharacterizationRelation_1 +...+` ` CharacterizationRelation_N),` ` (w.IntrinsicMoment_1 +...+` ` w.IntrinsicMoment_N)]` `}`

Weak Supplementation Constraint	`fact weakSupplementationConstraint {`
	`  all w: World, x: w.WholeClass |`
	`  #(PartRelationEnd_1[x,w]`
	`  +...+ PartRelationEnd_N[x,w])>=2`
	`}`

constraint is required, considering the union of all characterization relations and the union of all the intrinsic moments.

Finally, the weak supplementation axiom states that the number of parts composing a whole must be equal or greater to two, otherwise the whole and the part would be the same. This rule is enforced by generating a fact block for every `WholeClass`, constraining the minimum of parts in every whole individual.

As an example, consider the entire running example (Figure 2). The mapping of its additional facts is shown in Listing 3.9. The existence and the identity constraints are mapped into fact fields of the `World` signature (lines 3–5 and 6–7). Note how only top-level classes are taken into consideration. In addition, the relator constraint is defined for the Employment relator (lines 10–12). Moreover, top-level datatypes must form a disjoint complete generalization set, mapped to a fact block(lines 14–17). And lastly, an `acyclic` predicate should be considered for the `inheresin` characterization (lines 19–21). Note that Skill is the only intrinsic moment of the model.

Listing 3.9 – Fragment of the Alloy specification of the running example for additional facts.

```
abstract sig World {
} {
    disjoint[SocialEntity,(Number+Date+Skill)]
    disjoint[Person,(Number+Date+SocialEntity+Skill)]
    disjoint[Skill,(Number+Date+SocialEntity+Person)]
    exists:>Object in SocialEntity+Person
    exists:>Aspect in SocialEntity+Skill
}

fact relatorConstraint {
    all w: World, x: w.Employment | #(Organization1[x,w]+Employee1[x,w])>=2
}

fact additionalDatatypeFacts {
    Datatype = Number+Date
    disjoint[Number,Date]
}

fact acyclicCharacterizations {
    all w: World | acyclic[(w.inheresin),(w.Skill)]
}
```

4 Simulation Scenarios

This chapter presents three simulation scenarios created for the running example model transformed to Alloy. They are all executed for a small scope to limit the discussion to a few focused details. We show various instances generated by the Alloy Analyzer, which are visualized as graphs composed of nodes and arcs. To achieve these instances, the models were run for at least a dozen of instances.

For all the studied scenarios, a theme was used to improve model visualization. This theme is the same one used in (SALES, 2014), with some minor adaptations. Atoms of the Object signature are represented as yellow rectangles. Atoms of the Aspect signature are represented as pink ellipses. Datatypes are represented as gray hexagons. All elements are projected over the `World` signature, so that the instances of `World` do not show as nodes. The simulation makes it possible to choose which world to visualize with a dropdown feature. Also, because of this projection, the relations between endurant nodes are shown as arcs, much like in an OntoUML model. The Alloy Analyzer names every atom with the name of their container signature (corresponding to the ontological nature of the instance), followed by a number (starting from 0) to differentiate atoms of the same signature.

Due to the fact that datatypes are atemporal, any associations between datatypes can transcend worlds. In other words, with the proposed idea of projecting the atoms over the `World` signature, the visualization of these associations is a bit trickier. One could switch themes to visualize datatypes in a better manner, if they wish, but this will not be explored in depth in this work.

4.1 Scenario 1: Single World

In this scenario, the `singleWorld` predicate was run for a small scope of at most 10 atoms for each top-level signature. A notable instance is shown in Figure 8. This figure shows the current (and only) world observed for this instance.

Note that Object0 and Object1 are Persons, but Object1 also assumes the role of an Employee. Object1 is hired by the Active Organization Object3. This material relation is mediated by Aspect1, an Employment relator with a `salary` attribute of Datatype0 (a Number). The arcs relation1 and relation2 represent the mediation relations. In addition, Aspect0 is a skill that inheres in Object1 and Object2 is an Active Organization with no employees.

For this scenario, the OntoUML model of Figure 2 has generated a predictable instance. There is no undesired behavior here and the modeler might conclude that their model is well defined. However, with only a single example, this conclusion may seem a bit hasty.

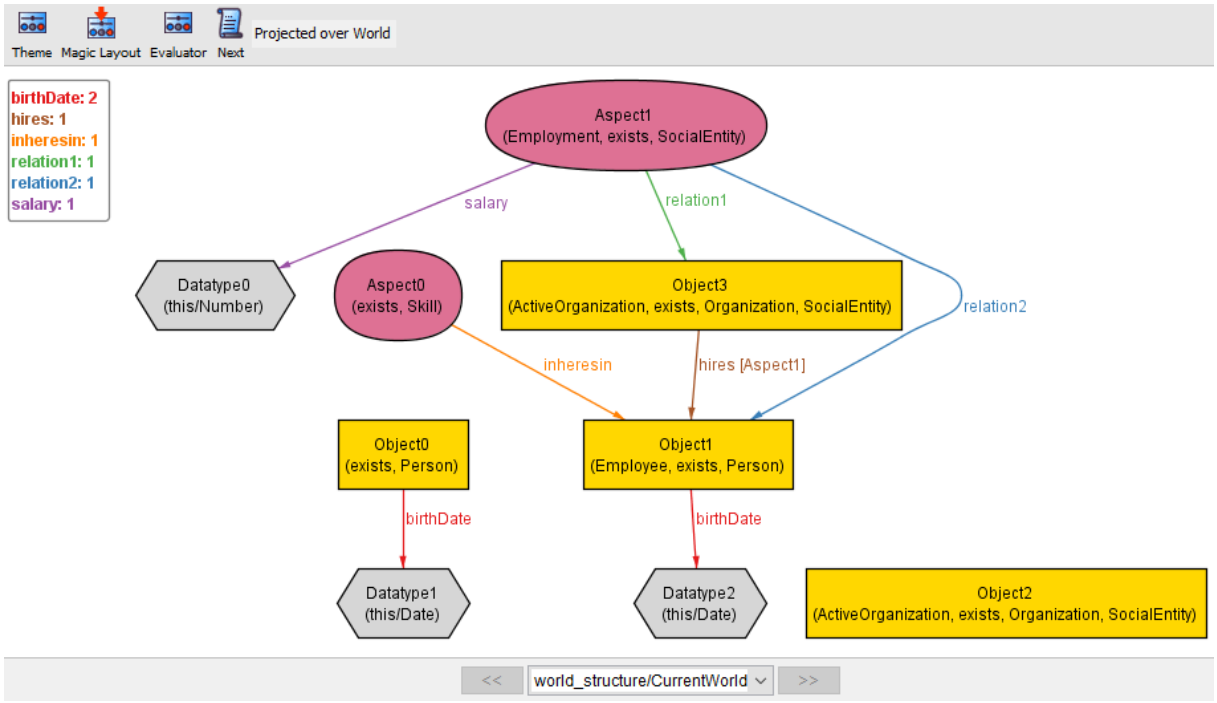

Figure 8 – Current (and only) world of the simulation scenario 1.

4.2 Scenario 2: Linear Worlds

In this scenario, the `linearWorlds` predicate was run for a small scope of at most 10 atoms for each top-level signature. Some interesting instances were generated, one of which is shown in Figures 9, 10 and 11. These figures show the past, current and future worlds of this instance, respectively.

In Figure 9 (past world), Object2 is an Employee that is hired by the Inactive Organization Object1. Aspect3 is the Employment relator that mediates the Employee and the Organization in the `hires` material relation. Also notice that Employment has a `salary` attribute of the Number datatype (Datatype4) and the Employee has a `birthDate` attribute of the Date datatype (Datatype7). Lastly, Aspect2 is a skill that inheres in the Employee.

In Figure 10 (current world), Object2 no longer exists. Another Person, Object3, shows up in the timeline and is not employed. Also, Object1 remains an Inactive Organization, but with no employees.

In Figure 11 (future world), Object3 is now employed, but in two Organizations, Object0 and Object1. Notice how Object1 became an Active Organization. Two Employment relators mediate the `hires` material relations, with the same `salary` attributes of Datatype4.

An interesting thing that can be analyzed about this scenario is the fact that an Inactive Organization has hired employees. One could say that this is unwanted. By simulating their models in Alloy, a modeler can then observe details such as this, where additional constraints are desired to improve their models. Then, they can redesign their models to prevent this

behavior, or even add invariants such as OCL constraints (WARMER; KLEPPE, 2003). Particular observations like these are the core purpose of using Alloy to support modelers creating models with higher quality.

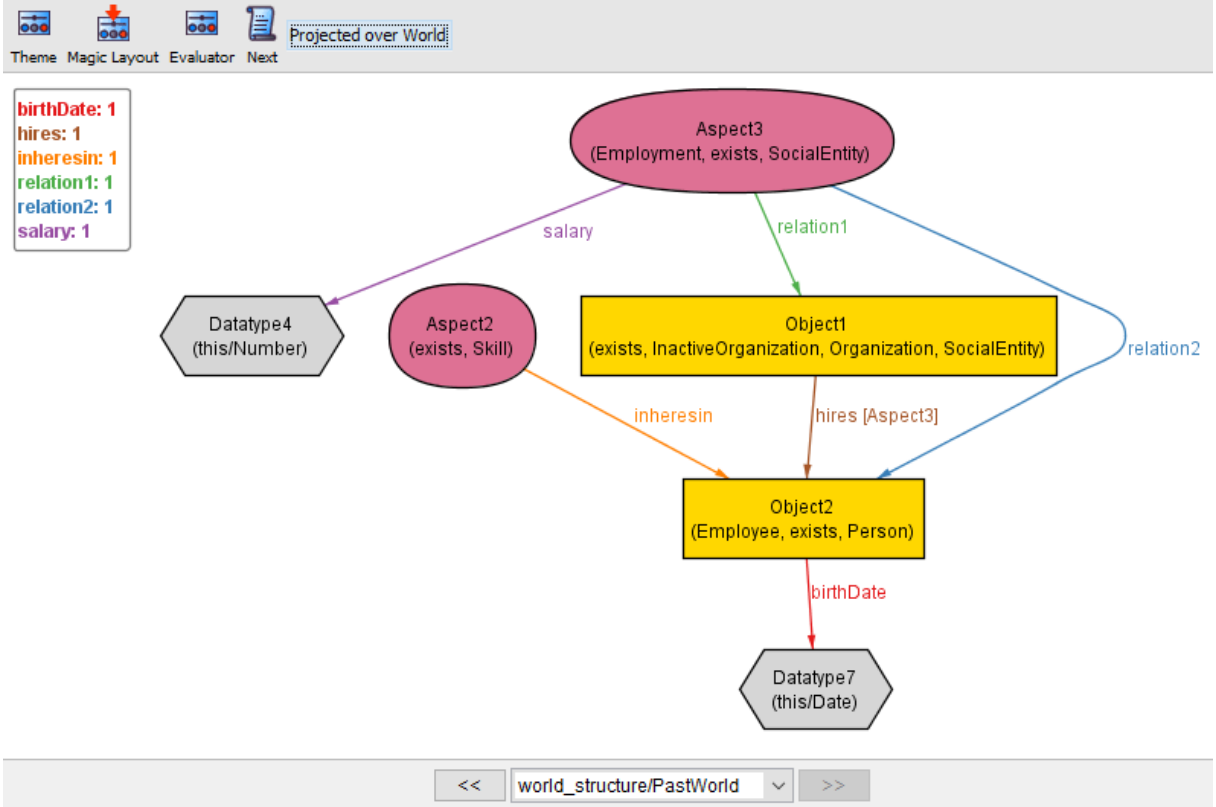

Figure 9 – Past world of the simulation scenario 2.

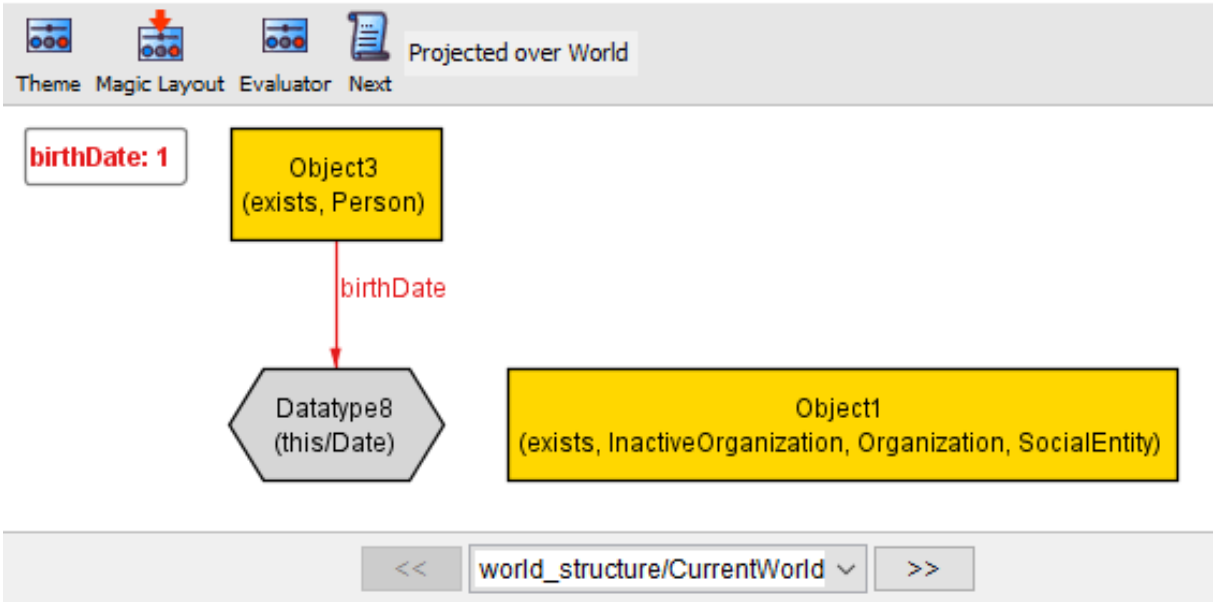

Figure 10 – Current world of the simulation scenario 2.

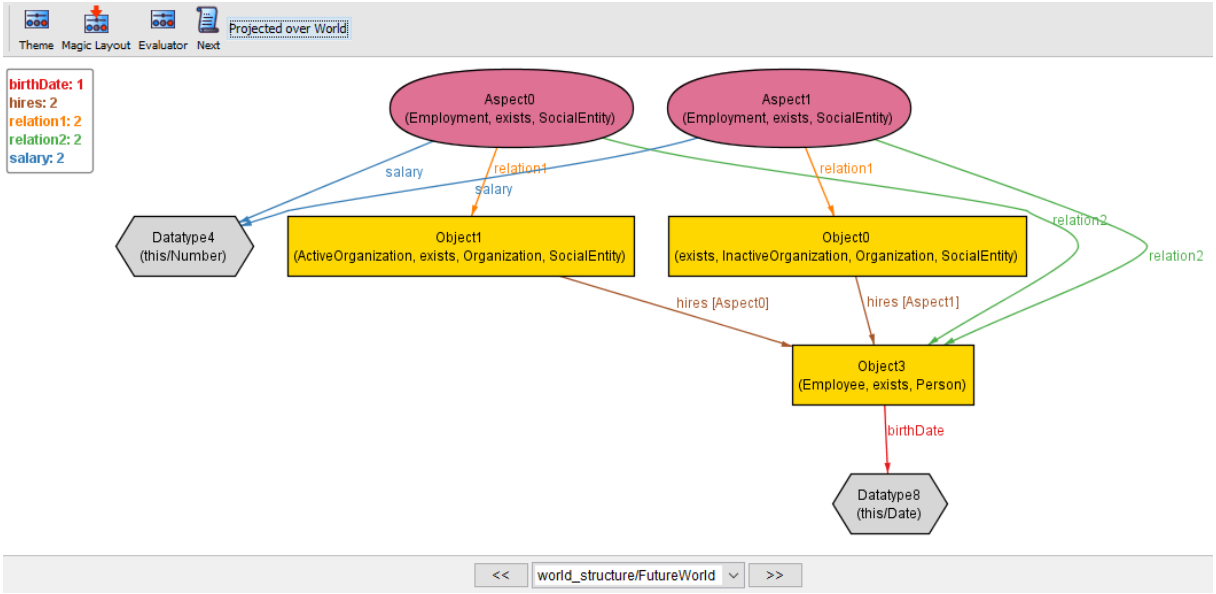

Figure 11 – Future world of the simulation scenario 2.

4.3 Scenario 3: Multiple Worlds

In this final scenario, the `multipleWorlds` predicate was run for a small scope of at most 10 atoms for each top-level signature. Figures 12, 13, 14 and 15 show the past, current, future and counterfactual worlds of a generated instance, respectively.

In Figure 12 (past world), Object1 is an Employee that is hired by the Active Organization Object4. Aspect2 is the Employment relator with `salary` of Datatype0.

In Figure 13 (current world), Object1 no longer exists. Object4 turned into an Inactive Organization, now only with Employee Object2.

In Figure 14 (future world), no Employment relations are seen. All previous Persons and Organizations no longer exist. Object3 is a Person and Object0 is an Inactive Organization with no employees.

Finally, Figure 15 (counterfactual world) presents an alternate timeline from the past world. Here, Object1 still exists, but is no longer employed. A different Employment is visualized, connecting the Inactive Organization Object0 to the Employee Object3. Also, three Person objects share the same `birthDate` attribute of Datatype1.

This scenario shows an instance very similar to that of scenario 2. Again, Inactive Organizations can be seen hiring Employees, reinforcing the notion that this detail might require special attention. The main goal of this scenario, however, was to simply exemplify the counterfactual world structure.

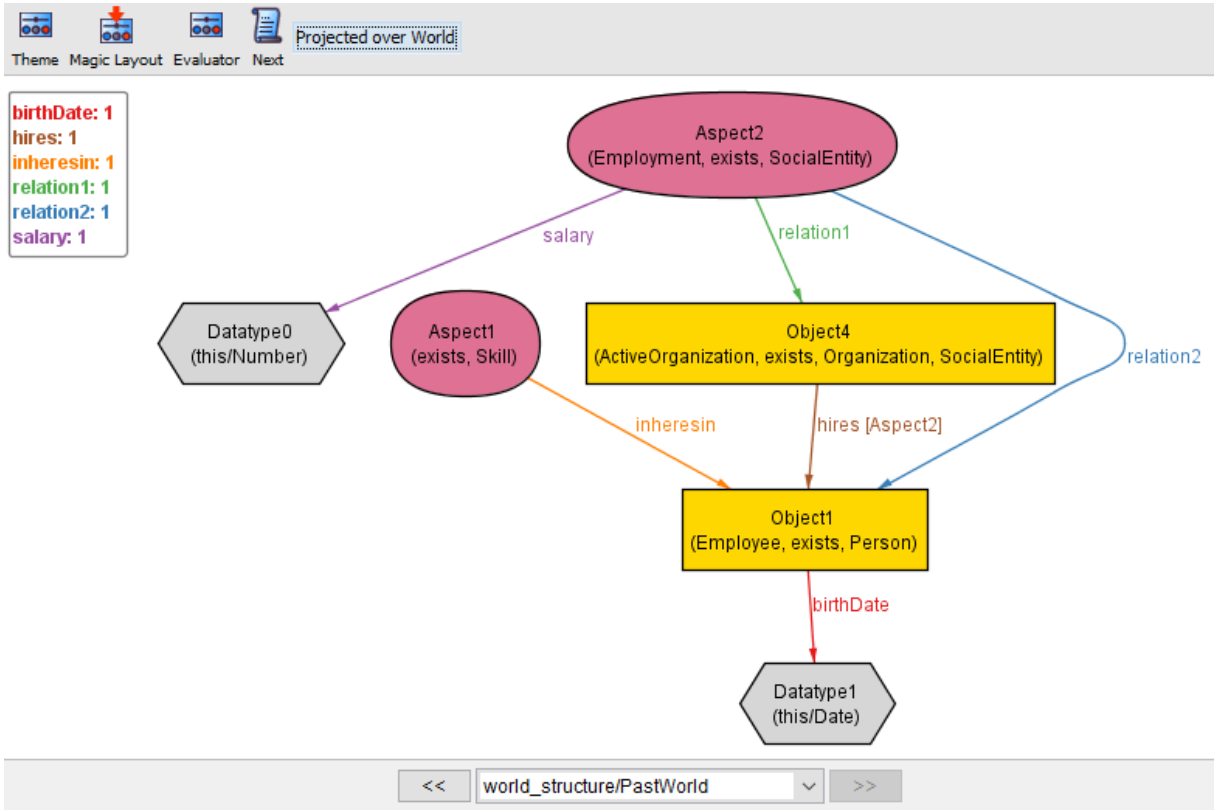

Figure 12 – Past world of the simulation scenario 3.

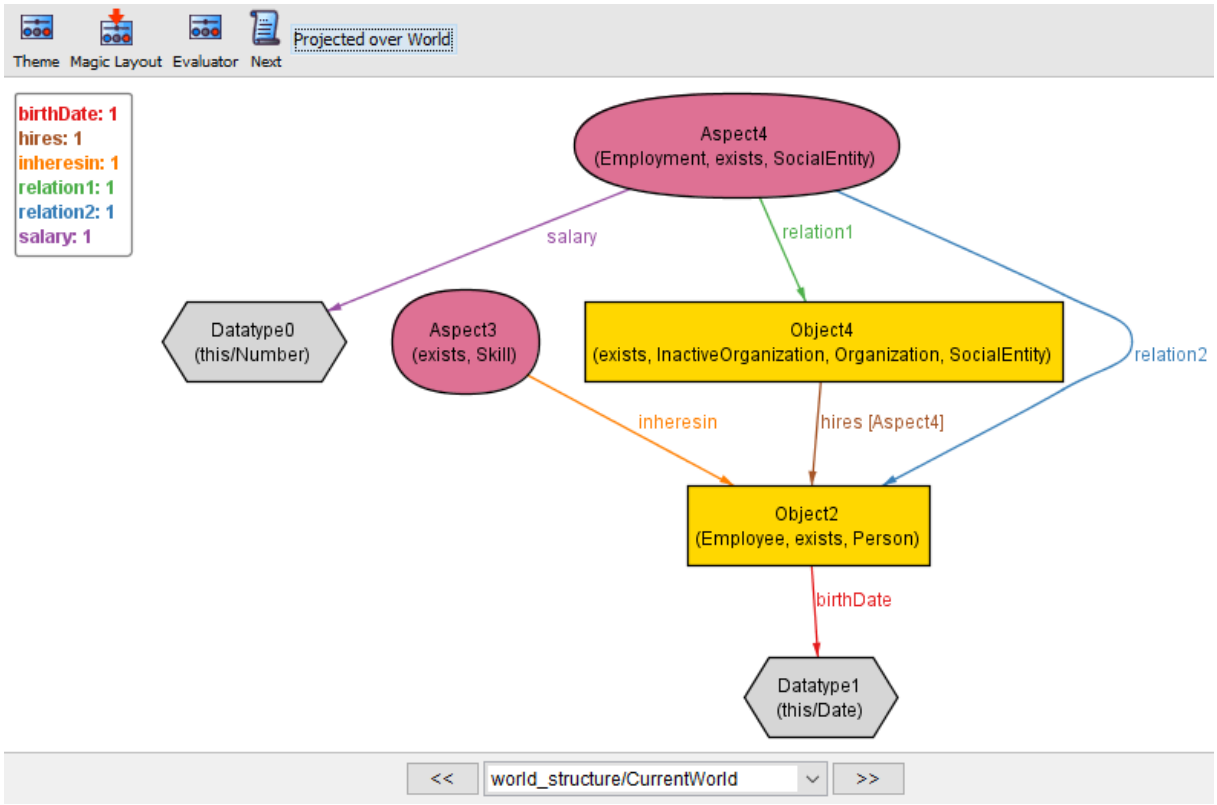

Figure 13 – Current world of the simulation scenario 3.

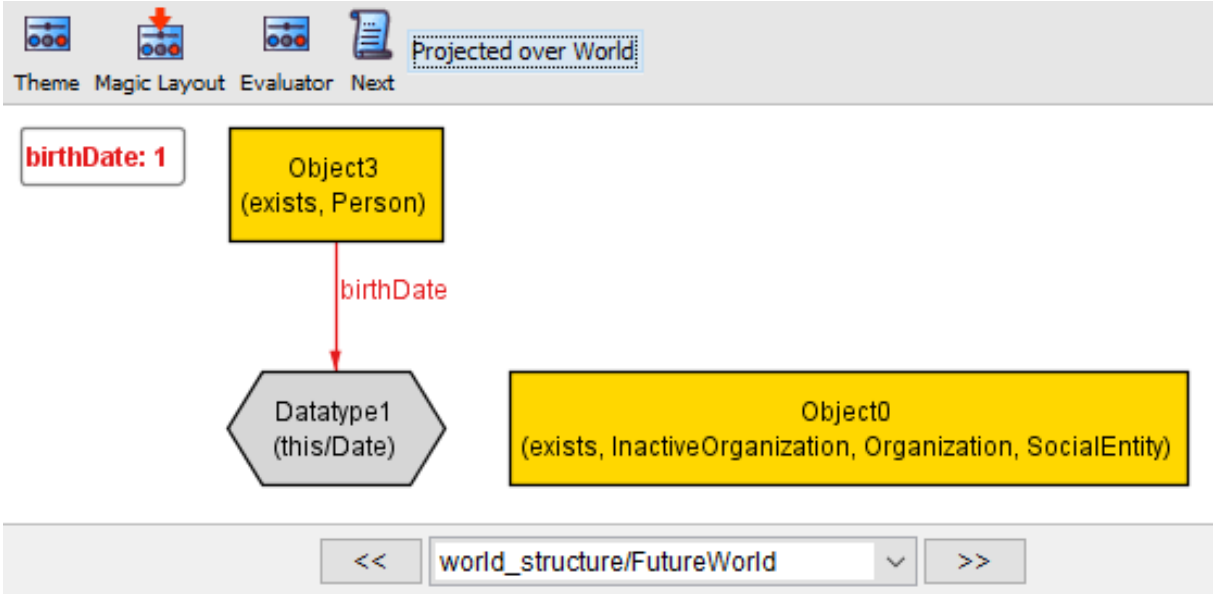

Figure 14 – Future world of the simulation scenario 3.

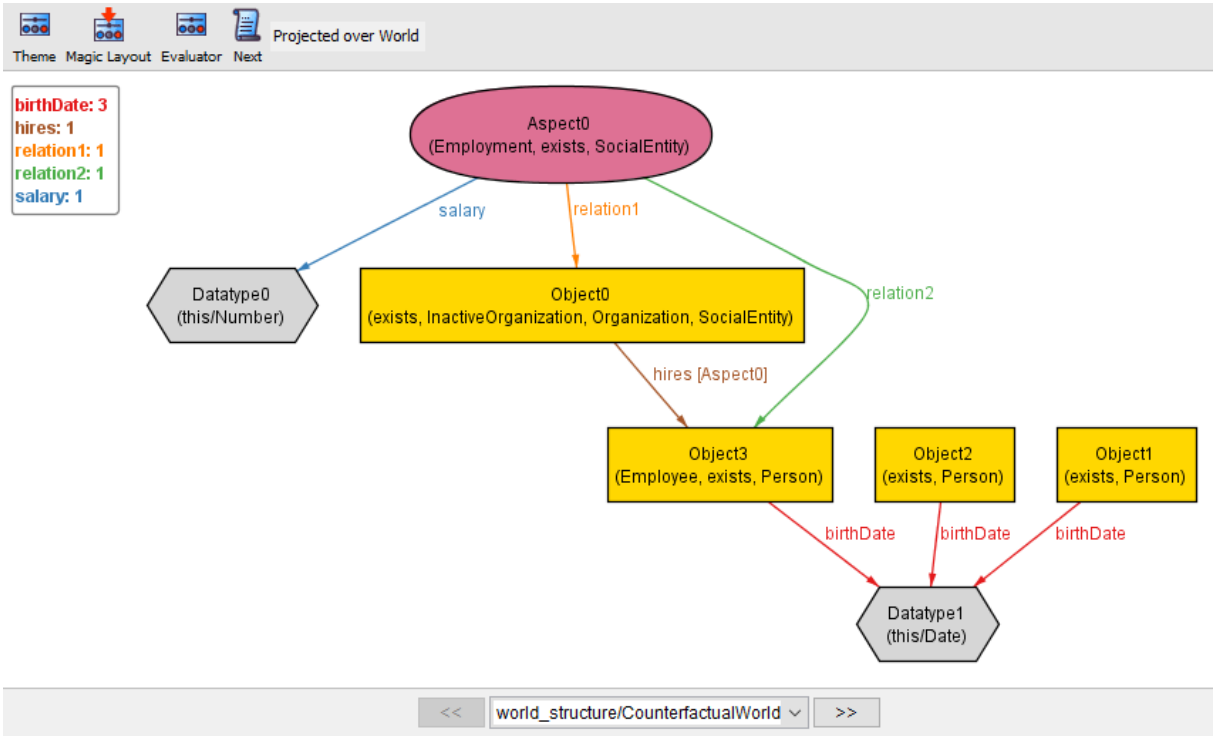

Figure 15 – Counterfactual world of the simulation scenario 3.

5 Implementation

This chapter briefly showcases the implementation process to code the OntoUML 2.0 to Alloy transformation using Typescript. Because of the change of frameworks and programming languages, the reimplementation of parts of the OntoUML to Alloy transformation (SALES, 2014) was necessary.

First of all, Typescript was chosen because the OntoUML Server is mostly being developed in this language. Thus, many utilities such as model parsing and model handling are already implemented and can be used. Furthermore, the OntoUML Server has a model transformation service for gUFO, a lightweight implementation of UFO currently under development (ALMEIDA et al., 2019). The patterns and code structure used in the implementation of this service were highly influential to the implementation of the transformation to Alloy.

The code is structured with the following files:

- ontouml2alloy.ts
- class_functions.ts
- relation_functions.ts
- property_functions.ts
- relation_functions.ts
- generalization_functions.ts
- generalization_set_functions.ts
- util.ts
- index.ts

The ontouml2alloy.ts file contains the *Ontouml2Alloy* class that implements the *Service* interface. This class stores the world field declarations, world field facts, fact blocks, functions, datatypes, enumerations and some additional information, used when generating the Alloy source code. An instance of this class is generated in the beginning of every transformation and is used by all the files suffixed with “functions”. The model is provided in the constructor of this class and is essentially a tree of OntoUML elements (represented as classes). This model structure was already implemented in the context of the OntoUML Server project.

The most important function of the *Ontouml2Alloy* class is *transform()* (refer to Listing 5.1). This function calls functions for every element type (lines 2–6). Then, after all elements

are transformed, the main module is written (lines 14–20), followed by the world structure and ontological properties modules (lines 22 and 23).

Listing 5.1 – Fragment with the transform() function of the ontouml2alloy.ts file

```
1  transform() {
2      this.transformClasses();
3      this.transformGeneralizations();
4      this.transformGeneralizationSets();
5      this.transformProperties();
6      this.transformRelations();
7
8      // removes possible duplicate facts and funs
9      this.worldFieldFacts = [...new Set(this.worldFieldFacts)];
10     this.facts = [...new Set(this.facts)];
11     this.relationPropertiesFacts = [...new Set(this.relationPropertiesFacts)];
12     this.funs = [...new Set(this.funs)];
13
14     this.writePreamble();
15     this.writeDatatypes();
16     this.writeEnums();
17     this.writeWorldSignature();
18     this.writeFacts();
19     this.writeFuns();
20     this.writeRuns();
21
22     this.writeWorldStructureModule();
23     this.writeOntologicalPropertiesModule();
24 }
```

As an example, the *transformClasses()* function (refer to Listing 5.2) loops through every class in the model, calling the *transformClass()* function for each one (lines 4–6). This function is defined in the class_functions.ts file and will be shown shortly. Afterwards, two functions are called to transform the additional facts for all classes and datatypes (lines 8 and 9).

Listing 5.2 – Fragment with the transformClasses() function of the ontouml2alloy.ts file

```
1  transformClasses() {
2      const classes = this.model.getAllClasses();
3
4      for (const _class of classes) {
5        transformClass(this, _class);
6      }
7
8      transformAdditionalClassConstraints(this);
9      transformAdditionalDatatypeConstraints(this);
10
11     return true;
12   }
```

The files suffixed with “functions” contain the functions used to transform specific OntoUML elements. In class_functions.ts, for instance, the function *transformClass* (refer to Listing 5.3 is called to decide which type of class is being transformed. Additional functions are then called depending on the class’s stereotype and meta-attributes.

Listing 5.3 – Fragment of the transformClass() function of the class_functions.ts file

```
1  export function transformClass(transformer: Ontouml2Alloy, _class: Class) {
2    if (_class.hasAnyStereotype([ClassStereotype.EVENT, ClassStereotype.SITUATION])) {
3      return;
4    }
5
6    if (_class.hasDatatypeStereotype()) {
7      transformDatatypeClass(transformer, _class);
8      return;
9    }
10
11   if (_class.hasEnumerationStereotype()) {
12     transformEnumerationClass(transformer, _class);
13     return;
14   }
15
16   if (_class.isRestrictedToEndurant()) {
17     transformEndurantClass(transformer, _class);
18   }
19
20   if (_class.hasRelatorStereotype()) {
21     transformRelatorConstraint(transformer, _class);
22   }
23
24   if (_class.isAbstract) {
25     transformAbstractClass(transformer, _class);
26   }
27
28   transformWeakSupplementationConstraint(transformer, _class);
29   transformDisjointNaturesConstraint(transformer, _class);
30 }
```

Consider an endurant class, for instance. In line 17 of Listing 5.3, the transformEndurantClass() function will be called (refer now to Listing 5.4). The first thing this function does is decide the nature of the class, according to the Alloy signatures `Object`, `Aspect` or `Endurant` presented previously (lines 5-11). Then, a world field declaration is added to the transformer (the Ontouml2Alloy instance that contains the model and all the fragments being generated by the transformation). Note that the decision here was to represent everything as strings, that later on will all be concatenated to form the entire Alloy specification model.

Next, lines 17 to 32 will generate a `rigid` fact if the class is top-level and rigid, depending on its stereotype, an `antirigid` fact if it is a top-level antirigid class, or none.

Listing 5.4 – Fragment of the transformEndurantClass() function of the class_functions.ts file

```
1  function transformEndurantClass(transformer: Ontouml2Alloy, _class: Class) {
2    const className = getNameNoSpaces(_class);
3    let nature = '';
4
5    if (_class.isRestrictedToSubstantial()) {
6      nature = 'Object';
7    } else if (_class.isRestrictedToMoment()) {
8      nature = 'Aspect';
```

```
9   } else {
10    nature = 'Endurant';
11  }
12
13  transformer.addWorldFieldDeclaration(
14    className + ': set exists:>' + nature
15  );
16
17  if (isTopLevel(_class, transformer.model.getAllGeneralizations())) {
18    if (_class.hasRigidStereotype()) {
19      transformer.addFact(
20        'fact rigid {\n' +
21        '        rigidity[' + className + ',' + nature + ',exists]\n' +
22        '}'
23      );
24    } else if (_class.hasAntiRigidStereotype()) {
25      transformer.addFact(
26        'fact antirigid {\n' +
27        '        antirigidity[' + className + ',' + nature + ',exists]\n' +
28        '}'
29      );
30    }
31  }
32 }
```

This logic is repeated for every OntoUML model element. The full implementation can be found in: <https://github.com/fernandoam14/ontouml-js>.

While the transformation has been entirely coded using OntoUML Server's packages and classes, the deployment process responsible to offer it as a service is currently ongoing. Any tests done in the development process were done separately, but according to the model structure presented in the ontouml-js[1] library used by the OntoUML Server.

Lastly, it is worth to point out that some important decisions were made regarding naming conventions. OntoUML does not enforce naming classes, attributes and relations. Also, named elements may contain spaces and other special characters, unsupported by Alloy. The approach used to work around this issue does not cover every possible situation and can be enhanced in future works. For named elements, any spacing is removed. For unnamed relations and properties (attributes and relation ends), an alias is provided with a number to differentiate each one. For instance, two unnamed relations would be called relation1 and relation2. Finally, unnamed classes are not handled.

[1] Available at: <https://github.com/OntoUML/ontouml-js>

6 Conclusion

6.1 Contribution

This work contributes to the theory and practice of ontology-driven conceptual modeling with an adjustment of the core parts of the OntoUML to Alloy transformation (SALES, 2014) to contemplate the new ontological properties of OntoUML 2.0, focused specifically on UFO-A aspects.

Moreover, the studied scenarios show important evidence that model simulation is a powerful tool for model validation. This confirms our motivation that Alloy can be used by modelers to improve the quality of their OntoUML models.

In addition, the Typescript code produced for the transformation was developed under the OntoUML Server framework and its integration as a service to be provided is ongoing. Thereby, an updated OntoUML 2.0 to Alloy transformation tool will be available to use by modelers to allow model verification and model validation.

Furthermore, The OntoUML Server is currently used by other projects, such as the Visual Paradigm OntoUML plugin[1]. The transformation developed in this work can also be included as a functionality of the plugin, which already includes features such as model serialization to JSON and model transformation to gUFO. With this, modelers will have a centralized work environment for OntoUML 2.0 modeling.

6.2 Final Considerations

This project greatly contributed to the consolidation of conceptual modeling, model-driven development and ontology-driven conceptual modeling theory and model transformation techniques studied for three years, during the course of two Scientific Initiation projects and this graduation project.

Time limitations and other external factors did not make possible the development of a robust framework such as the one proposed in Sales's M.Sc. Thesis (SALES, 2014). Nevertheless, there is a great feeling of accomplishment. The academic journey is tough and all these years working in this environment provided great experience and growth. Doing research in the Software Engineering field helped finding my academical place and gave me ideas and aspirations for a possible Master's degree in the future.

[1] Available at: <https://github.com/OntoUML/ontouml-vp-plugin>

6.3 Future Work

Due to limitations mentioned previously, UFO-B events were not taken into consideration for the transformation developed in the context of this research project. Also, some constraints related to part-whole relations and other kinds of derivations were not supported and should be explored in a future work. Subsetted and redefined properties were also not considered. Additionally, the naming convention rules chosen for naming classes, attributes and relators does not cover every case possible in OntoUML modeling. And, finally, the transformation was not designed to contemplate partial models due to the idea of integrating it to the Visual Paradigm OntoUML plugin, which currently verifies the syntax of OntoUML models ensuring they are complete, following strict syntactic rules. Thus, this transformation can then be enhanced by these new features and by more thorough testing cases to sharpen rough edges, providing an even better tool for OntoUML modelers.

Furthermore, the deployment of the service in the OntoUML Server should open a number of work opportunities. For instance, the inclusion of an updated transformation of OCL to Alloy to supplement the work done with the OntoUML 2.0 to Alloy transformation is encouraged, since many OntoUML 2.0 models are subject to OCL constraining.

In the near future, the incorporation of the transformation as a feature of the Visual Paradigm OntoUML plugin can also encourage the addition of transformation parameters, used to customize the simulation. Some examples of parametrization are: the inclusion of facts for additional constraining; the definition of custom scopes; and the enforcement of antirigidity behavior in the simulations and other UFO axioms, such as Weak Supplementation and the Identity Principle. In addition, the Alloy Analyzer interface can also be integrated to the plugin for an even more compact modeling experience.

Bibliography

ALMEIDA, J. P. A. et al. *gUFO: A Lightweight Implementation of the Unified Foundational Ontology (UFO)*. 2019. <http://purl.org/nemo/doc/gufo>. Citado na página 44.

BENEVIDES, A. B. et al. Validating modal aspects of ontouml conceptual models using automatically generated visual world structures. *Journal of Universal Computer Science*, v. 16, p. 2904–2933, 2011. Citado 3 vezes nas páginas 13, 14, and 21.

BRAGA, B. F. B. et al. Transforming OntoUML into Alloy: Towards conceptual model validation using a lightweight formal method. *Innovations in Systems and Software Engineering (ISSE)*, Springer-Verlag, v. 6, n. 1, p. 55–63, 2010. Citado 3 vezes nas páginas 13, 14, and 24.

FONSECA, C. M. *ML2: An Expressive Multi-Level Conceptual Modeling Language*. Dissertação (Mestrado) — Universidade Federal do Espírito Santo, 2017. Citado na página 13.

FONSECA, C. M. et al. Relations in ontology-driven conceptual modeling. In: *38th International Conference on Conceptual Modeling (ER 2019), LNCS, 2019. v. 11788*. [S.l.]: Springer, 2019. p. 1–15. Citado 2 vezes nas páginas 14 and 18.

GUIZZARDI, G. *Ontological foundations for structural conceptual models*. Tese (Doutorado) — University of Twente, 2005. Citado 4 vezes nas páginas 12, 15, 16, and 18.

GUIZZARDI, G.; FALBO, R. de A.; GUIZZARDI, R. S. Grounding software domain ontologies in the unified foundational ontology (ufo): The case of the ode software process ontology. In: CITESEER. *CIbSE*. [S.l.], 2008. p. 127–140. Citado na página 15.

GUIZZARDI, G. et al. Types and Taxonomic Structures in Conceptual Modeling: A Novel Ontological Theory and Engineering Support. *Data & Knowledge Engineering*, accepted, forthcoming, 2021. ISSN 0169023X. Citado 2 vezes nas páginas 14 and 16.

GUIZZARDI, G. et al. Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: *Conceptual Modeling - 37th International Conference, ER 2018*. [S.l.]: Springer, 2018. p. 136–150. Citado 2 vezes nas páginas 12 and 16.

GUIZZARDI, G. et al. Towards Ontological Foundations for the Conceptual Modeling of Events. In: *Proc. 32th International Conference on Conceptual Modeling (ER)*. [S.l.]: Springer, 2013. (Lecture Notes in Computer Science, v. 8217), p. 327–341. Citado na página 15.

JACKSON, D. *Software Abstractions: Logic, Language and Analysis – Revised Edition*. [S.l.]: MIT Press, 2012. ISBN 978-0-262-01715-2. Citado 4 vezes nas páginas 13, 14, 19, and 20.

MYLOPOULOS, J. Conceptual modeling and telos. In: *Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development*. [S.l.]: Wiley, 1992. p. 49–68. Citado na página 12.

SALES, T. P. *Ontology Validation for Managers*. Dissertação (Mestrado) — Universidade Federal do Espírito Santo, 2014. Citado 7 vezes nas páginas 13, 14, 21, 24, 38, 44, and 48.

WARMER, J.; KLEPPE, A. *The Object Constraint Language: Getting Your Models Ready for MDA*.

2. ed. [S.l.]: Addison-Wesley, 2003. (Object Technology Series). ISBN 978-0-321-17936-4. Citado 2 vezes nas páginas 13 and 40.

Appendix

Listing 1 – Full Alloy specification of the transformed OntoUML running example model.

```
module main

open world_structure[World]
open ontological_properties[World]
open util/relation
open util/sequniv
open util/ternary

abstract sig Endurant {}

sig Object extends Endurant {}

sig Aspect extends Endurant {}

sig Datatype {}

sig Number in Datatype {}

sig Date in Datatype {
    day: one Number,
    month: one Number,
    year: one Number
}

abstract sig World {
    exists: some Endurant,
    SocialEntity: set exists:>Endurant,
    Person: set exists:>Object,
    Organization: set exists:>Object,
    Employment: set exists:>Aspect,
    Employee: set exists:>Object,
    ActiveOrganization: set exists:>Object,
    InactiveOrganization: set exists:>Object,
    Skill: set exists:>Aspect,
    birthDate: set Person set -> one Date,
    salary: set Employment set -> one Number,
    hires: set Organization -> Employment -> Employee,
    relation1: set Employment -> one Organization,
    relation2: set Employment -> one Employee,
    inheresin: set Skill -> one Employee
} {
    disjoint[SocialEntity,(Number+Date+Skill)]
    disjoint[Person,(Number+Date+SocialEntity+Skill)]
    disjoint[Skill,(Number+Date+SocialEntity+Person)]
    exists:>Object in SocialEntity+Person
    exists:>Aspect in SocialEntity+Skill
}

fact additionalFacts {
    continuous_existence[exists]
    elements_existence[Endurant,exists]
}

fact relationProperties {
    immutable_target[Employment,relation1]
```

```
56      immutable_target[Employment, relation2]
57      immutable_target[Skill, inheresin]
58  }
59
60  fact rigid {
61      rigidity[SocialEntity, Endurant, exists]
62  }
63
64  fact abstractClass {
65      all w: World | w.SocialEntity = w.Organization+w.Employment
66  }
67
68  fact rigid {
69      rigidity[Person, Object, exists]
70  }
71
72  fact relatorConstraint {
73      all w: World, x: w.Employment | #(Organization1[x,w]+Employee1[x,w])>=2
74  }
75
76  fact rigid {
77      rigidity[Skill, Aspect, exists]
78  }
79
80  fact additionalDatatypeFacts {
81      Datatype = Number+Date
82      disjoint[Number, Date]
83  }
84
85  fact generalization {
86      Employee in Person
87  }
88
89  fact generalization {
90      Organization in SocialEntity
91  }
92
93  fact generalization {
94      Employment in SocialEntity
95  }
96
97  fact generalization {
98      ActiveOrganization in Organization
99  }
100
101 fact generalization {
102     InactiveOrganization in Organization
103 }
104
105 fact generalizationSet {
106     disjoint[ActiveOrganization, InactiveOrganization]
107     Organization = ActiveOrganization+InactiveOrganization
108 }
109
110 fact multiplicity {
111     all w: World, x: w.Employee | #Organization2[x,w]>=1
112 }
```

```
113
114 fact multiplicity {
115     all w: World, x: w.Organization | #Employee2[x,w]>=0
116 }
117
118 fact acyclic {
119     all w: World | acyclic[w.relation1,w.Employment]
120 }
121
122 fact acyclic {
123     all w: World | acyclic[w.relation2,w.Employment]
124 }
125
126 fact derivation {
127     all w: World, x: w.Organization, y: w.Employee, r: w.Employment |
128             x -> r -> y in w.hires iff x in r.(w.relation1) and y in r.(w.relation2)
129 }
130
131 fact acyclicCharacterizations {
132     all w: World | acyclic[(w.inheresin),(w.Skill)]
133 }
134
135 fun visible : World->univ {
136     exists+select13[birthDate]+select13[salary]
137 }
138
139 fun birthDate1 [x: World.Person, w: World] : set Date {
140     x.(w.birthDate)
141 }
142
143 fun salary1 [x: World.Employment, w: World] : set Number {
144     x.(w.salary)
145 }
146
147 fun Organization2 [x: World.Employee, w: World] : set World.Organization {
148     (select13[w.hires]).x
149 }
150
151 fun Employee2 [x: World.Organization, w: World] : set World.Employee {
152     x.(select13[w.hires])
153 }
154
155 fun Employment1 [x: World.Organization, w: World] : set World.Employment {
156     (w.relation1).x
157 }
158
159 fun Organization1 [x: World.Employment, w: World] : set World.Organization {
160     x.(w.relation1)
161 }
162
163 fun Employment2 [x: World.Employee, w: World] : set World.Employment {
164     (w.relation2).x
165 }
166
167 fun Employee1 [x: World.Employment, w: World] : set World.Employee {
168     x.(w.relation2)
169 }
```

```
170
171 fun Skill1 [x: World.Employee, w: World] : set World.Skill {
172     (w.inheresin).x
173 }
174
175 fun Employee3 [x: World.Skill, w: World] : set World.Employee {
176     x.(w.inheresin)
177 }
178
179 -- Suggested run predicates
180 run singleWorld for 10 but 1 World, 7 Int
181 run linearWorlds for 10 but 3 World, 7 Int
182 run multipleWorlds for 10 but 4 World, 7 Int
183 run singleWorld for 20 but 1 World, 7 Int
184 run linearWorlds for 20 but 3 World, 7 Int
185 run multipleWorlds for 20 but 4 World, 7 Int
```