Universidade Federal do Espírito Santo, Centro Tecnológico

Programa de Pós-Graduação em Informática

John Guerson

Representing Dynamic Invariants in Ontologically Well-Founded Conceptual Models

Vitória - ES, Brazil

May 2015

John Guerson

Representing Dynamic Invariants in Ontologically Well-Founded Conceptual Models

Dissertação apresentada ao Programa de Pós Graduação em Informática da Universidade Federal do Espírito Santo como requisito parcial para obtenção do título de Mestre em Informática.

ORIENTADOR

PROF. DR. JOÃO PAULO ANDRADE ALMEIDA

PROGRAMA DE PÓS GRADUAÇÃO EM INFORMÁTICA,
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Vitória - ES, Brazil

May 2015

Dissertação de Mestrado sob o título "Representing Dynamic Invariants in Ontologically Well-Founded Conceptual Models", defendida por John Guerson e aprovada em 28 de Maio, 2015, em Vitória, Estado do Espírito Santo, pela banca examinadora constituída pelos doutores:

Prof. Dr. João Paulo Andrade Almeida

Departamento de Informática - UFES

Orientador

Prof. Dr. Giancarlo Guizzardi

Departamento de Informática - UFES

Examinador Interno

Prof. Dr. Clever Ricardo Guareis de Farias

Departamento de Computação – USP/Ribeirão Preto

Examinador Externo

Acknowledgements

I would like to thank God, for His deep love and mercy. Thank you Jesus, for your grace.

I thank my parents (Nélio and Cristina) and my sisters (Susie, Dianne and Cindy) for their heart, complicity and friendship in every adversity, I love you.

I would like to honor my advisor João Paulo for his knowledge, wisdom and patience. I am glad I had the chance of working with you for two (more) years and benefit from all of your guidance.

A special thanks to Júlio Nardi and Victorio Carvalho for their enormous support not only in the academy but especially outside of it, You guys rock.

I would like to thank my fellow colleague and friend Tiago Sales, with whom I had the pleasure of working with in the past two years. It is incredible doing research with you too. The tools we have developed are just a shadow of what we can accomplish working together.

Finally, I would like to thank each single one from our lab, with whom I had the pleasure of meeting and living with in a daily basis. Just to name a few Diorbert Pereira, Vinicius Sobral, Ernani Santos, Laylla Duarte, Freddy Brasileiro, Victor Amorim, Cássio Reginato, Pedro Paulo Barcelos, Lucas Bassetti, Bernardo Braga and Jordana Salamon.

Resumo

Modelos conceituais frequentemente capturam os aspectos invariantes dos fenômenos que nós percebemos. Estes invariantes podem ser considerados estáticos quando se referem a estruturas que nós percebemos do fenômeno em um ponto particular do tempo ou dinâmicos/temporais quando se referem a regularidades entre pontos diferentes do tempo. Enquanto invariantes estáticos têm recebido uma atenção significativa, invariantes dinâmicos têm recebido um suporte marginal em técnicas amplamente adotadas tais como UML e OCL. Este trabalho tem por objetivo abordar esta lacuna propondo uma técnica para a representação de invariantes dinâmicos de domínio em modelos conceituais baseados em UML. Para esse propósito, uma extensão temporal de OCL é proposta. Ela enriquece o perfil ontologicamente bem fundamentado OntoUML e permite a expressão de uma variedade de restrições temporais arbitrárias. A extensão é completamente implementada em uma ferramenta para especificação, verificação e simulação de modelos OntoUML temporalmente enriquecidos.

Abstract

Conceptual models often capture the invariant aspects of the phenomena we perceive. These invariants may be considered static when they refer to structures we perceive in phenomena at a particular point in time or dynamic/temporal when they refer to regularities across different points in time. While static invariants have received significant attention, dynamics enjoy marginal support in widely-employed techniques such as UML and OCL. This thesis aims at addressing this gap by proposing a technique for the representation of dynamic invariants of subject domains in UML-based conceptual models. For that purpose, a temporal extension of OCL is proposed. It enriches the ontologically well-founded OntoUML profile and enables the expression of a variety of (arbitrary) temporal constraints. The extension is fully implemented in the tool for specification, verification and simulation of temporal enriched OntoUML models.

"For now we see in a mirror dimly, but then face to face;
now I know in part, but then I will know fully
just as I also have been fully known"
I Corinthians 13:12

List of Figures

List of Listings

List of Axioms

List of Definitions

List of Tables

Contents

1 Introduction

In this chapter, we introduce the subject of ontology-driven conceptual modeling and the study of its dynamic aspects. In particular, we motivate the study within this research presenting a brief background of the related field. Then we define the general and specific objectives of this research alongside with contributions to the area. Finally, we present the approach used to achieve such results.

1.1 Background

In a broad perspective, 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" [1]. Many of the efforts in conceptual modeling attempt to represent a conceptualization about a given subject domain [2], which is often accomplished by capturing in a model the invariant aspects of the phenomena we perceive. These invariants may be considered *static* when they refer to structures we perceive in phenomena at a particular point in time or *dynamic* when they refer to regularities across different points in time.

Take for instance a domain about persons, their stages in life and their marriages. At a particular point in time, a number of persons will exist, each of which may be male or female, may be a child, a teenager or an adult, and may be related to someone else by marriage. The static invariants that may be represented in a conceptual model of this domain include the various categories of entities in a domain (in our example, "person", "male", "female", "child", "teenager", "adult", "elder", "marriage") as well as their relations (a "child" is a "person", "marriage" may be established between two "persons", etc.). The dynamic invariants in turn reflect the fact that across different points in time entities of the domain undergo change. In our example, persons are born and die, become teenagers and adults, marry, divorce, etc. Dynamic invariants represent what may change and what must remain constant in time. For example, children cannot suddenly become adults, adults cannot later in life become teenagers and elders cannot become children, teenagers or adults.

Much attention has been paid to the representation of static invariants in a number of modeling notations including ER diagrams, ORM diagrams [3], and UML Class Diagrams [4]. The UML for example has been enriched with the Object Constraint Language (OCL) to capture static invariant expressions [5]. With respect to the dynamic invariants, these have been mostly confined to the representation of OCL pre- and post-conditions for operations or simple UML meta-attributes for features such as "read only" [4, p.125, 129]. Further, due to the strict correspondence that is often established between modeling languages and programming languages, many UML-based approaches lack support for dynamic classification (e.g. USE [6], HOL-OCL [7], UML2Alloy [8, 9]). While this facilitates the mapping to specific programming languages or formalisms, this renders these approaches less suitable to enable the expression of important conceptual structures that rely on dynamic classification (e.g., the classification of persons into life phases: child, teenager, and adult, the classification of persons into roles they play contingently such as husband and wife)[1].

In order to address the deficiencies of the UML and OCL specifications, many approaches have been proposed to extend UML and OCL with dynamic aspects. Some of these address dynamic aspects as part of an overall approach to handle temporal/time aspects [2, 7, 10, 11, 12, 13, 14, 15, 16, 17, 18]. The OntoUML [2], for example, introduces various dynamic aspects through stereotypes referring to meta-attributes of classes and properties such as rigidity and immutability. Similarly, [12] extends UML with stereotypes, augmenting it with dynamic notions of durability and frequency. Others have aimed at enriching OCL with extensions in order to cope with dynamic/temporal properties of systems. For example, some have extended OCL with Linear-Temporal Logic and Computational-Tree logic (LTL/CTL) operators [10, 13, 17, 18], created new logic formalisms [11, 14], extended OCL with temporal patterns [16], defined a Real-Time extension for OCL with a temporalized CTL [15], etc.

[1] Note that while dynamic classification is supported in principle by UML diagrams, this is not reflected in tool support and language usage, with little mention in the UML specification.

Despite these recent advances, most approaches are not adequate in the representation of dynamic aspects at the conceptual level. This gap is addressed in this research, in which we support the expression of rich dynamic constraints in ontologically well-founded conceptual models written with OntoUML.

1.2 Objectives

The overall aim of this research is the representation of dynamic constraints in ontologically well-founded structural conceptual models. As specific objectives of this research, we aim at: (1) increasing the expressivity of the diagrammatic notation of OntoUML with the inclusion of pre-defined dynamic aspects, considering the amount of complexity that is reasonable to introduce in a diagrammatic notation; (2) complementing the graphical representation of OntoUML with a textual language to enable the representation of richer dynamic constraints; and, finally, (3) extending a formal validation approach [19] to support the modeler in assessing whether the resulting conceptual model represents his/her domain conceptualization.

1.3 Research Approach

In order to identify the opportunities to extend OntoUML with dynamic aspects (specific objective 1) it is necessary to understand which properties of an OntoUML model refer to purely static aspects and which properties refer to dynamic aspects. Important aspects of OntoUML include its support for dynamic classification (e.g. the classification of persons into life phases: child, teenager, and adult) and for modal meta-properties of classes and associations such as rigidity and immutability. With an understanding of what there is, we can explore the barriers regarding the definition of new dynamic aspects for OntoUML. We aim at proposing a simple extension for OntoUML to capture some additional dynamic aspects which are recurrent in some conceptualizations about subject domains, based on different views of reality, according to different philosophical theories about time and existence such as the Presentism theory and the Growing Block Universe theory [21]. The additional dynamics introduced in OntoUML is important to represent as accurately as possible conceptualizations based on any of these theories.

In order to address textual dynamic constraints (specific objective 2) and complement OntoUML's diagrammatic notation, we identify existing approaches on temporal (constraint-based) conceptual modeling to judge their adequacy in representing dynamics alongside OntoUML. Important dynamic aspects of a temporal constraint language include constraining the order in which an individual instantiate types (the change of instantiations allowed for an individual regarding different types, e.g. an adult later in time becomes an elder but not a child), constraining the way individuals can exist in time (if they can exist permanently or must cease to exist at a certain time in time), defining that some population of individuals are derived from the past (e.g. ex-spouses are people who participated in a past marriage, which no longer exists in the present) [20], and specifying historical dependence facts [21] (facts that cross the present time, for example, my grandfather, which does not exist anymore, cannot be a descendant of itself). Similarly to many approaches that have extended OCL to represent temporal/dynamic properties e.g. [10, 16, 18], we also propose an extension of OCL in order to enrich OntoUML models with arbitrary dynamic aspects of subject domains. This should facilitate the adoption of the approach by UML/OCL modelers.

Lastly, in order to assess whether the resulting dynamic-enhanced OntoUML model represents the modeler's conceptualization, it is necessary to extend the existent formal validation approach based on Alloy simulation and analysis [22, 23] with the inclusion of dynamic textual constraints written with a dynamic/temporal OCL language (specific objective 3). In this technique a conceptual model is translated into Alloy to be fed into the Alloy analyzer tool. Alloy [19] is a declarative language based on the notion of relations and first-order logics to describe and explore structure accompanied of an automatic tool called the Alloy Analyzer. In order to extend this approach, it is necessary to extend the current translation from OntoUML and (static) OCL to Alloy, with the support for a dynamic/temporal OCL language. Our aim is to leverage the use of a formal method and automatic analyzer, without exposing the complexities of the formal method to the user. The experience of exploring a conceptual model with an automatic analyzer (building a model incrementally with a continual, automatic review, simulating and checking as you go along) is

thrilling and humiliating as it is highly likely to reveal flaws and omissions. This is discussed by Jackson, the designer of Alloy, in [19, p. XIII]: "The sense of humiliation sets in, as you discover that there's almost nothing you can do right; what you write down doesn't mean exactly what you think it means, and when it does, it doesn't have the consequences you expected".

In summary, our modeling approach is required to:

- Support dynamic classification i.e., allow for individuals to change types throughout their existence (an assumption underlying OntoUML); (*Requirement 1*)
- Enable the expression of modal constraints on types e.g., rigidity, non-rigidity, anti-rigidity, immutability (mechanisms underlying OntoUML); (*Requirement 2*)
- Enable the expression of classifications rules, constraining the order in which individuals instantiate types e.g. elders cannot become children; (*Requirement 3*)
- Enable the expression of transient, permanent and eternal existence rules, constraining the way individuals exist in time (e.g. if they should cease to exist, if they can exist permanently); (*Requirement 4*)
- Enable the expression of derivations by past specializations [20] e.g. ex-spouses are derived from people who participated in a past marriage, which no longer exists in the present; (*Requirement 5*)
- Enable the expression of other arbitrary dynamic invariants, i.e., invariants whose satisfaction is determined by examining the world at more than one point in time; (*Requirement 6*) and
- Finally, enable the representation of historical relationships such as the "descendant" relationship between people at all times specifying that, for example, my father, a present entity, is a descendant of my grandfather which is a wholly past entity and no longer exists in the present, and the representation of historic dependence facts called as trans-temporal facts by [21] (e.g. a fact stating that people cannot be descendant of themselves); (*Requirement 7*)

In addition, our approach must not rely on operations of classes, as these are not employed by OntoUML [2] and should not employ specialized tense/temporal log-

ic-based operators [10, 13, 17, 18], in order to retain its ease of use for UML/OCL modelers. This enables the approach to be used by modelers that do not have an advanced level of logic expertise.

We exclude as a potential solution diagrammatic languages such as UML state chart diagrams, as we are aiming here a more general approach with the definition of *arbitrary* (user-defined) dynamic constraints.

1.4 Thesis Structure

The remaining of the thesis is structured as follows. Chapter 2 investigates the dynamic aspects implicit in the semantics of OntoUML structural conceptual models complemented with standard (static) OCL. We refer to Section 2.4 on the dynamics already captured by OntoUML. Chapter 3 introduces our set of additional dynamics for the OntoUML conceptual modeling language. We properly present and formally characterize each one of these dynamics set out as requirements in our modeling approach, demonstrating how they can be applied in practice, according to a set of philosophical theories about time and existence. Chapter 4 defines a temporal extension of OCL to complement OntoUML, proposed to capture the requirements set out previously in Section 1.3. Chapter 5 extends the existing ontology-driven formal validation approach based on Alloy to include dynamics written with our temporal OCL extension. Chapter 6 explains the tooling developed in this research. Chapter 7 discusses related work and Chapter 8 presents some concluding remarks.

2 Implicit Dynamic Aspects in Structural Conceptual Models

While structural models (such as UML class diagrams) are often thought of as representing only static aspects of a subject domain, some constructs are in fact used to capture dynamic aspects. This chapter examines such constructs for structural conceptual models written in OntoUML (as a UML profile).

As a means to establishing some conceptual basis for the rest of the work, we start by examining the relation between phenomena, conceptualizations and structural conceptual models (Section 2.1). We then formally characterize what a structural conceptual model written in plain UML represents about the phenomena it models (Sections 2.2 and Section 2.3). Later, we examine how OntoUML extends UML revealing the dynamic aspects that are implied by the various stereotypes and meta-attributes of OntoUML. We present OntoUML and its dynamic aspects by means of a running example (Section 2.4). Finally, we present some final considerations (Section 2.6) that motivate the need for additional dynamics in OntoUML since OntoUML's dynamics is currently limited to a set of pre-defined constraints implied by rigidity and immutability.

2.1 Phenomena, Conceptualization and Conceptual Model

"Conceptual Modeling is the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication." [1]. Conceptual modeling results in representations (or "descriptions") which we call conceptual models. These are not something merely abstract but are represented in some concrete form in order to be used by and shared among humans. In the case of *structural* conceptual models, they represent a category of *entities* of the domain that persist (exist) in time such as "Book", "Person", "Group of People", and the *relationships* between those entities such as a relationship "is married with" relating "husband" and "wife". A structural conceptual model defines a stakeholder's view (a person, a community of people) about an *abstraction* of phenomena being

perceived. The stakeholder's abstraction of phenomena (which is in his mind) is known as *conceptualization* (or domain conceptualization) and is captured in a concrete artifact called conceptual model. A model thus reflects one's view about the phenomena. These aspects captured by a conceptual model are called *invariants* of a subject domain. Figure 1 illustrates the relation between phenomena, intended conceptualization (as perceived by a stakeholder) and a conceptual model artifact.

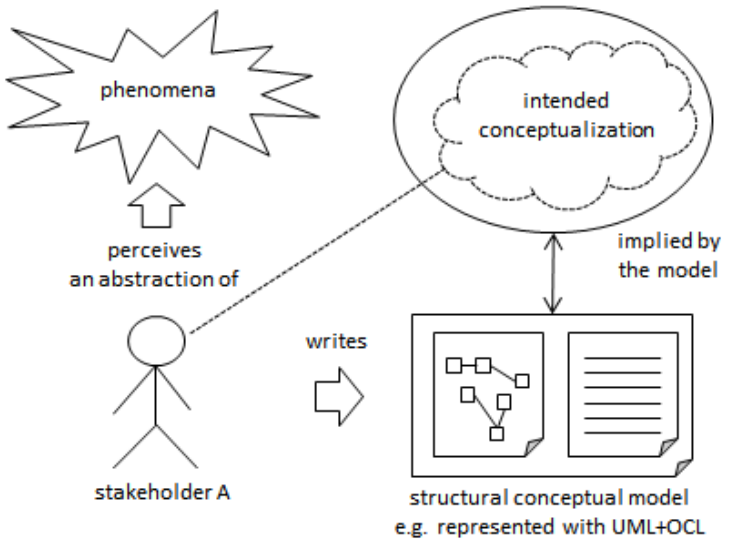

Figure 1 Phenomena, Conceptualization and Conceptual Model

Figure 1 shows that there is a tension between the intended conceptualization of a stakeholder and what is ultimately represented in the model. Figure 2 expands Figure 1 illustrating the tension between the intended conceptualization and the resulting model.

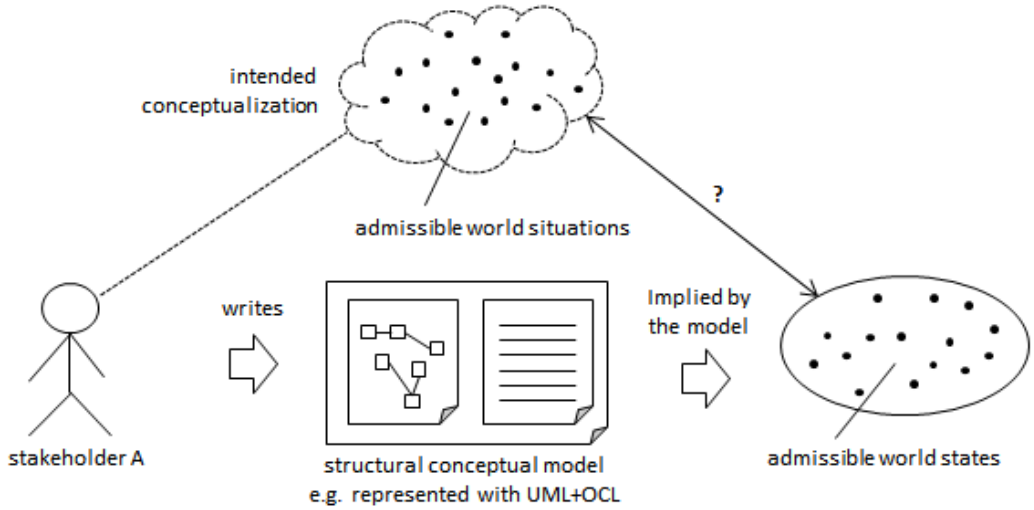

Figure 2 Tension between Intended Conceptualization and the Model

The model must represent as accurately as possible the intended conceptualization of the stakeholder. In other words, the model should ideally state only what the stakeholder intended to state about the portion of phenomena.

The conceptualization captures invariants about the phenomena that can be considered *static* when they refer to structures we perceive in phenomena at a particular point in time or *dynamic* when they refer to regularities across different points in time. For example, in a domain about people and marriages, a conceptualization may capture that in every situation of the world a person can marry with only one partner at a time. This conceptualization admits for instance a situation wherein Abraham is solely married with Sarah while it does not admit a situation wherein Abraham is married with both Sarah and Hagar, at the same time. Moreover, this conceptualization may capture that the marriage between Abraham and Sarah should exist forever while the two are alive. This conceptualization does not admit for instance a situation wherein Abraham and Sarah were not married, then another situation wherein they married and later on one wherein they cease to be married and are still alive. In this way, a conceptualization defines not only situations allowable at a time (static aspects) but also how things can behave (dynamic aspects).

Finally, a structural conceptual model is expressed in a conceptual modeling language. A conceptual modeling language must have a clear semantics in order to enable the accurate interpretation of its expressions (the models). Natural languages (such as English and Portuguese) cannot be used as conceptual modeling languages due to their ambiguity and although they are "understandable" and shared among humans, their automatic interpretation is problematic. Mathematical languages on the other hand lack the so-desired comprehensibility and understandability as the models in these languages should be used by and shared among humans for communication, problem solving, learning and understanding about a subject domain. In order to fill the gap between understandability and precision, there have been several different conceptual modeling notations (e.g. UML, OCL, and OntoUML) which embed mathematical axiomatizations underlying their (visual, textual) modeling constructs. In the sequel, we discuss that axiomatization i.e. we provide a formal struc-

ture for a conceptual model in order to give it a formal semantics. This notion is important for a precise understanding of what a conceptual model represents.

2.2 Formal Semantics of a Conceptual Model Structure

Our aim is to reveal the static and dynamic aspects that are represented in a model. To that end, the structure we use is an ordered couple ⟨W, D⟩, where W is a non-empty set of possible world states deemed admissible according to the conceptualization and D is the domain of quantification that includes all possible entities of a domain, including universals (or types) and their instances [2]. The structure is formally characterized by a system of modal logics called *alethic* (a logic of necessity and possibility) which can define contingent and necessary truths about the entities of the domain. All world states contained in the set W are equally accessible through a binary accessibility relation called R defined in W x W. This means that this accessibility relation links any two possible worlds, i.e. any world state w to any other world state w' (which can also include the very world w).

Let $w \in W$ be a specific world state and G a domain entity such that $G \in D$. The extension function ext(G, w) maps G (also called Universal) to the set of individuals of that concept that exist (i.e. are present) in world state w. The extension function ext(G) in turn provides a mapping to the set of individuals of the universal G that eventually exist in W i.e. that could exist in any possible world state [2, p.100, 101] as formalized in Definition 1.

Definition 1 Universal's Extension Function

$$\mathrm{ext(G)} \stackrel{\text{def}}{=} \cup\, \mathrm{w} \in \mathrm{W}, \mathrm{ext(G, w)}$$

The extension function maps G to the set of individuals of G that exist in a given world state. According to this definition, if an individual does not instantiate G in world, i.e. $x \notin \mathrm{ext}(G, w)$, we cannot state that x does not exist (i.e. is not present) in w since it might exist instantiating other universal U such that $x \in \mathrm{ext}(U, w)$. Therefore, let the predicate "existsIn" denote existence, we can formally state that an in-

dividual *x* exists in world state *w* iff there is at least one universal *G* such that *x* belongs to the *G*'s extension as formalized in Definition 2.

Definition 2 Individual's Existence Function

$$\text{existsIn}(x, w) \stackrel{\text{def}}{=} \exists\, G, x \in \text{ext}(G, w)$$

$$x \in \text{ext}(G, w) \rightarrow \text{existsIn}(x, w)$$

Consequently, if *x* belongs to a universal's extension in world *w* then *x* also exists at *w*. In the following, we will use the extension and existence functions, in order to define the statements that are represented in a conceptual model represented in plain UML.

2.3 Conceptual Models Represented as UML Class Diagrams

The Unified Modeling Language (UML) [4] is a language initially proposed as a unification of several different visual notations and modeling techniques used for systems design [24]. According to [12], the UML is a non-temporal conceptual modeling language. Thus, a UML class diagram represents the actual state of a system assuming that the "information base" contains only the current instances of classes and relationships. The language has become a de facto standard for conceptual modeling, proposed as an ontology representation language [24, 25].

Although UML class diagrams define in principle only the *static* aspects of conceptualizations, a number of language constructs may be given a temporal interpretation. Consider, for example, UML multiplicities, which may raise different temporal interpretations.

Figure 3 depicts a UML class diagram about people and marriages demonstrating the use of multiplicities with UML. A UML multiplicity defines how many elements are valid on a set that an entity of the domain is linked to. For instance, a multiplicity of "2" from a domain entity "Marriage" to "Person" means that a marriage must be linked to exactly two persons, and the multiplicity "*" (or "0..*") from "Person" to "Marriage" that a person must be linked to any number of marriages (the star

character "*" is used to represent an infinite upper bound meaning that a person may be linked to an unbound number of marriages).

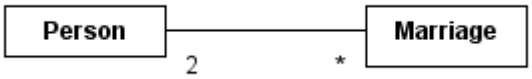

Figure 3 UML Class Diagram: People and Marriages

From a pure conceptual modeling point of view, it is reasonable to interpret the diagram of Figure 3 in two different ways. First, a person (let's say Abraham) may be linked to several marriages at the same time e.g. Abraham could be married with Sarah and Hagar in the same point in time (we characterize this as interpreting the diagram with a *current semantics*). In a second interpretation, Abraham may be linked to several marriages through his entire life, but not necessarily at the same time (we characterize this as interpreting the diagram with a *lifetime semantics*). The different interpretations here are crucial to establishing the intended conceptualization: in the first case, the model admits polygamy explicitly, while the second model does not rule it out.

In current semantics, the UML multiplicity specifies cardinality constraints that should hold for each single point of time. In lifetime semantics, a multiplicity specifies cardinality constraints that should hold considering the set of all possible instants of time. Axiom 1and Axiom 2 formally characterize these two interpretations of the UML multiplicity in the example above.

Axiom 1 Current Multiplicity

$$\forall w \in W, \forall m \in ext(Marriage, w), \#m.person(w) = 2$$

Axiom 2 Lifetime Multiplicity

$$\forall m \in ext(Marriage), \#m.person = 2$$

W is the set of worlds states, the operator # denotes the number of values of a set and the expression *m.person* denotes the persons linked to a specific marriage. Axiom 1 states that for every world, for every marriage that exists at that world, the set of partners (persons) that marriage is linked to, at that world, is equal to 2. Axiom 2

states that for every marriage that will eventually exist in all possible worlds, that marriage will have in his entire existence exactly 2 partners (they may or not be the same). Note the difference of between the two statements. In current semantics a class has an extension at a particular world *w,* e.g. ext(Marriage, *w*), whilst in lifetime semantics a class has the same extension at all possible worlds, e.g. ext(Marriage).

Consider the UML class diagram depicted in Figure 4. The figure shows that a person may have several marriages where a marriage can be a current or a former marriage (i.e. a past marriage). Here we can notice informally, looking at the classes' names that lifetime semantics does not seem to apply. If there is a particular entity called "Current" Marriage, this means that the class diagram is specified with a current semantics, i.e. for a single point in time, where "current" is the on-going (actual) marriage of a person and "Former" Marriage the set of a person's past marriages (which is reified in order to keep track of the past).

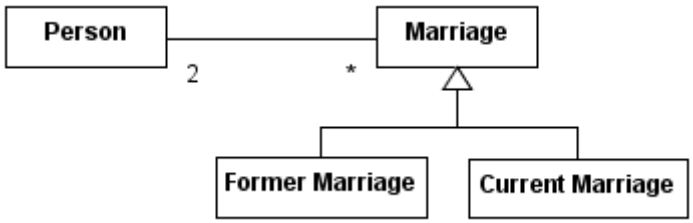

Figure 4 UML Class Diagram: Former and Current Marriages

What contributes for such misinterpretation of UML multiplicity/class semantics is that the majority of conceptual models about different subject domains have the same multiplicities in both current and lifetime semantics, and this hinders the correct interpretation of the model. For example, it is reasonable for a given domain that a person has several marriages both at the same time and through his/her life (polygamy). However, despite this fact, UML does assume (as it was originally created to assume) a current semantics, intended to represent only the current state of a system (even if the past is reified in order to keep the world's history as part of the current state of the system). Therefore, classes in plain UML has extensions at a particular point in time and multiplicities specify cardinality constraints with current semantics. Figure 5 depicts the implied formal semantics in the previous class dia-

gram of Figure 3, showing that all UML multiplicities are defined by formulae as specified in Axiom 1 and thus classes has extensions functions at a particular world.

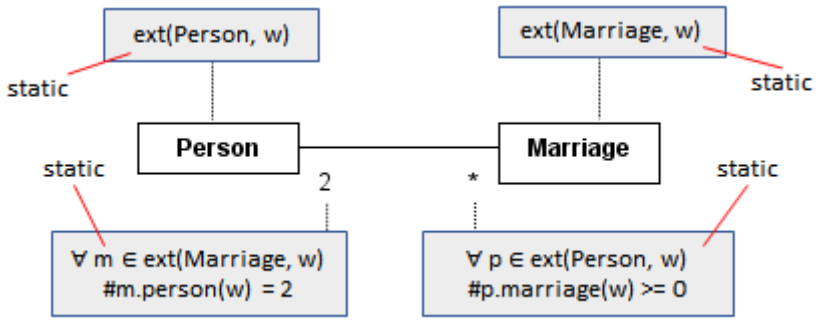

Figure 5 Formal Semantics of a Current UML Class Diagram

In addition to these static aspects, plain UML can represent a single type of *dynamics* called immutability. Immutability is denoted in UML using the simple meta-attribute "readOnly" into the immutable UML association end-point (or attribute), such as exemplified in Figure 6**.**

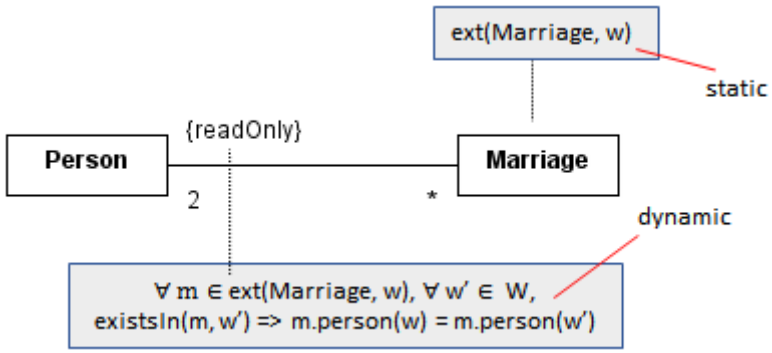

Figure 6 Current UML Class Diagram: Immutability Dynamics

The UML *readOnly* defines that the UML association end-point (or attribute) cannot be updated once assigned. This means that their values cannot change when time evolve [4, p.125, 129]. The readOnly from Marriage to Person means that the participating partners of a marriage cannot change. In other words, if a marriage exists at a point in time *w*, then at every time *w'* that the marriage exists, that marriage will have in *w'* the same partners as in *w*. Figure 6 then depicts the dynamic formal semantics of immutability implied by the UML readOnly feature.

The diagrammatic notation of UML is limited with respect to the static (and dynamic) aspects that it can represent. Therefore, in the following, we start presenting

a constraint-based language (OCL) used to complement UML class diagrams with static aspects in order to increase its (static) expressiveness.

2.4 The Constraint-Based Language (OCL)

The Object Constraint Language (OCL) [5] is a textual, semi-formal constraint representation language adopted by the OMG to express restrictions on MOF-based models e.g. UML. OCL is declarative and based on first-order logic. OCL is used with UML to represent static aspects that cannot be represented using class diagrams. OCL is used to describe and complement UML models respectively with UML (static) class invariants, derivation rules for UML association end-points and UML attributes, pre- and post-conditions for UML operations, definition of UML class operations, among other features [5, p.5, 6].

OCL static invariants are conditions that must be satisfied at any time. In other words, they are conditions that should be respected at every single state of the system. OCL derivation rules in turn express how attributes or association end-points (also called as "properties" using UML terminology) can be inferred from other conceptual model elements. OCL derivations are static since the model properties themselves are defined with a current semantics representing thus a snapshot of the system (as discussed in previous Section 2.3). Since OntoUML disallows UML operations, interfaces and association classes [2], only a subset of OCL is meaningful to OntoUML. So, we focus here on UML class invariants and derivation rules for UML attributes and association end-points [22].

In the following, we briefly explain OCL by means of another example about a domain conceptualization of people and marriages. In this conceptualization, a person can have at most one marriage at a time and a marriage always involves two or more partners. A polygamous marriage is a marriage that also includes more than two partners at a time and a monogamous marriage includes only two partners. Moreover, every marriage has identified if it is under-aged i.e. if it has at least one partner under the age of 18.

Figure 7 depicts this domain modeled with plain UML. It shows a class Marriage partitioned as {Polygamous Marriage, Monogamous Marriage} and having a derived attribute of primitive type Boolean called "isUnderAge" (derived properties in plain UML are often denoted by a slash "/" previous to the property name). Furthermore, a person has an integer attribute denoting a person's age at a time.

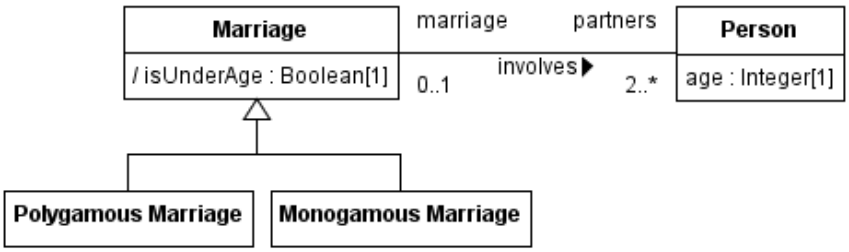

Figure 7 UML Class Diagram: Polygamy and Monogamy

The conceptual model of Figure 7 is not able to define two desired aspects of phenomena. Firstly, a derivation rule stating that the "isUnderAge" attribute of Marriage is derived from the partners of that marriage, checking if there is at least one partner under the age of 18. Secondly, two static class invariants, one stating that the age of a person must always be greater than 0 and that a monogamous marriage must always involve exactly two partners at a time. Listing 1 specifies these two static aspects represented with standard (plain) OCL.

```
context Marriage::isUnderAge: Boolean

derive: self.partners->exists(p | p.age < 18 )

context Person inv: self.age > 0

context _'Monogamous Marriage' inv: self.partners->size() = 2
```

Listing 1 OCL Static Constraints about Polygamy and Monogamy

The OCL keyword "inv" specifies an OCL invariant and the keyword "derive" an OCL derivation rule. In the case of invariant, the keyword "context" specifies the domain concept to which the condition must hold (in UML terms a Class). In the case of derivation, the context specifies the UML attribute or UML association endpoint to be derived and has the format "*Class*::*Attribute* : *Type*", where *Class* is the owner of the *Attribute* and *Type* its respective type.

The keyword "self" represents a specific object, an individual of the context class. In the derivation rule, this means that "self" represents an individual of "Marriage" and in the invariants an individual of "Person" and "Monogamous Marriage" respectively. OCL specifies that class or property names with spaces or accented characters should be preceded by underscore inside of single quotes.

In the derivation rule, the OCL expression "self.partners" returns the set of partners (people) related to a specific marriage called "self" and, in the first invariant rule, the OCL expression "self.age" returns the integer number related to the age of the specific person called "self". In OCL, the dot notation (i.e. ".") is used to navigate through the association end-points or access attributes of the model. OCL also specifies that we can use the name of the class (owner of the attribute or association end-point) with lower case characters in order to navigate through the model, if a name for that property is not given.

The OCL "size" operator returns the number of elements in an OCL collection. An OCL Set is a specific type of an OCL Collection. Other OCL collections include Bags, Ordered Sets and Sequences but Bags are only meaningful to OntoUML in the context of material relationships. The OCL "exists" operator iterates over an OCL collection ensuring that at least one element of the collection (e.g. the set of partners of a specific marriage) satisfy a boolean condition, for instance, checking whether a person's age is fewer than 18. It is important to emphasize that although the derivation rule or invariant is specified for a particular individual (i.e. self), the condition or rule must be true for every single instance of the context class.

We have seen that a conceptual model is then represented by (i) UML class diagrams, with its *static* (and very limited *dynamic*) constraints implied by the diagrammatic notation and by (ii) a set of *static* user-defined constraints specified in plain OCL. In the following we start incorporating additional *dynamics* in plain UML using the ontologically well-founded profile for UML class diagrams.

2.5 The Ontologically Well-Founded UML Profile

Figure 8 depicts a different conceptual model in plain UML about a domain of persons, their stages in life and marriages. It shows that there are people, which can be children, teenagers, adults or elders, and orthogonally, man or woman. In the scope of a marriage, a man can be a husband and a woman can be a wife.

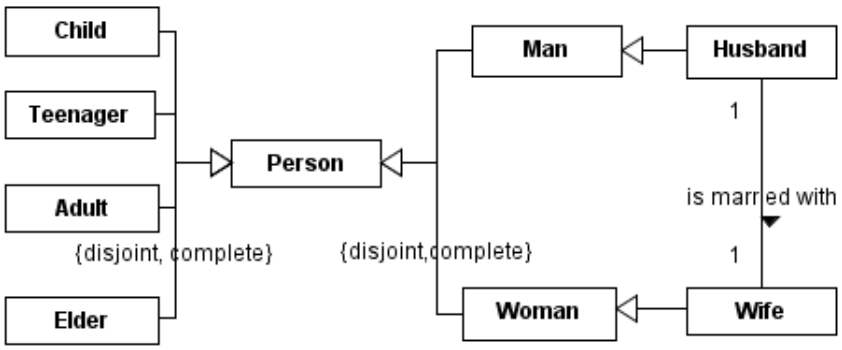

Figure 8 UML Diagram: Marriage, People and Stages of Life

A partition {disjoint, complete} is a UML generalization set with attributes "isDisjoint" and 'isCovering" equal to true. For example, Person is classified by the partition {Man, Woman}. The attribute *isCovering* means that every instance of Person must be either an instance of Man or Woman at a time. The attribute *isDisjoint* states that, there is no person who can be both man and woman, at the same time [2].

UML has been widely used and adopted as a conceptual modeling language [26] but as a conceptual modeling language, UML should be suitable to represent relevant aspects of the phenomena of a domain and be effective and clear in representing such phenomena through the constructs of the language [27]. The class diagram fragment of UML 2.0 was re-designed and evaluated according to the structural layer of the Unified Foundational Ontology (UFO) dubbed UFO-A (which we cover in Appendix A). The result is a well-founded version of UML for ontology-based conceptual modeling dubbed OntoUML.

UFO gives the OntoUML language's constructs. Thus, while in UML the constructs of the language were UML classes and UML relationships, in OntoUML these are refined according to the hierarchy of UFO-A. The allowed OntoUML classes are: Kind, Collective, Quantity, Subkind, Role, Phase, RoleMixin, Category, Mixin,

Relator, Mode, Quality and PhaseMixin. The allowed set of OntoUML relationships are: Material, Formal, Characterization, Derivation, Mediation, ComponentOf, MemberOf, SubQuantityOf, and SubCollectionOf. OntoUML only incorporates constructs with ontological interpretations; consequently, OntoUML disallows UML constructs such as UML Interfaces, Association-Classes and Operations. Lastly, OntoUML defines a number of constraints derived from UFO that restricts the ways the OntoUML classes and relationships can be related in order to produce syntactically valid conceptual models [2, 27].

The model of Figure 9 revisits the model of Figure 8 employing the OntoUML profile. The profile uses class stereotypes to determine which ontological category from the UFO applies to each class [2]. This means that OntoUML can address some of dynamic aspects of this domain that are not addressed in plain UML.

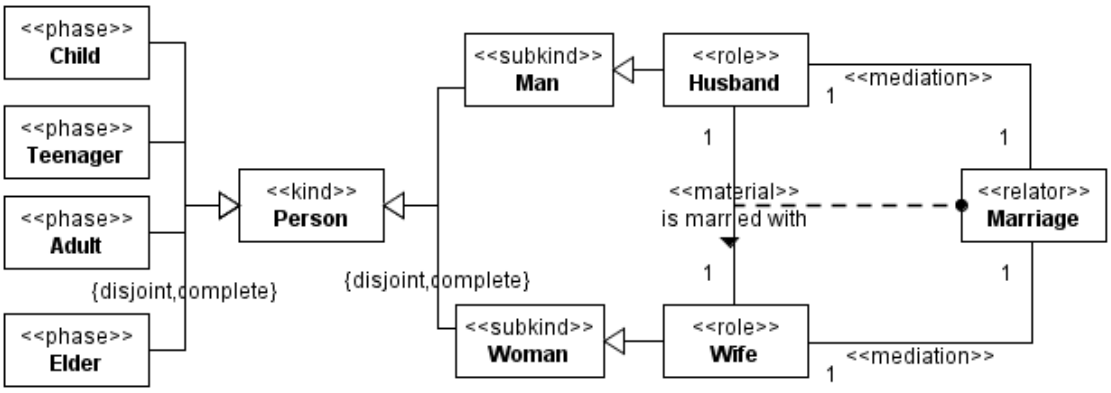

Figure 9 OntoUML Diagram: People, Stages of Life and Marriages

For example, the class Person is stereotyped as «kind», meaning that it applies necessarily to its instances. Thus, a person cannot cease to be a person without ceasing to exist. This modal notion corresponds to what is called Rigidity in UFO. The consequence of rigidity in terms of dynamic aspects is that an instance of a rigid class instantiates this class throughout its life. A kind can be used in a taxonomic structure with rigid subtypes known as subkinds (e.g., Man and Woman).

Other examples of dynamic aspects expressed in Figure 9 include those implied by the use of the stereotypes of the classes Husband, Wife, Child, Teenager, Adult and Elder. Husband and Wife are stereotyped as «role» and Child, Teenager, Adult and Elder as «phase». Roles and phases are anti-rigid concepts (e.g. a wife can cease

to be a wife without ceasing to exist). Anti-Rigidity states that a class C is anti-rigid iff for all its instances, there will be a possible world w in which they exist but do not instantiate C, at w. The difference between roles and phases is that the former defines contingent properties of an instance exhibited in a relational context (e.g. a person is a wife contingently and only in the context of a marriage) while the latter through an intrinsic change of an instance's property (e.g. a child has the intrinsic property of being a child).

The class Marriage is stereotyped as «relator». Relators can be viewed as objectified properties, as entities that "connect" other entities. They are the truthmakers of the so-called «material» relationships. For example, it is the existence of a particular marriage connecting man X and wife Y that makes true the relation "ismarried-with(X, Y)". A derivation relationship on the other hand holds between a relator and a material relationship and exemplifies the truth-maker relation. Relators are rigid concepts and existentially dependent on the instances they connect through «mediation» relationships. A mediation is a type of relationship that defines existential dependence from their source entity, e.g. Marriage, to their target entities, e.g., a Wife and a Husband. This means that a marriage only exists at some point in time, if wife and husband also exist at that point in time. A particular marriage then depends specifically on two "fixed" persons, i.e., the marriage between Bob and Alice cannot ever become the marriage between Bob and Anna. Mediations are thus always defined as readOnly at their target-side by default. From a logical point of view, this dynamic aspect of existential dependence can be viewed as a type of immutability (a marriage never changes their participating wife and husband). Finally, the classes Husband and Wife are related through exactly 1 Marriage, meaning that we represent monogamous heterosexual marriage, i.e., a partner can only be married to one partner at a time.

In the sequel, we simulate the diagram of Figure 9 as a means to demonstrate that even with the additional dynamics introduced in UML by the OntoUML profile, the class diagram of Figure 9 does not represent some important dynamic constraints that serve to rule out inadmissible situations. We generate an instantiation which is possible according to the model. We use the existing ontology-based approach that

is based on Alloy simulation and analysis [22, 23]. In this approach, a temporal interpretation is given to the model suitable for validation purposes in order to show how the entities change from world to world in some sort of story, even though OntoUML is neutral with respect such interpretation. This means that worlds are not anymore equally accessible but are accessible in an ordered manner through a binary accessibility relation called next.

The following figures: Figure 10, Figure 11 and Figure 12, represent three subsequent states of the world. In the first state of the world called *World0* (which simulates a past state of the world), there were two marriages existing in time: a marriage *Property1* between a man *Object1* who was a child and a teenager woman *Object2*, and a marriage *Property0* between a man *Object3* who was an elder and another elder woman *Object0*.

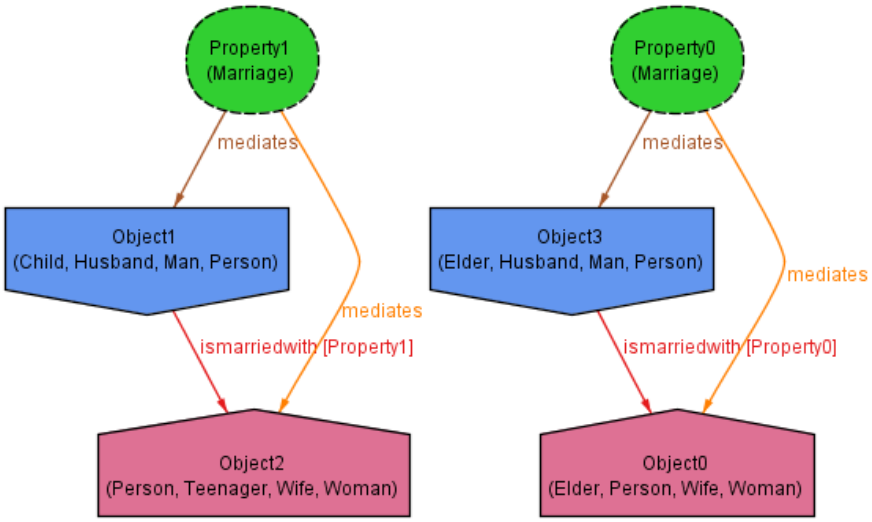

Figure 10 A Past State of the World to the Marriage Example

In a second state of the world (*World2*, simulating a possible present world), *Man1* (i.e. *Object1*) suddenly ceases to exist as an elder. This makes his marriage (*Property1*) with W*oman2* (*Object2*) cease to exist. His wife, *Woman2*, however, which was previously a teenager, continues to exist in time as a child woman. In addition, in this second world, *Man3* continues married with *Woman0* but *Man3* turns from being an elder to an adult while *Woman0* continues to exist as an elder.

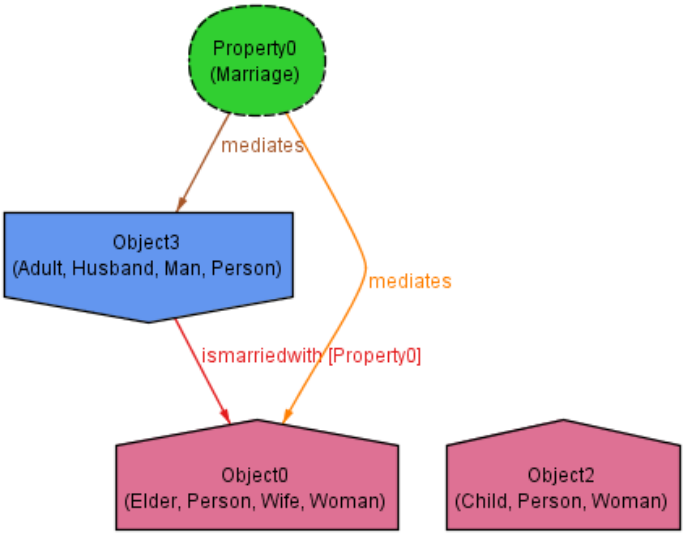

Figure 11 A Present State of the World to the Marriage Example

In the last, and third state of the world (*World3*, simulating a future possible world from the present world), *Woman2* suddenly ceases to exist and *Man1* who was once married with *Woman2* (in the past) and ceased to exist (in the present), now suddenly comes back into existence (in the future) as a child. The marriage *Property0* between *Man3* and *Woman0* continues to exist but *Man3* who was in the past a child and in the present an adult, now (in the future) suddenly turns into a child (his wife *Woman0* continues to be an elder).

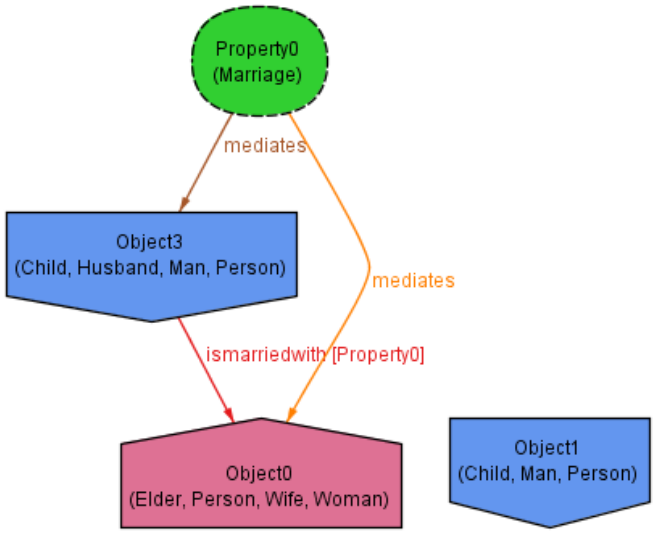

Figure 12 A Future State of the World to the Marriage Example

Note that, while formally correct according to the model in Figure 9, these sequences of world states are clearly inadmissible according to our common sense notions

of this subject domain. For example, in the first state of the world (the past world), only *Man1* is a child. *Man3*, *Woman0* and W*oman2* are not created as child; they were created as elders and teenagers, respectively (Initial Classification Rule). In the second state of the world (the present), *Man3* turn from adult to elder (Final Classification Rule) i.e. *Man3* should, after being an elder, cease to exist of continue to be an elder. In addition, in this present world, *Woman2* turned from being a teenager to a child and this should not happen since once teenager, a person should only be after that an adult or remain a teenager. Lastly, in the third world (the future), we see that *Man1* who once ceased to exist in the present, now in the future exist again (Continuous Existence Rule). This is undesired since a person should not come into existence once it ceased to exist. Also, Man3 and Woman0 always exist from the first to the last state of the world (Transient Existence Rule), which is also undesired since a person should eventually cease to exist at some point.

We see that OntoUML is not expressive enough to represent all relevant aspects of a given domain conceptualization due to its nature as a diagrammatic notation and due to its still limited support for dynamics. Even with the re-design and evaluation of UML according to a foundational ontology such as UFO, the resulting language still lacks the ability to express some important dynamic aspects. That, affects the accuracy of models with respect to a domain conceptualization, and motivates the extension of the language as discussed in this work.

2.6 Final Considerations

In the current semantics, a UML class diagram represents *static* invariants that are implied by the diagrammatic notation (with the exception of the UML "readOnly" feature which implies a type of *dynamic* invariant called immutability). UML can be complemented with arbitrary *static* aspects using OCL textual constraints since UML's diagrammatic notation is limited with respect to the static aspects it can represent. In OntoUML, in addition to the UML's static invariants, *dynamic* ones are included from the ontological distinctions of UFO. The dynamic invariants captured by OntoUML are the formulae implied by the different types of rigidity of classes (rigidity, anti-rigidity and semi-rigidity) and dependences (immutability). However,

OntoUML is still limited with respect to dynamics. With alethic modality we can state that there will be a point in which things can change or that, at any point, things will be the same (this logics is called a logic of possibility and necessity). Note that no temporal interpretation is necessary for that. However, a temporal interpretation is required if we want to state how things should behave as time evolves in some ordered manner. With a temporal interpretation, we will be able to enforce the admissible histories for the entities in the domain, determining how they may behave in time throughout various world states. In order to capture these temporal invariants, we build on UFO's distinctions and augment the profile to support the representation of more accurate models. This leads to a simple extension of the OntoUML profile capable of expressing some additional temporal notions.

3 Introducing Temporal Aspects in Conceptual Models

In this chapter, we formally characterize some *temporal* dynamic aspects identified for OntoUML conceptual models and demonstrate by examples how they can be used within OntoUML. We first formally characterize a temporal interpretation for our previous "alethic" model structure in order to address temporal aspects (Section 3.1). We then briefly discuss the default semantics of OntoUML's categories and the need for a specific type of dynamic aspects called continuousness (Section 3.2). We formally present the set of identified dynamic aspects according to the existence of endurants (Section 3.3) and their change in instantiation (Section 3.4). Finally, we demonstrate how these aspects can be included in OntoUML class diagrams in order to be accurate in the representation of a subject domain (Section 3.5).

3.1 Temporal Accessibility Relation

In Section 2.2, we formally defined a semantics for a model structure using the alethic modality, defining a binary accessibility relation R defined in W x W with all worlds being equally accessible. This is the default semantics assumed for structural OntoUML conceptual models, where no temporal modality is assumed. However, if we want to specify the behavior of model entities in time, worlds must be accessible following some order in a structure called *World Ordered Structure* or just *World Structure*. A world structure encompasses a set of worlds ordered together through a temporal accessibility relation, which is a special case of the accessibility relation R. A temporal interpretation will restrict the way R can relate worlds. Let us assume an accessibility relation called *next* as a partial order relation. In other words, *next* is *irreflexive* (a world is not next to itself), *asymmetric* (if world *w* is next to world *w'* then *w'* is not next to *w*) *transitive* (any world next to any world *w''* which is next to world *w*, is also next to *w*), and *acyclic* (a world must not be transitively next to itself). We thus assume our world structure to be a tree wherein no joining branches are allowed.

In the following, we will use the temporal accessibility relation *next* to describe dynamic aspects that were not able to be represented solely with alethic modality, which is implicit in the OntoUML formal semantics.

3.2 UFO Semantics

In UFO, individuals that persist in time are called *endurants*. Endurants are divided into *substantials* and *moments*. Moments can be seen as objectified properties that inhere in other individuals. For example, a headache is an intrinsic property of a person and does not exist by itself. A headache only exists if that particular person also exists. A person is an example of substantial, an individual that has an identity [2], is not existentially dependent on other individuals and as such does not inhere in any individual. Substantials are often called *objects*.

UFO's ontological distinctions such as the different types of rigidities and dependences are silent with respect to whether endurants exist *continuously* in time. By *continuous* we mean the unbroken and constant existence of something over a period of time. For example, UFO is silent with regard to whether a person (an object) can cease to exist and later in time exist again, or whether a person's headache can cease to exist and then exist again. If a headache is created, a new headache should be created and not the same headache as before. It is thus undesired that a person's headache exist intermittently in time, as it is undesired that a person (a physical object) exists intermittently in time. In this sense, both substantials and moments (endurants) should have continuous existence (i.e. single existence) in time.

We formally characterize this dynamic aspect in Axiom 3. The axiom states that an endurant universal (a type) E is continuous if for all its individuals, they exist in all worlds between any two worlds in which they exist. In other words, if the individual exists at any world *w* and *w'* which are transitively next to each other, then that individual should exist in every world between *w* and *w'*.

Axiom 3 Continuousness of Endurants

$$\forall x\, \mathrm{ext}(E), \forall w, w' \in W, \mathrm{next}(w', w) \text{ and } \mathrm{existsIn}(x, w) \text{ and } \mathrm{existsIn}(x, w') \rightarrow$$
$$\forall w'' \in W, \mathrm{next}(w'', w) \text{and } \mathrm{next}(w', w'') \rightarrow \mathrm{existsIn}(x, w'')$$

Continuousness is addressed in [30] as a specific axiom in the formal validation approach with Alloy simulation and analysis. Although OntoUML is silent with respect continuousness, the author incorporated this axiom as part of the temporal interpretation used for validation.

Substance Sortal universals [2] are types that provide a principle of application and identity to its instances. A principle of application supports the judgment of whether the type applies to an individual e.g., whether an individual is recognized as instance of the type Person. A principle of identity supports the judgment of whether two individuals are the same through their identity criteria [2, p.98]. Every substantial individual must be an instance of a Substance Sortal Universal, namely a Kind, Collective and Quantity. Similarly, in order to a moment individual exists, it must instantiate one of the Moment universals namely a Relator, a Mode or a Quality. Therefore, we propose that UFO should by default incorporate this axiom for every Substance Sortal and Moment universal i.e. Endurant universal.

3.3 Durability

In the sequel, assuming that all endurants (substantials and moments) are by default continuous in time, we define an orthogonal type of dynamics denoted *Durability*. Durability refers to *how long* an endurant should exist over a period of time and should not be confused with continuousness. Continuousness refers to *how many* existences an endurant should have in time (i.e. only a single existence is allowed).

3.3.1 Permanence

By *permanent* we mean that an individual cannot be destroyed, or in other words, we call a type G permanent when its individuals, once they exist, always exist, as formalized in Definition 3. The definition states that if an instance of G exists at world *w* then it exists at every subsequent world *w'* from *w*.

Definition 3 Permanence

$$\text{permanent}(G) \stackrel{\text{def}}{=} \forall x \in \text{ext}(G), \forall w \in W, \text{existsIn}(x, w) \rightarrow \forall w' \in W, \text{next}(w', w) \rightarrow \text{existsIn}(x, w')$$

"Celestial Person" could be an example of a permanent person, a person whose identity relates to a spiritual or non-physical realm, not defined by its biological body conditions but by the existence of his/her own soul or spirit, which after conception never ceases to be. This means that a permanent, celestial person endures in all possible worlds after existing at some world. In this conceptualization, death is a change in phase, not the end of existence. Similarly, another example could be "Celestial Marriage" which would in this particular religious conceptualization last forever even in the afterlife of partners.

3.3.2 Transience

By *transient* we mean that an individual will eventually be destroyed, in other words, we call a type G transient when its individuals, once they existing, cease to exist eventually. In other words, there will be a case (some world state after it existed) wherein the individual no longer exists, as formalized in Definition 4.

Definition 4 Transience

$$\text{transient}(G) \stackrel{\text{def}}{=} \forall x \in \text{ext}(G), \forall w \in W,$$
$$\text{existsIn}(x, w) \rightarrow \exists w' \in W, \text{next}(w', w) \rightarrow \text{not existsIn}(x, w')$$

"Biological Person" could exemplify this type of dynamics i.e. a person whose identity relates to him/her existing as a living organism, having an identity defined by his/her biological body conditions. A biological person should, at some point, cease to exist. Another example could be "Civil Marriage" which is a marriage between two people that will come to an end whether (i) with the death of at least one of the partners (which will eventually happen assuming people to be biological beings) or (ii) with the partner's divorce.

3.3.3 Eternity

By *eternal* we mean that an individual exists in all worlds, in all branches of worlds of the structure adopted, in other words, we call a type G eternal when its instances always exist and there is no world wherein an eternal individual does not exist, such as formalized in Definition 5.

Definition 5 Eternity

$$\text{Eternal}(G) \stackrel{\text{def}}{=} \forall x \in \text{ext}(G), \forall w \in W, \text{existsIn}(x, w)$$

"God" could be an example of such dynamic aspect if considered to always have existed and to always exist into the future. "Planet" could also be considered eternal in a conceptualization capturing a relatively short-term human perspective where it would be irrelevant to define an eventual destruction/creation of planets and our universe.

A moment in UFO is divided into intrinsic moments (modes and qualities) and relational moments (such as relators). Intrinsic moments and substantials (kinds, collective and qualities) can be eternal but relational moments cannot because it would be contradictory to the anti-rigidity of role playing. Substantials are by definition externally independent individuals, as they should not depend on any other individual in order to exist. If a marriage is eternal (i.e. always existed and will always exist) so are the two people participating in that marriage eternal as well. The two people are bound to each other as long as the marriage exists, but as the marriage is eternal and always exist at any time, each person is existentially dependent on each other via eternal marriage. If so, they are not be externally independent substantials as stated by the substantial definition, thus the contradiction.

3.4 Classification Dynamics

By *classification* we mean the process or a period in which an individual change from one type or to another. In the sequel, we define two types of classification dynamics for anti-rigid types called *Initial* and *Final* classifications.

3.4.1 Initial Classification

The Initial Classification is a peculiar type of classification rule where there is no antecedent world state, only a subsequent world state, and the condition (the instantiation of an individual at a particular type) should hold at the subsequent world state, which is the first world of an individual's existence. For example, an anti-rigid type T (namely Role, Phase, PhaseMixin, and RoleMixin) is an initial classification

type iff all their individuals are of T in the first world in which they exist, such as formalized in Definition 6. The definition states that there will be a world *w* in which if the individual exists in *w* and does not exist in all previous worlds from *w* then it is of type T at *w*. As all individuals are continuous by default, this formalization assumes that an individual will always have a single existence in time.

Definition 6 Initial Classification

$$\text{Initial}(T) \stackrel{\text{def}}{=} \forall x \in \text{ext}(T), \exists w \in W, \text{existsIn}(x, w) \text{ and } (\forall w' \in W, \text{next}(w, w') \rightarrow \text{not existsIn}(x, w')) \text{ implies } x \in \text{ext}(T, w)$$

"Fetus", "Living" and "Baby" could be examples of initial classifications of people. In the context of a "Human Conception", the first role a person should play in life could be the role of "Fetus". In addition, if {Living, Deceased} and {Baby, Child, Teenager, Adult and Elder} are orthogonal stages of a person's life, the phases Living and Baby are examples of initial phases as a person should always be a living baby in the first world he/she exists (we can have more than one initial role for the same individual at the same time).

3.4.2 Final Classification

The final classification is another special case of a classification rule where there is no subsequent world state, only an antecedent world state, and the condition (instantiation of an individual at a particular type) when reached, must always hold until the subsequent world is reached i.e. until an individual ceases to exist. An anti-rigid type T is a final classification type when all their individuals, when of type T, are always of T unless they cease to exist, as formalized in Definition 7**.** Considering the phases of a person's life, "Elder" could be an example of final phase meaning that a person only ceases to be an elder when he/she ceases to exist. Another example could be "Deceased Person", meaning that a person would not be allowed to resurrect in the domain (once deceased always deceased). This formalization also assumes that all individuals exist continuously i.e. have a single existence in time, as discussed previously that this should be the default semantics of UFO.

Definition 7 Final Classification

$$\text{Final}(T) \stackrel{\text{def}}{=} \forall x \in \text{ext}(T), \forall w \in W,$$
$$x \in \text{ext}(T, w) \rightarrow \forall w' \in W, \text{next}(w', w) \rightarrow \text{not existsIn}(x, w') \text{or } x \in \text{ext}(T, w')$$

3.5 Examples

Given UFO's focus on alethic modality, OntoUML currently lacks expressivity with regard to dynamic aspects. It cannot express (i) continuousness, (ii) durability (permanence, transience and eternity), and classifications (initial and final) invariants. In the next section, we demonstrate how these dynamic aspects can be applied within OntoUML in order to represent as accurately as possible a domain conceptualization. We propose a simple extension to OntoUML in which these dynamic distinctions are represented using UML tagged values. We demonstrate that our OntoUML extension enables structural conceptual models to be aligned with different philosophical views about time and existence. We first start briefly presenting a philosophical theory about time called *Presentism* showing how the dynamic aspects introduced in OntoUML can enable models aligned with this view. We later discuss the *Growing Block Universe* theory, also discussing how the dynamic aspects introduced in OntoUML can enable this other view.

3.5.1 Presentism

Presentism is a philosophical theory of time that states that events and entities that are wholly past or future do not exist in the present. In Presentism, only present things exist [21]. Let us consider that a modeler aligned with presentism produces a conceptual model with all universals marked {transient}. In this view, past things such as ex-marriages, deceased people, ex-husbands, ex-wives, deceased parents, deceased children do not exist in the present, rather, only living people, current marriages, husbands, wives, living parents, living children exist. Figure 13 depicts an example about people, marriages aligned with the presentism view represented with the proposed OntoUML extension. Note the syntax {transient} to denote transient existence of biological persons, monogamous marriages and current parenthoods. We defined our own specific concrete syntax notation for the dynamics introduced in

OntoUML (the tagged value name between curly brackets at the bottom of the class). We are not advocating that this is the best concrete syntax, but that they are required if we want to capture accurately some dynamic invariants about that the portion of that world.

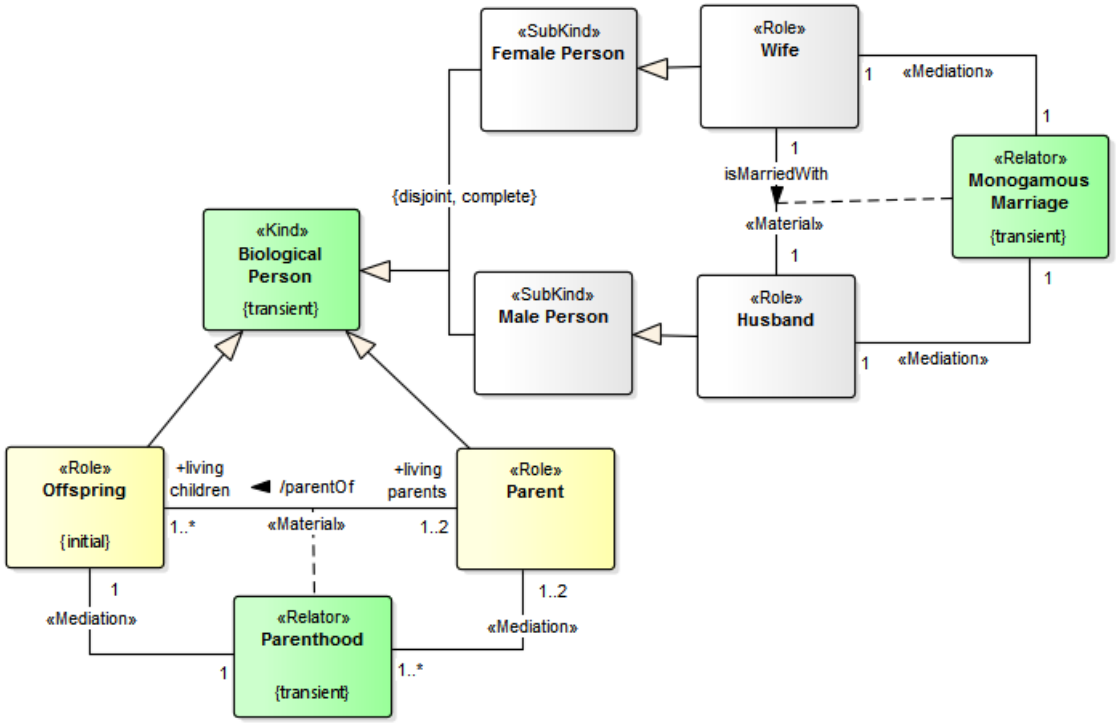

Figure 13 OntoUML Diagram: People and Marriages in Presentism

Each person, marriage and parenthood is transient which means it ceases to exist at some point in time. A person is considered to cease existence when all its functions that sustain a person to exist as a living biological organism cease. We are considering that resurrection is not allowable, i.e., a deceased person (who ceased to exist) cannot come back to existence. Similarly, a marriage that no longer holds ceased to exist and cannot be brought back into existence. A marriage can be seen as a socially or religiously recognized union or legal contract between husband and wife that establishes rights and obligations between them. A male spouse is called husband whereas a female spouse wife. A marriage is defined monogamous i.e., a form of relationship in which a husband/wife can only marry one partner at a time. The material relationship *parentOf*, on the other hand, relate living existing parents to their living existing children. It defines that a person has a set of children and one or two parents. A person is created as a living child (note the syntax {initial}) and a

parenthood between two living parents and their child will hold until the parents or the child cease to exist (the parenthood is existentially dependent on them).

In this "presentist model", we do not define the notions of ex-husbands, ex-wives, or ex-marriages. Further, we cannot refer to my grandfather (which is a father to my father) since he does not exist in the present. We cannot refer to John Lennon's father as both John and his father are deceased in the present. In presentism what exists is what exists in the present. If we wanted to represent an ancestral relationship (to relate my grandfather which is today a wholly past entity to my father which is a living, existing entity) that would be not possible using that "presentist model" solely represented with OntoUML. The ancestral relationship is a type of *Historical* relationship because it depends on entities that are not necessarily in the present but in the past. Historical relationships are temporal relationships and relate entities at all worlds. Such relationship cannot be represented in the style of "presentist model" discussed in this section, as only presently living things are considered to exist.

In order to represent as accurately as possible this particular conceptualization, besides the historical relationship of ancestry, we would also want to represent a fact involving that relationship, for example, an invariant stating that people cannot be descendants/ancestors of themselves, where people here are both people from the present and the past. This type of fact is called *Trans-Temporal Fact* [21] or just *Historical Dependence Fact*. Not only the "presentist model" cannot represent historical relationships but the OntoUML language is not sufficient to express trans-temporal facts due to its nature as a diagrammatic notation. In this sense, we would need an additional language to express both temporal historical relationships and trans-temporal facts.

3.5.2 Growing Block Universe

The Growing Block Universe theory of time (or the growing block view) states that the past and present exist and the future does not exist. By the passage of time more of the world comes into being, therefore the block universe is "growing". The grow-

ing block view is an alternative to both Eternalism (according to which past, present, and future all exist together) and Presentism (according to which only the present exists) [21]. In this view, ex-marriages, deceased people, ex-husband, ex-wives, deceased parents and children, all exist in the present time together with current marriages, living people, husbands, wives, living parents and living children. Figure 14 depicts our running example about people and marriages using our OntoUML extension, now more aligned with the growing block universe view.

In this view, the "universe" is said growing with all entities. It is important to note that the model is still a snapshot of the abstraction of phenomena. The difference is that wholly past entities are now part of what exists together in the present with present entities. In this manner, all entities are permanent; note the syntax {permanent} to denote permanent existence of biological persons, monogamous marriages and parenthoods. People, marriages and parenthoods are permanent in existence because they indeed should never leave existence i.e. the block of universe is said to be always growing. Thus, once they are created, they can never leave existence, but they can assume different classifications with regard to the time of their existence (e.g. living and deceased people, current and ex-marriages; living parents and deceased parents, living children and deceased children). By defining all endurants as permanents we are actually defining that the set of individuals of the model always increase in time (people always increase, marriages always increase and etc.)

In addition, according to this "growing block model", once people are created they must be living people (hence living people are initial classifications) and when they become deceased they cannot be alive again i.e. they must be deceased from this point forward (hence deceased people are final classifications). This same "pattern" is applied to other entities such as current and former marriages which are initial and final classifications of permanent marriages. Once a permanent marriage is an ex-marriage, it will always be an ex-marriage from that point forward and thus once a husband/wife becomes and ex-husband/ex-wife, he/she will always an ex-husband/ex-wife, meaning they are final classifications by implication. The same holds for current husbands/wives which are initial classifications of permanent mar-

riages i.e. a marriage must be created as a current marriage. People are created as living children and once they are parents, they should always be parents.

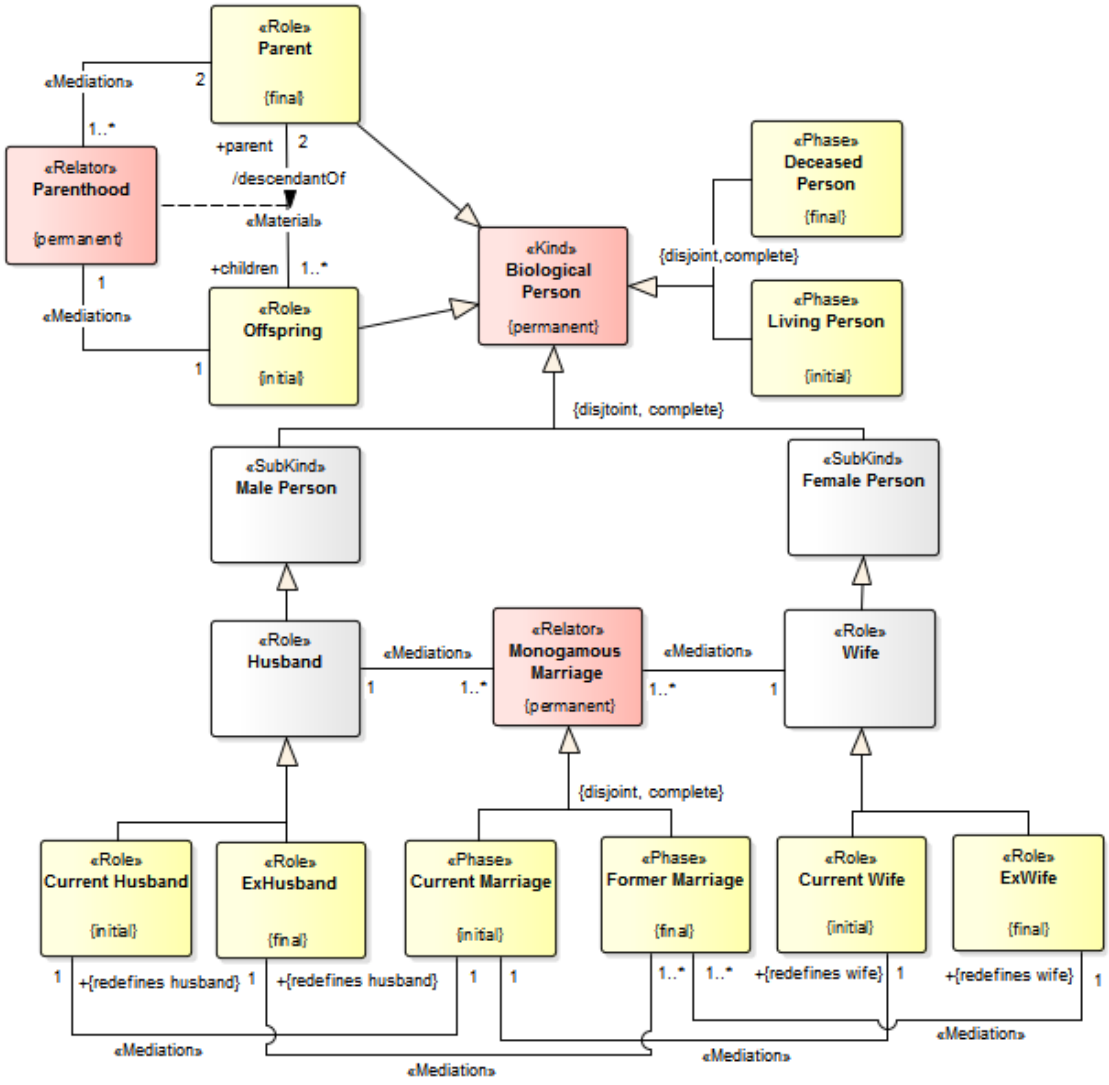

Figure 14 OntoUML Diagram: People and Marriages as Growing Blocks

In this model, the material relationship *descendantOf* relates not only living parents and children (as opposite to the "presentist model"), but also deceased ones. In this manner, the relationship could be called *descendantOf* because it represents our former conceptualization about an ancestry relationship, which relates ancestors with their descendants. In presentism, this relationship was considered historical because it depended on wholly past entities. As in the growing block view past things exist as part of the "block of universe which is always growing", we can represent our ancestry relationship as an (Onto-) UML relationship that by default uses a current semantics (Section 2.3) i.e. it relates concepts at a particular point in time. Not only

we can represent the ancestry relationship, but also our trans-temporal fact stating that people should not be descendants/ancestors of themselves. The trans-temporal fact can be represented as a static fact using (static) OCL as in Listing 2.

```
context _'Biological Person'
inv: self->asSet()->closure(children)->excludes(self)
```

Listing 2 Trans-Temporal Fact in Plain OCL in the Growing Block View

Lastly, derivation by past specialization [20] is a type of dynamic aspects wherein an existing set of entities are derived from the past. For example, for some domain conceptualizations, past marriages are not a present existing entity but ex-husbands and ex-wives are. Therefore, an ex-husband or ex-wife would be a person who was a husband/wife in a marriage that existed in the past, which no longer exists in the present. If the former (ex-) marriage is not a present entity (alignment with presentism for marriages) but the ex-husband and ex-wife are (alignment with the growing block universe for husbands and wives) this would characterize a case of derivation by a past specialization. Therefore, in hybrid models it would be possible to have past specializations whilst in in the growing block view theory they do not make sense as all past entities are indeed part of the present.

3.6 Final Considerations

In this chapter, we defined dynamic aspects that could be added in OntoUML in order to precisely represent some dynamic invariants of conceptualizations. These aspects regard the existence and classification of individuals. We have given a temporal interpretation to the structure of world states in order to precisely define semantics of the tagged values that were added to the profile. The temporal dynamic invariants addressed in this chapter enable the specification of models in different styles i.e. models more aligned with the presentism theory or with the growing block universe theory. We have seen that when we adopt a model aligned with the growing block view, we have means to express trans-temporal facts, since the past is reified and considered current. Therefore, there are still a number of challenges with respect to the representation of dynamic invariants. For example, so far, we just en-

abled the representation of trans-temporal facts in the case where existence is reified, in models aligned with the growing block universe theory while derivations by past specialization cannot be represented in models aligned with either of the theories. We address some of the remaining challenges in the next chapter, in which we propose that additional dynamic invariants should be specified using a complementary textual constraint language, more specifically a temporal extension of OCL. This extension is able to represent trans-temporal facts, past specializations and richer dynamic invariants about a subject domain without requiring the adoption of a growing block approach.

4 OCL Temporal Extension for Ontology-Driven Conceptual Modeling

In this chapter, we present our temporal extension of OCL. A standard OCL invariant is a static condition that should hold for each single state of the model's instances. Consequently, the so-called "context" of a standard invariant is a single state, and no notion of "state" is manipulated in standard OCL invariants. In order to enable the manipulation of states and consequently the representation of dynamic aspects, we reify the notion of "world states" (or simply "worlds") (Section 4.2). Reification gives the ability of referencing, quantifying and qualifying over an objectified entity (in this case, "worlds"). We use the "world" as an index to refer to the properties at a particular point in time (Section 4.3). We propose a temporal interpretation for our extension as a branching world structure, which can be used to enable arbitrary reference to worlds and branches (paths of worlds) in temporal constraints (Section 4.4). We adjust few standard OCL predefined operations in order to support world indexing (Section 4.5). We also define a concrete syntax for the definition of historical relationships with our extension using the modeling infrastructure described here to represent the dynamic aspects developed in this research (Section 4.6).

4.1 OCL Extension Approach

In our approach, the modeler produces an OntoUML model enriched with temporal OCL constraints. This enriched OntoUML model is automatically translated into a "world-reified model" in plain UML in order to give context to the temporal OCL constraints attached to it, allowing the temporal constraints to be parsed and syntactical verified against the background model (see Figure 15). This world-reified model in plain UML is enriched with constraints in plain OCL to ensure that the OntoUML model semantics is preserved.

Temporal OCL constraints are just constraints written in adjusted OCL in the context of the background world-reified model. Only few adjustments in plain (standard) OCL are required in order for OCL to behave as a temporal language.

We employ these adjustments: (i) adding built-in operations for temporal navigations (Section 4.3), (ii) adding built-in operations for manipulation of world states and world paths (Section 4.4) and (iii) revisiting a few plain OCL built-in operations regarding objects and the *allInstances* operation (Section 4.5).

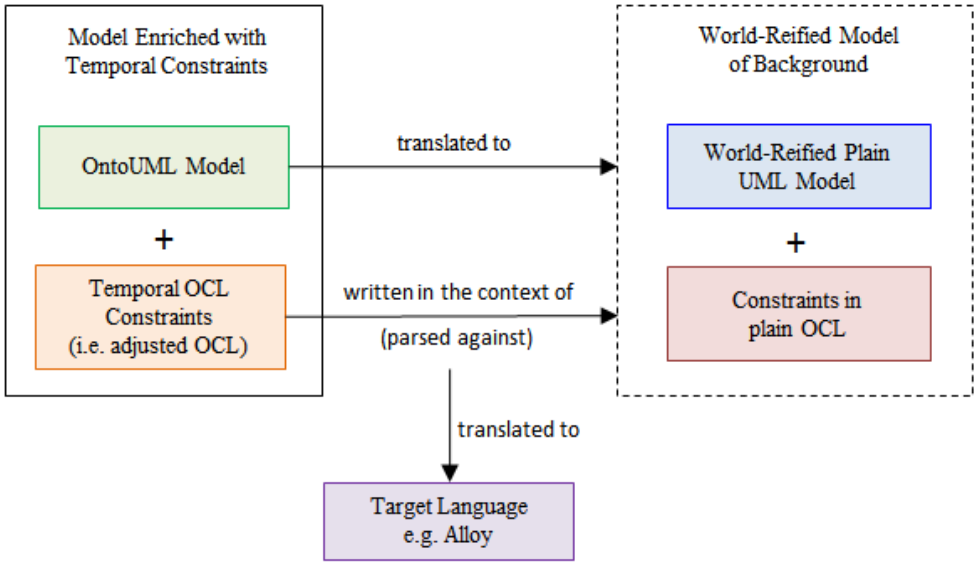

Figure 15 Extension Approach: World-Reified Model of Background

These Temporal OCL constraints are parsed (syntactic verified) against the world-reified model of background in order to be transformed to a target language such as Alloy [19]. The modeler expresses a conceptual model in OntoUML and Temporal OCL and is shielded completely from the underlying support, which ultimately generates an Alloy model for simulation and validation of constraints.

In the sequel, we present first a fragment of the world-reified model in plain UML enriched with constraints in plain OCL, used as background to give context and analyze syntactically Temporal OCL constraints. To do that, we use our previous running example of people and marriages.

4.2 World-Reified Model of Background

The idea behind world states reification is to treat the world states (or "worlds") as entities, thus, we introduce the class "World" in this reification step. The OntoUML model example about people, their stages in life and marriages, of previous Section 2.4 in Figure 9 is depicted again in the sequel in Figure 16.

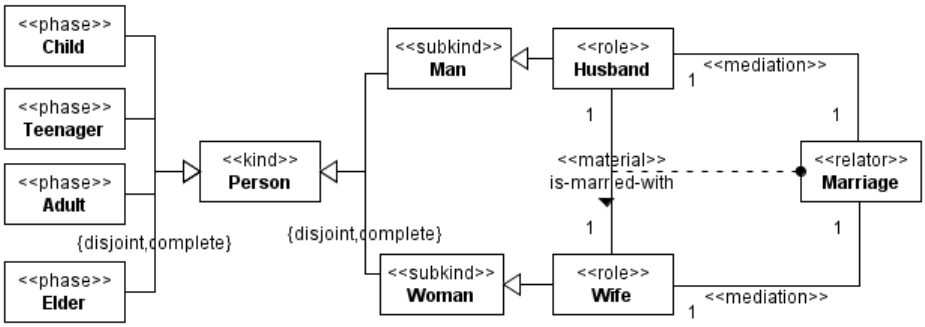

Figure 16 OntoUML Example: People, Stages in Life and Marriages

This OntoUML model is (automatically) translated into a world-reified plain UML model enriched with plain OCL constraints. Figure 17 depicts a fragment of the resulting reified model. UML is employed here as a temporal model and therefore UML classes represent instances existing at all possible states of the world. Every OntoUML class (e.g. the kind Person, the relator Marriage) now specialize *Endurant*, in order to support the *existsIn* relation, which holds for the worlds in which an endurant exists. All OntoUML classes are then indexed in time through this relation of existence.

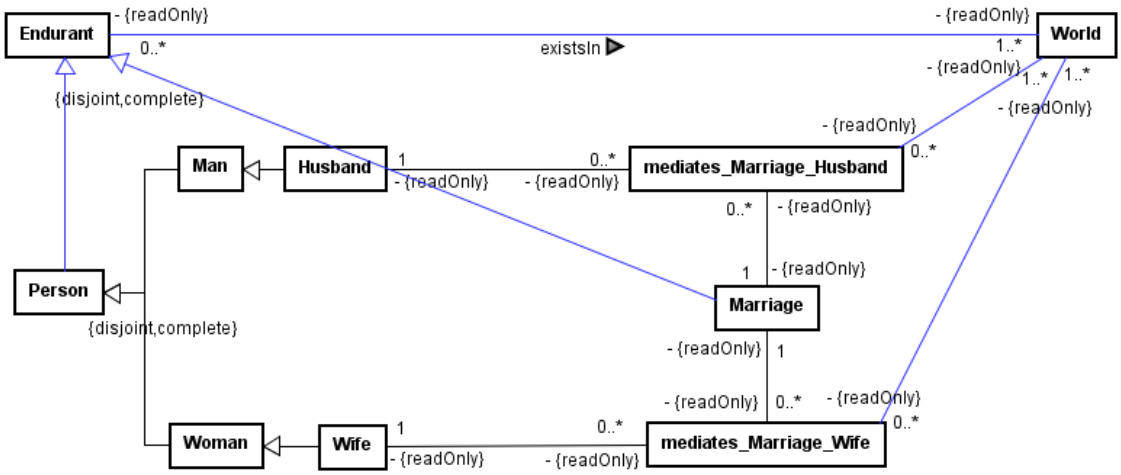

Figure 17 A Fragment of World-Reified Plain UML Model of Background

In order to capture the dynamics of relationships in this reified model, we basically represented all OntoUML relationships as a pair between two types indexed by worlds. For instance, the UML class *mediates_Marriage_Wife* represents the pair (the OntoUML mediation relationship) between the types Marriage and Wife, and that relationship (the pair) existsIn a non-empty set of Worlds. In addition, in each

world, there may be a set of relationships (pairs) relating marriage and wife. The original OntoUML relationship is reified (translated) into a UML class, with three UML binary relationships and additional plain OCL constraints to maintain the semantics of the original OntoUML relationship. Finally, note that all UML relationships in this reified model are *readOnly* by default since time was reified and each property change is now characterized by a change in the world states.

Further, all original OntoUML multiplicities define current multiplicity constraints (i.e. they restrict how many instances an instance may be linked to at a single world state). We chose to represent these actual multiplicity constraints of OntoUML in our world-reified plain UML model as additional constraints using plain OCL. We did that because only the OntoUML lower (current) cardinality can be represented using a temporal UML multiplicity (e.g. a wife has exactly one marriage at a time, which means that she has also at least one marriage in her lifetime). We would not be able to represent the OntoUML upper multiplicities using plain UML in our temporal model. For this reason, we chose to represent all original OntoUML multiplicities as additional plain OCL constraints and therefore, in our world-reified model, the temporal multiplicities from UML classes Wife and Marriage to the UML reified mediation (*mediates_Marriage_Wife)* is defined as just 0..*.

This world-reified model of background is used as the context for navigations and context declarations in our Temporal OCL extension. All the additional constraints used to maintain the semantics of this world-reified UML model according to our OntoUML model example are presented properly in Appendix B. There, we define additional constraints in plain OCL over the world-reified model such as to ensure current multiplicities, existence cycles, immutability of relata, the collection types for relationships, and the different types of rigidity, which cannot be represented using plain UML. Note that these constraints and the world-reified model are all generated automatically as a result of our translation and are not exposed to the modeler.

4.3 Built-In Temporal Navigations

Navigation in OCL means the task of navigating from a specific object to its set of associated objects via an association end in order to produce an OCL expression. The result of a navigation expression can comprise any number of associated objects (including zero, i.e., no object). In the case in which the association end is defined with cardinality value greater than 1, the navigation expression always results in a collection of elements. If no object is associated then the result is an empty collection. If the association end defines a cardinality value of at most one and no object is associated, the result of the navigation expression is an *undefined* value [5].

Navigations expressions are represented in OCL by the DOT notation (i.e. "."). For example, in our OntoUML model example, consider the variable *sarah* an object of the role Wife and the predicate *marriage* the mediation end-point from role Wife to relator Marriage. The navigation *sarah.marriage* results in the collection of marriages that *sarah* participates with at a single world state. Single navigations in mediations and all OntoUML relationship always result in a Set type (collection of elements without ordering or repetition). The only exception regards OntoUML material relationships, which can result in collections of the type Bag because they are derived relationships from their respective relator and relator's tying mediations.

In our reification approach, all original OntoUML relationships were reified into a respective UML class with three UML binary associations acting as a world-indexed pair linking domain and range classes and the world class. For example, the UML class *mediates_Marriage_Wife* acts as the ternary relationship linking (via association end) the UML classes "Marriage", "Wife' and "World", respectively. OCL navigations on ternary relationships can proceed in three stages: (i) navigating from the ternary relationship to each class it relates, (ii) from each related class to the ternary relationship itself, and (iii) navigating from a first related class to a second related class but filtering the result with respect to the third related class. Only (iii) is allowable in our temporal OCL extension and restricted to the case wherein we filter the result of navigation with respect to a single world state. For example, we can navi-

gate from "Marriage" to "Wife" filtering the result with respect to a world *w*, or vice-versa, we can navigate from "Wife" to "Marriage" filtering the result w.r.t. *w*.

Listing 3 defines these two temporal navigations using plain OCL in the world-reified model of background. The idea is that these temporal navigations should be part of the world-reified model and provided to the modeler as built-in operations, because the generated UML class acting as the reified relationship is hidden from the modeler and is only used at background for syntactical verification of the Temporal OCL expressions. Thus, for each OntoUML relationship there are always two world-indexed navigations for it available to the modeler.

In the listing, the first built-in operation *Wife::marriage(w)* is defined as a navigation from Wife to Marriage filtered by a specific world state. It returns all marriages of a wife at world *w*. The second built-in operation *Marriage::wife(w)* is defined as a navigation from Marriage to Wife, returning the wife related to a specific marriage at world *w*. These world-indexed navigations are available to the modeler in order to refer to the relation in a particular state. This is the implicit default in plain OCL as the expressions are always evaluated in the context of a single world state, we just made explicit that world.

```
context Wife def: marriage(w: World): Set(Marriage) =
    self.mediates_Marriage_Wife->select(m | m.world=w)->collect(marriage)->asSet()
context Marriage def: wife(w: World): Set(Wife) =
    self.mediates_Marriage_Wife->select(m | m.world=w)->collect(wife)->asSet()
```

Listing 3 Definition of Built-In World Indexed Navigations

In addition to these world indexed built-in navigations, we also enabled temporal navigations without a world parameter, which returns all instances linked to a particular association end considering all possible worlds. For example, if *sarah* is a wife, then the temporal OCL expression *self.marriage()* (or alternatively just *self.marriage*) returns all marriages of that wife in her entire life. We describe these two additional built-in operations for temporal navigations in Listing 4 using Plain OCL. For each OntoUML relationship, these other two built-in navigations at all worlds are availa-

ble to the modeler. It is important to emphasize that as a marriage has always the same wife related to it (existential dependence), both navigations *m.wife()* and *m.wife(w)* result in the same set of wives (just one).

```
context Wife def: marriage():Set (Marriage) =
    self.mediates_Marriage_Wife->collect(marriage)->asSet()
context Marriage def: wife(): Set(Wife) =
    self.mediates_Marriage_Wife-> collect(wife)->asSet()
```

Listing 4 Definition of Built-In Temporal Navigations at all Worlds

4.4 Built-In World Structure and Operations

An ordered structure of world states models how the subject domain behaves in time. We adopt a structure of possible worlds inspired in the Kripke structures of modal logic semantics [42]; more specifically, we assume the branching structure previously defined in [30] as part of the temporal interpretation of validation with Alloy simulation. We represented this world structure in UML as depicted in Figure 18. This structure of worlds is a built-in part of every world-reified UML model, dictating how worlds are accessible from each other and specifying a number of pre-defined temporal operations for Worlds and Paths.

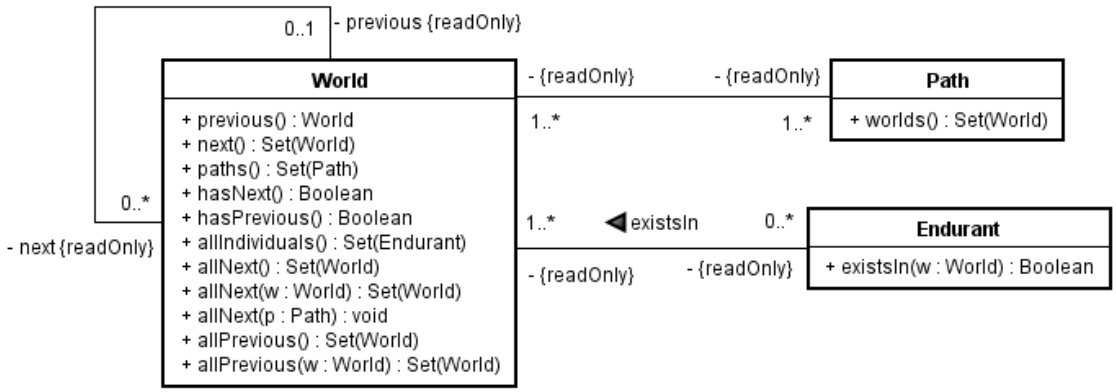

Figure 18 World Structure Fragment of the World-Reified Model

In this temporal interpretation, each world has a set of (immediate) next worlds and at most one (immediate) previous world (it is a tree, with branches towards the future, capturing that the future may unfold in different ways). For each world state,

there is only one sequence of worlds to a future state of the world (meaning that branches do not join again). We express these additional constraints at the World structure (which is part of the world-reified model of background) using plain OCL as described in Listing 5.

```
context World inv no_cycle: self->asSet()->closure(next)->excludes(self)

context Path inv no_parallel_structure: Path.allInstances()->forAll(p |

        self.world->intersection(p.world)->notEmpty())
```

Listing 5 General Constraints of the World Structure in Background

Furthermore, in our world structure, a history (i.e., a path) is comprised by a non-empty set of worlds while a world must be included in at least one history. Differently from Benevides's structure [30], we have reified the notion of paths. Since Path is also an entity as World, several additional constraints are needed in order to enforce the semantics of histories (paths). We describe these additional constraints in Listing 6 using plain OCL.

```
context Path

inv one_terminal_world: self.world->one(w | w.next->isEmpty())

inv one_initial_world: self.world->one(w | w.previous.oclIsUndefined())

inv no_two_paths_with_same_end: Path.allInstances()->forAll(p | p<>self implies

    p.world->select(w |w.next->isEmpty()) <>

    self.world->select(w |w.next->isEmpty()))

inv worlds_of_a_path_derived:

    let t: Set(World) = self.world->select(w| w.next->isEmpty())

    in (self.world-t) = t->closure(previous)

inv every_end_in_one_path:

    let ts: Set(World) = World.allInstances()->select(w |w.next->isEmpty())

    in ts->forAll(t | Path.allInstances()->one(p | p.world->includes(t)))
```

Listing 6 Path Constraints of the World Structure in Background

These constraints specify that a history must contain exactly one initial and one terminal world, no two histories should have the same terminal world and every termi-

nal world must be in exactly one history. Additionally, the worlds contained in a history should be derived from all previous worlds of that history's terminal world. These constraints are added to every world-reified model of background.

Finally, path reification is useful to represent constraints usually expressed in the CTL tense logic, quantifying not only over states but also over paths of states, both universally and existentially. Quantifying universally and existentially over paths is an important feature to some dynamic properties of systems. We validated this world structure using the lightweight formal method of validation based on Alloy simulation and analysis [19], as a means to check the correct semantics of the reified histories (paths) that we introduced in the world structure. A possible simulation for our world structure is depicted below in Figure 19. Note the circles, which are world states and boxes, which are paths. Further, the "next" accessibility relation between worlds, and the world states ordered in the form of a branching structure.

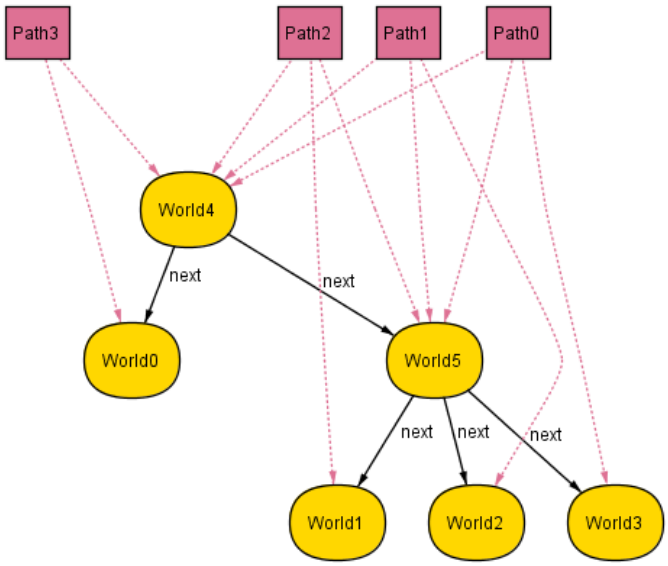

Figure 19 Simulation of the World Structure in Background

In addition to the built-in world structure in the world-reified model of background we define several built-in World and Path operations (which we have shown in the UML model of Figure 18). These operations enable the manipulation of worlds and paths which are necessary if we want to represent the behavior of model entities. These operations are built-in, available to the modeler, and part of every world-

reified model. We define these built-in operations using Plain OCL (in particular, body conditions) as described in Listing 7.

```
context World::next():Set(World) body: self.next
context World::previous():World body: self.previous
context World::paths():Set(Path) body: self.path
context Path::worlds():Set(World) body: self.world
context World::allEndurants():Set(Endurant) body: self.endurant
context World::hasNext():Boolean body: not self.next->isEmpty()
context World::hasPrevious():Boolean body: not self.previous.oclIsUndefined()
context Endurant::existsIn(w: World):Boolean body: w.endurant->includes(self)
context World::allNext():Set(World) body: self->asSet()->closure(next)->asSet()
context World::allNext(w: World):Set(World)
body: if self.allNext()->includes(w) then w.allPrevious() - self.allPrevious() -
      self->asSet() else Set{} endif
context World::allNext(p: Path):Set(World)
body: self->asSet()->closure(next)->asSet()->select(w | w.paths()->includes(p))
context World::allPrevious():Set(World)
body: self->asSet()->closure(previous)->asSet()
context World::allPrevious(w: World):Set(World)
body: if self.allPrevious()->includes(w) then self.allPrevious()-w.allPrevious()-
      w->asSet() else Set{} endif
```

Listing 7 Definition of World and Path Built-In Operations

The operations *next* and *previous* return an immediate next world and immediate previous world from a particular world. The operations *hasNext* and *hasPrevious* checks whether a world has an immediate previous or immediate next world. The operation *allEndurants* returns all existing endurants at a specific world. The operation *existsIn* checks the existence of an endurant at a specific world. The operation *allNext* returns all subsequent worlds of a particular world. This operation in particular has two variants *allNext(w)*, which returns all subsequent worlds until a particular world *w* is reached (not including *w*) and *allNext(p)*, which returns all subsequent worlds from a world, contained in a given path *p*. Analogously, *allPrevious* returns all prece-

dent worlds of a particular world. Finally, *worlds* returns all worlds of a path, and *paths* returns all paths in which the world *w* is contained.

4.5 Revision of Plain OCL Built-In Operations

In addition to the built-in temporal navigations, built-in world structure and the set of world and path built-in operations, we revisit some plain OCL operations (and define new one inspired by plain OCL) in order for OCL to behave as a temporal constraint language. The *oclIsNew* operation is only allowed in post-conditions [5, p.154]. As our subset of OCL does not consider pre-/post- conditions (OntoUML disallows operations) *oclIsNew* is not supported. Instead, we define two temporal operations (inspired by the *oclIsNew* operation) to check an endurant's creation and deletion at a world. The operation *oclIsCreated(w)* checks if an endurant exists in a world *w* but does not exist in its immediate previous world, checking if the endurant was created at *w*. The operation *oclIsDeleted(w)* on the other hand checks if an endurant does not exist in *w* but does exist in its immediate previous world, checking if the endurant was deleted in *w*. We specify these two endurant's built-in operations in Listing 8 using plain OCL on the world-reified model of background. These are operations on Endurants since existence is a characteristic of domain entities that persist in time.

```
context Endurant
def: oclIsCreated(w: World) : Boolean = if(not w.previous.oclIsUndefined() and not
    self.existsIn(w.previous) and self.existsIn(w)) then true else false endif
def: oclIsDeleted(w: World) : Boolean = if(not w.previous.oclIsUndefined() and
    self.existsIn(w.previous) and not self.existsIn(w)) then true else false endif
```

Listing 8 Definition of oclIsCreated and oclIsDeleted Built-In Operations

We define two additional built-in operations for Endurants regarding the classification of an endurant at a world. The operation *oclBecomes(C, w)* checks whether an endurant is classified as class C at *w* but is not classified as C in *w*'s immediate previous world. The operation *oclCeasesToBe(C, w)* on the other hand checks whether an endurant ceases to be classified as C at *w*. That is, the endurant does not instantiate

C in *w*'s immediate previous world, but instantiate C at *w*. These were inspired by the *oclIsKindOf* operations in plain OCL.

There are only few adjustments to some built-in plain OCL object and classifier operations that need to be established due to our world reification approach. Type conformance operations must explicit the point in time in which the types are checked. Since plain OCL does not natively support world states, we include a world state parameter in *oclIsKindOf(T, w)*, *oclIsTypeOf(T, w)*, *oclAsType(T, w)* and *oclType(w)*. The plain OCL *allInstances* operation is still allowed and it returns the extension of a class at all possible worlds i.e. the set of all instances of a class independent of their actual existence in a particular point in time. In this manner, Temporal OCL expressions such as *World.allInstances()*, *Path.allInstances()*, or *Endurant.allInstances()* are all valid constructions in our extension. They return respectively, the set of all possible worlds, the set of all histories and the set of all endurants at all worlds. Additionally, we assume a temporal UML static operation *allInstances(w)* for every UML domain class. The operation *allInstances(w)* returns all instances of a class at world *w*. Temporal OCL expressions such as *World.allInstances(w)* or *Path.allInstances(w)* are invalid constructions since worlds were reified and neither worlds nor paths exist within worlds..

Lastly, all other plain OCL built-in operations which are considered in accordance to their meaningfulness to OntoUML, remain the same with regard to our temporal extension, e.g., OCL collection operations, primitive value (e.g. integers, booleans, and strings) operations, OCL iterators. These are, by nature, mathematic and logic operations, which do not require states reification to work appropriately.

4.6 Modeler's View

In this section, we represent the dynamic aspects set our as requirements in Section 1.3 and in the running example initially presented in Section 2.4, thereby showing how the approach satisfies the requirements. We thus demonstrates that all the dynamic aspects previously set out as UML tagged values in the proposed OntoUML extension of chapter 3 can be expanded into Temporal OCL constraints (as the classification rules and existence of endurants). In addition, we demonstrate that

also past specializations and trans-temporal facts can now be expressed even if a "presentist model" is adopted.

4.6.1 Classification dynamics

Listing 9 exemplifies the Initial classification rule (formalized in Section 3.4.1) in our OCL temporal extension. It states that there will be a world in which if the person exists in a world but does not exist in all previous worlds from that world, then that person is a Child. The keyword *temp* defines a temporal invariant. The temporal *context* defines a class extension at all worlds e.g. all instances that at some point will instantiate the class Person. The condition must hold for each of these instances.

```
context Person
temp initialChild: World.allInstances()->exists(w | self.exists(w) and
     w.allPrevious()->forAll(p| not self.existsIn(p)) implies
     self.oclIsKindOf(Child, w))
```

Listing 9 Initial Classification Rule in Temporal OCL

Listing 10 exemplifies the Final classification rule (formalized in Section 3.4.2) in Temporal OCL. The first temporal OCL invariant states that for every person, for every world, if that instance is an Elder at that world, then for every world after that, if the instance exists, then it instantiates Elder or does to exist. In other words, there is no other allowed classification for it before ceasing to exist.

```
context Person
temp finalElder: World.allInstances()->forAll(w |
     self.oclIsKindOf(Elder, w) implies w.allNext()->forAll(n |
     not self.existsIn(n) or self.oclIsKindOf(Elder, n)))
```

Listing 10 Final Classification Rule in Temporal OCL

Initial and final classifications are a special case of a classification rule; they do not have, respectively, an antecedent and a subsequent world state. In a general type of classification both world states are specified and an instance classified as A1 at the antecedent world state can transition into one or more types S1+... +SN at the sub-

sequent world state. This means that an instance of type A1 can only be of one of those types while existing or remain being A1. Listing 11 exemplifies this more general classification rule in temporal OCL stating that a teenager (A1 = Teenager) can only transition to Adult (S1 = Adult) or continue to be a teenager.

```
context Teenager
temp teenagerToAdult: World.allInstances()->forAll(w |
     self.oclIsKindOf(Teenager, w) implies w.allNext()->forAll(n |
     self.existsIn(n) implies self.oclIsKindOf(Teenager, n) or
     self.oclIsKindOf(Adult, n)))
```

Listing 11 General Classification Rule in Temporal OCL

Using general, initial and final classification rules, we can specify accurately the phase transitions of a subject domain. For example, in our running example of the domain of people, their stages in life and marriages, a person must be created as a baby, then as a baby he/she can only be transitioned to teenager, then from teenager to adult, then from adult to elder, and finally, as an elder he/she might cease to exist. This will define a sequence of admissible phase instantiation. In fact, using this general classification rule we can specify any phases transitions and not only sequential ones. A general transition defines that a type can transit not only to one, but to any number of other types. If the modeler does not specify any classification constraint for phases, the model is assumed to allow any transition between any of the phases e.g. a person that was deceased now comes back alive, an adult that become a child again, and etc.

4.6.2 Existence

Listing 12 exemplifies the existence rules (formalized in Section 3.3.1, Section 3.3.2 and Section 3.3.3) using Temporal OCL. It specifies that a biological person is a transient entity, a celestial marriage is permanent and a planet is eternal. The temporal OCL invariant "transient" states that for every person that exists, there will be at least one world after that in which that person ceases to exist. The second temporal OCL invariant called "permanent" states that for every marriage that exists, it

exist at all possible worlds after that. Note that by doing this, if a marriage exists permanently, also does husband and wife. Therefore, by implication, the roles Husband and Wife are final classifications of a person and both married persons are permanent in existence by implication. If these two invariants (*transientPerson* and *permanentMarriage*) were represented together in a conceptualization in which biological people are married forever, then an inconsistency would be introduced in the model. We can check/validate that using our support for Alloy simulation (which we develop in next chapter). Lastly, the third temporal OCL invariant called "eternal" states that all instances of Planet exist in all possible worlds.

```
context _'Biological Person'
temp transientPerson: World.allInstances()->forAll(w | self.existsIn(w) implies
    w.allNext()->exists(n | not self.existsIn(n)))
context _'Celestial Marriage'
temp permanentMarriage: World.allInstances()->forAll(w | self.existsIn(w) implies
     w.allNext()->forAll(n | self.existsIn(n)))
context Planet
temp eternalPlanet: World.allInstances()->forAll(w | self.existsIn(w))
```

Listing 12 Existence Rules in Temporal OCL

For the sake of completeness, we also specify the Continuous Existence rule (as formalized in Section 3.2) in Temporal OCL as described in Listing 13. It states that a person has a continuous existence in time (i.e. a person is not allowed to be re-created). That temporal OCL invariant states that all instances of Person exist in all worlds between any two worlds in which they exist.

```
context Person
temp continuousPerson: World.allInstances()->forAll(w,w2 |
     w.allNext()->includes(w2) and self.existsIn(w) and self.existsIn(w2)
    implies w.allNext(w2)->forAll(b | self.existsIn(b)))
```

Listing 13 Continuous Existence Rule in Temporal OCL

4.6.3 Past Specializations

Listing 14 exemplifies a case where ex-husbands and ex-wives are required in the model as cases of a Derivation by Past Specialization [20] (as discussed in previous Section 3.5.2). The temporal invariant states that for every wife at *w*, for every subsequent world to *w,* if she exists at that world but her marriage does not exist at that world, then she is an ex-wife at that world.

```
context _'Person'
temp past_spec: World.allInstances()->forAll(w | self.oclIsKindOf(Wife, w) implies
    w.allNext()->forAll(n | not self.oclAsType(Wife, w).marriage(w).existsIn(n)
    and self.existsIn(n) implies self.oclIsKindOf(ExWife, n)))
```

Listing 14 Past Specialization Rule in Temporal OCL

4.6.4 Historical Relationships

Historical relationships are a type of dependence between present and past entities. As structural conceptual models (e.g. OntoUML, UML) represent a snapshot of a conceptualization of a subject domain, they only define relationships between present entities, unless past entities also exist in the model, for instance, a model aligned with the Growing Block Universe Theory as discussed in Section 3.5.2. Listing 15 specifies historical relationships in our Temporal OCL. We defined our own concrete syntax as plain OCL does not support this definition.

```
context Person::descendantOf : Person
temp: { children: Person[0..*]; parents: Person[2]; }
```

Listing 15 Ancestry Historical Relationship in Temporal OCL

The listing describes a relationship *descendantOf* between people at all worlds. The context is defined in the form *Source*::*relationship*:*Target*. The brackets states that there will be two ends (i.e. domain and range) for that relationship, each one is named and a multiplicity is given. The *descendantOf* relationship has an end-point called *children* that relates a person to its set of children (any number of children) and end-point called *parents* that relates a person to its parents (exactly two). The multiplicities fol-

low the UML standard such as "0..*", "1" ,"0..1", "1..*", and etc. We demonstrate the semantics of this historical relationship in Figure 20**.** The figure depicts a graphical simulation using Alloy according to the technique developed in next chapter. It shows an object 0 (a person) which exists at world state 0 and two persons, 1 and 2, existing at the next world 1. Assuming that world 1 is the present state of the world, person 0, which is a past object, is the parent of both persons 1 and 2. Note that we only defined the historical relationship with no constraint imposed on it. Hence, person 0 is allowed to be a descendant of his/herself.

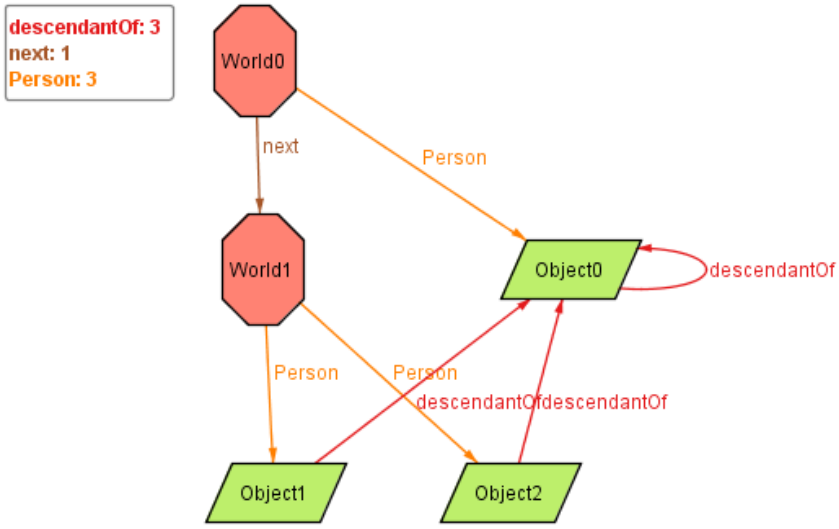

Figure 20 Simulation of Historical Relationship (with no constraint imposed)

4.6.5 Trans-Temporal Facts

Listing 16 specifies a trans-temporal fact [21] stating that a person cannot be the descendant/ancestor of itself. A trans-temporal fact (or just historical dependence fact) is a type of constraint that involves historical relationships. For example, the *descendantOf* relationship that was previously created using our temporal OCL.

```
context Person temp: self->asSet()->closure(parents)->excludes(self)
```

Listing 16 Trans-Temporal Fact in Temporal OCL

Note that the temporal navigation in historical relationships is not defined at a particular world since they are not defined within time; they relate class extensions at all worlds. Further, the same keyword *temp* is used for trans-temporal facts since this is a normal temporal constraint; the difference is that it uses a historical navigation

returning the set of elements independent of a world parameter. Note also the simplicity of this rule and similarity with plain OCL as it is almost identical to the representation of trans-temporal facts using OCL in a model aligned with the Growing Block Universe Theory (Section 3.5.1); instead of keyword *inv* we just use *temp*.

4.7 Final Considerations

In this chapter, we have defined a temporal extension for OCL to cope with dynamic aspects in ontologically well-founded conceptual models with OntoUML. The temporal OCL extension developed requires only few adjustments to standard OCL; in particular, to few OCL type conformance operations and a classifier operation. Our temporal OCL is expressive to enable user-defined dynamics aspects to be incorporated into conceptual models. The main core of plain OCL is maintained the same i.e. OCL iterators, OCL collections, and OCL primitive types. In addition, we defined temporal built-in objet operations such as *oclIsCreated*, *oclIsDeleted*, *oclBecomes*, and *oclCeasesToBe*, respectively. We included a set of built-in operations for worlds, paths and endurants, explaining temporal built-in navigations and historical relationships. We adopted a temporal interpretation based on Kripke structure of possible worlds, which was already addressed in [30] as a temporal approach for model simulation with the Alloy lightweight formal method of validation. Our temporal interpretation is a tree, with branches of worlds towards the future, which do not join together, capturing that the future may unfold in different ways. We have shown that all requirements can be expressed with this extension of OCL, from the trans-temporal facts to derivation by past specializations and all the dynamics elicited as requirements and represented as tagged values from our proposed OntoUML extension. However, there are still a number of challenges regarding the understanding of the implications these temporal constraints impose on structural conceptual models, for instance, validating whether the model becomes inconsistent and whether it reflects one's domain conceptualization. This challenge is addressed in the next chapter, in which we present an approach to validate OntoUML models enriched with Temporal OCL constraints (via visual simulation and model checking).

5 Validating Ontologically Well-Founded Models Enriched with Dynamics

In this chapter, we present a technique based on Alloy simulation and analysis used to validate structural conceptual models written with OntoUML enriched with dynamic OCL constraints. By validation, we mean the task in which we judge whether the conceptual model represents the intended conceptualization. In this technique, we compare if the world states that are admissible by the conceptual model are in pace with the world situations admissible in the domain conceptualization. In particular we first present the validation approach (Section 5.1). We then describe a fragment of the translation from OntoUML class diagrams to Alloy (Section 5.2) that is relevant to the understanding of the mappings from static OCL operators (Section 5.3) and from our temporal OCL extension to Alloy (Section 5.4). We illustrate the results by executing a simulation for our running example of people, their stages in life and marriages enriched with dynamics written in Temporal OCL (Section 5.5).

5.1 Validation Extension Approach

The previous existing approach to support the validation of OntoUML conceptual models complemented with static OCL constraints [22, 23] was defined by a semantic preserving transformation from OntoUML class diagrams [23] and plain OCL constraints [22] into Alloy. The resulting Alloy specification is fed into the Alloy Analyzer tool to generate and visually confront the stakeholder with possible instances of the model. The Alloy instances and relations displayed by the Alloy Analyzer tool represent the classes and relationships of the OntoUML model. We extend this approach with the support for dynamic OCL constraints written in our Temporal OCL extension, as illustrated in Figure 21. The OntoUML diagram is translated into what we called here *Alloy Structure* meaning that it generates an Alloy structure served as a basis for other constraint translations. Plain OCL constraints are in turn translated as Alloy statements, which need to the added to the Alloy structure, while our extension of the current approach will define a semantic pre-

serving mapping from Temporal OCL constraints into Alloy statements, adding them into the Alloy structure. The mappings must be in pace with the mappings from OntoUML of [23] and plain ("static") OCL of [22], as Temporal OCL encompasses all static OCL operators as part of the language and is written (at the modeler's view) in the context of the OntoUML model.

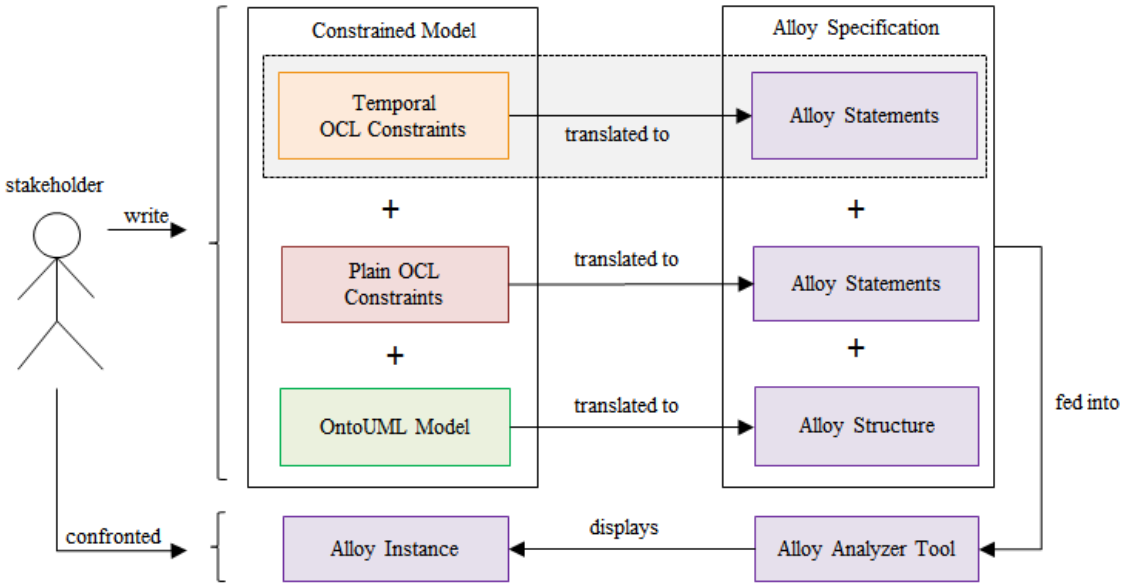

Figure 21 Temporal Extension of the Alloy Simulation Approach

These mappings to Alloy will enable the model (enriched with constraints) to be visually simulated and checked against the stakeholder's conceptualization. Before introducing these mappings, we refer to Appendix C to a briefly introduction about the Alloy language and analysis for the reader unfamiliar with them. The reader familiar with Alloy may skip that introduction and follow the next section to all of our mappings to Alloy.

5.2 Translation of OntoUML Class Diagrams

In the sequel we explain a fragment of the translation from OntoUML class diagrams to Alloy as developed by *Sales* [23]. In particular, we demonstrate (i) the skeleton Alloy code generated which forms what we called previously as the Alloy structure and serves as a basis to the class diagram translation and the other constraint translations, and (ii) the translation of model classes and relationships to Alloy with regard their existence within world states.

5.2.1 Skeleton Alloy Code

The skeleton code is specified in Listing 17. In the first line, the skeleton defines the name of the produced Alloy specification as the name of the respective class diagram being translated. In the second line, the skeleton imports the world structure [30] as an Alloy library alongside with common Alloy libraries such as "util/relation", "util/boolean" and "util/ternary" to deal respectively with Alloy boolean types, and common operations of Alloy binary and ternary relations. The skeleton then imports a pre-defined module for ontological properties defining UFO's distinctions such as rigidity, anti-rigidity, and immutability (dependences) in Alloy.

```
module running_example
open world_structure[World]
open util/relation
open util/ternary
open util/boolean
open ontological_properties[World]
sig Object{}
sig Property {}
sig DataType {}
abstract sig World { } {}
run {}
```

Listing 17 Skeleton Alloy Code

We can view the concept of Alloy signatures as "classes" and atoms as "instances". The "Object" signature then defines all atoms that will represent substantials (commonly referred as just objects), the 'Property' signature all moments (moments are commonly referred as just objectified properties) and the "DataType" signature all the data-type instances from the model. Lastly, the abstract signature called World represents world states whilst the command called "run" tells the analyzer to find a possible instantiation logically valid according to the specification. This skeleton is also used as a basis to the translation of OntoUML class diagrams because each class and relationship at the diagram is generated into an Alloy code that is introduced inside this skeleton [23].

5.2.2 Classes as Alloy Binary Relations

For example, Listing 18 specifies a fragment of the mapping from OntoUML classes of our running example of Figure 9 into Alloy. Each class is translated as an Alloy binary relation between signatures World and existing Objects/Properties. Since Marriage is a moment type (i.e. relator) it is translated as a relation between signatures World and Property. The "exists" relation relates signature "World" and the union of signatures "Object" and "Property". In other words, it relates world states with the endurants existent in each world i.e. substantials and moments. The Alloy operator ":>" filters the range of the relation "exists" w.r.t. specific signatures e.g. (Object + Property). The Alloy keyword "some" defines that the set of all endurants' existent in a world is a non-empty set whilst the keyword "set" that any number of atoms is allowed in the set.

```
abstract sig World {
   exists: some Object+Property,
   Adult: set exists:>Object,
   Husband: set exists:>Object,
   Marriage: set exists:>Property,
   Person: set exists:>Object,
   ...
}
```

Listing 18 Model Classes as Alloy Binary Relations

5.2.3 Relationships as Alloy Ternary and 4-ary Relations

With regard to the relationships, Listing 19 specifies a fragment of the mappings from OntoUML relationships of the previous running model example of Figure 9 to Alloy. Each relationship is translated as an Alloy ternary or 4-ary relation (except for OntoUML derivations) [23]. Material relationships are 4-ary tuples between a world, a relator, and the domain and range of the relationship e.g. the relationship "is married with" between Husband and Wife is derived from the relator Marriage and exist at a particular world state. All other relationships are mapped as ternary Alloy relations e.g. the mediation between Marriage and Husband, the mediation between Marriage and Wife. The Alloy operator "->" is the cartesian product between two sets and the keyword "one" specifies that a set must have exactly one element.

Additionally, each relationship end-point is translated as an Alloy function that receives a world state and the type of the end-point, and returns the end-point opposite type. For instance, the end-point "wife" from class Marriage to Wife receives a world and a marriage and returns a set of wives related to that marriage, as described in Listing 19. The Alloy expression "w.Mediation1" returns all the mediations between marriages and husbands at *w*. The expression "World.Mediation1" returns all mediation relations between marriages and husbands at all possible worlds. Finally, the OntoUML derivation relationship is translated as an Alloy fact stating that the material relationship is derived from the relator's relata and their tying mediations.

```
abstract sig World {
   exists: some Object+Property,
   ismarriedwith: set Husband -> Marriage -> Wife,
   Mediation1: set Marriage one -> one Husband,
   Mediation2: set Marriage one -> one Wife
}
fact derivation_relationship {
   all w: World, x: w.Husband, y: w.Wife, r: w.Marriage |  x->r->y in
   w.ismarriedwith iff x in r.(w.Mediation1) and y in r.(w.Mediation2)
}
fun wife [x: World.Marriage, w: World] : set World.Wife {
    x.(w.Mediation2)
}
fun marriage [x: World.Wife, w: World] : set World.Marriage {
    (w.Mediation2).x
}
```

Listing 19 Model Relationships as Alloy Ternary and 4-ary Relations

5.3 Translation of Plain OCL Operators

With an understanding that classes and relationships from the OntoUML class diagram are translated as Alloy binary, ternary and 4-ary relations, all indexed by a world state, we now present the mappings from static OCL operators into Alloy. Most static (plain) OCL operations and expressions do not require a world parameter in order to function since these are by nature mathematic operations e.g. OCL iterators, OCL primitive value operations, OCL collection operations.

5.3.1 Primitive Values

Alloy natively supports only the Integer and Boolean OCL primitive types. The supported OCL Boolean operations are *and*, *or*, *implies*, *not* and *xor*. They are directly represented in Alloy (with the same concrete syntax), with exception of *xor* which is not natively supported in Alloy but can be implemented through other boolean Alloy operators [22, Table 4]. The supported OCL Integer operations are the comparison operations <, >, <=, and >=, and are directly represented in Alloy with that same concrete syntax. Only some arithmetic operations are supported such as + (sum), - (subtraction), * (multiplication), *div*, *floor*, *round*, *max*, *min* and *abs*. They are represented in Alloy respectively as the Alloy predicates *plus*, *minus, mul*, *div*, *max*, *min* and *abs*. The latter three are not supported natively in Alloy but can be implemented through other Alloy logic operators [22, Table 4] whilst OCL *floor* and *round* are directly mapped to their source value since Alloy only support integers. Finally, the bit width for integers in Alloy is by default 7, which means that integer values range from -63 to 64.

5.3.2 Sets

Alloy supports all the OCL Set operations since it is a set-based language, as shown in Table 1. The symbol [[ ]] denotes a function that receives OCL concrete syntax and returns Alloy textual code. Given the set-based nature of Alloy, the following mappings are straightforward. The operation *size* is represented with the # *(*cardinality) Alloy operator, the operation *isEmpty* and *notEmpty* with the Alloy keywords *no* and *some,* respectively. The operation *includes*, *excludes* and *includesAll* with the Alloy set operators *in* and *not in*. The operation *excludesAll* with the Alloy operators #, & (intersection) and = (equality). The operations *union*, *intersection*, *difference* (i.e. "-"), *including*, *excluding*, and *symmetricDifference* with the Alloy Set operators - (difference), + (union) and & (intersection). The operation *asSet* and *flatten* are directly represented by their source object. Finally, *product* is represented by the Alloy cartesian product and *sum* by the respective Alloy *sum* operator.

Table 1 Translation of Plain OCL Set Operations

OCL Set Operation	Alloy expression
size()	# [[self]]
includes(obj: T)	[[obj]] in [[self]]
includesAll(s: Set(T))	[[s]] in [[self]]
excludes(obj: T)	[[obj]] not in [[self]]
excludesAll(s: Set(T))	# ([[s]] & [[self]]) = 0
isEmpty()	no [[self]]
notEmpty	some [[self]]
union(s: Set(T))	[[self]] + [[s]]
intersection(s: Set(T))	[[self]] & [[s]]
- (s: Set(T))	[[self]] - [[s]]
including(obj: T)	[[self]] + [[obj]]
excluding(obj: T)	[[self]] – [[obj]]
symmetricDifference(s: Set(T))	([[self]] + [[s]]) – ([[self]] & [[s]])
asSet()	[[self]]
product(s: Set(T2))	[[self]] → [[s]]
sum()	sum [[self]]
flatten()	[[self]]

5.3.3 Iterators

Table 2 shows the mappings from OCL iterators into Alloy. The word *col* represents OCL expressions that result in collections and the character *v* variables. These mappings are not straightforward as the Set mappings presented previously. OCL iterators are represented in Alloy as quantified formulae and comprehension sets. The iterator *forAll* and *exists* iterators are represented as Alloy formulae quantified universally (keyword *all*) and existentially (keyword *some*). The iterators *select* and *reject* iterators are represented as Alloy comprehension sets (denoted by curly brackets) whilst iterator *one* is also represented as an comprehension set but using operators such as # (cardinality) and = (equality) to state that the resulting set must be equal to 1. The iterator *collect* is represented combining comprehension sets, the keyword *univ*, the dot notation and a logical true Alloy primitive value (expressed in terms of keywords *no none*). The iterator *isUnique* is represented as an Alloy formula universally quantified plus the disjointness keyword *disj*. The iterator *any* is represented by an Alloy comprehension set but with a restriction of usage: the modeler must ensure

that the boolean expression evaluates to true in exactly one element of the source collection. Finally, the iterator *closure* combines Alloy comprehension sets, the transitive closure operator (^) and the Alloy true primitive value, similar to the collect mapping.

Table 2 Translation of Plain OCL Iterators

OCL Iterator	Alloy expression
col->forAll(v1,..,vn | be)	all v1,..,vn: [[col]] | [[be]]
col->exists(v1,..,vn | be)	some v1,..,vn: [[col]] | [[be]]
col->select(v | be)	{ v: [[col]] | [[be]] }
col->reject(v | be)	{ v: [[col]] | not [[be]] }
col->one(v | be)	#{ v: [[col]] | [[be]] } = 1
col->collect(v | expr)	univ.{ v: [[col]], res: [[expr]] | no none}
col->isUnique(v | expr)	all disj v, v': [[col]] | [[expr]](v)!=[[expr]](v')
col->any(v | be)	{ v: [[expr]] | [[be]] }
col->closure(v| expr)	[[col]].^{v: univ, res: [[expr]] | no none}

5.4 Translation of Temporal OCL Constraints

Our Temporal OCL extension includes all plain OCL operations except for type conformances and the *allInstances* operation which needed to be revisited. In this manner, Temporal OCL mappings include all mappings previously discussed from static OCL operators. In this section thus we define all remaining mappings from our Temporal OCL to Alloy such as from (i) OCL dynamic invariants, (ii) adjustments of Plain OCL type conformances and *allInstances* built-in operations, (iii) temporal OCL built-in operations, (iv) temporal OCL built-in navigations, and (v) temporal OCL historical relationships.

5.4.1 Dynamic Invariants as Facts

Table 3 specifies a mapping from an OCL dynamic invariant to Alloy. As [22], a constraint is represented as an Alloy fact and thus all instantiations of the OntoUML model must conform to that constraint.

Table 3 Translation of Temporal OCL Dynamic Invariants

Dynamic OCL Invariant	Alloy Statement
`context Class` `temp invariant_name: OclExpression`	`fact invariant_name {` `all self: World.[[Class]] | [[OclExpression]] }`
`context World/Path` `temp invariant_name: OclExpression`	`fact invariant_name {` `all self: World/Path | [[OclExpression]] }`

However, the temporal OCL *context* defines a class extension at all possible world (assuming that the context is not a World or a Path). Worlds were reified and therefore any reference to the classes of the model are not bound to a specific point in time (in contrast with static Standard OCL) but to all instances at all times.

5.4.2 Adjusted OCL Built-in Operators

Table 4 depicts the translation of our revision of OCL built-in operations of previous Section 4.5 into Alloy. The mappings follow the mapping of standard OCL from [22, Table 5] but now making explicit the world (time) parameter. The *oclIsKindOf* operation is represented as the Alloy subset operator (i.e. *in*), the *oclIsTypeOf* operation into the combination of operators *in*, *and*, # (cardinality), & (intersection), + (union) and = (equality). The *oclAsType* and the *allInstances* operations are mapped as their respective source object/type in Alloy, since Alloy is by default a set-based language.

Table 4 Translation of Temporal OCL Built-in Operations

Adjusted OCL Operation	Alloy Expression
`oclIsKindOf(T, w: World)`	`[[self]] in w.[[T]]`
`oclIsTypeOf(T, w: World)`	`[[self]] in w.[[T]] and # (w.[[T]] & w.[[subT1]] +..+ w.[[subTN]] = 0)`
`oclAsType(T, w: World)`	`[[self]]`
`T.allInstances()`	`[[T]]`
`Class.allInstances(w: World)`	`w.[[Class]]`

Additionally, our temporal extension of OCL defines four object temporal operations. Their mappings are shown below in Table 5.

Table 5 Translation of Temporal OCL Built-In Endurant Operations

OCL Operation	Alloy Expression
oclIsCreated(w: World)	[[self]] in w.exists and [[self]] not in (next.w).exists
oclIsDeleted(w: World)	[[self]] not in w.exists and [[self]] in (next.w).exists
oclBecomes(T, w: World)	[[self]] in w.[[T]] and [[self]] not in (next.w).[[T]]
oclCeasesToBe(T,w: World)	[[self]] not in w.[[T]] and [[self]] in (next.w).[[T]]

All these mappings use a combination of Alloy operators *in*, *not in*, *and* to access, respectively, if the endurant was created (*oclIsCreated*), deleted (*oclIsDeleted*), classified (*oclBecomes*) or ceased to be classified (*oclCeasesToBe*) at a particular world.

5.4.3 Temporal Built-In Operators

The structure of possible worlds adopted in the existing approach of validation with Alloy [23] does not reify the notion of paths that is part of our OCL temporal extension. For this reason, the mappings from our temporal OCL operators to Alloy are not straightforward. We assume that a Path in Temporal OCL is characterized in the existing structure as a terminal world in order to enable the mappings from our OCL's world structure to that existing structure. Listing 20 specifies four Alloy functions that will be used to manipulate paths (i.e. terminal worlds) in the existing world structure of Alloy as if a reified concept of Path existed.

```
fun Path : set World {
    World.next - (World.next & next.World)
}
fun Path [w: World] : set World {
    w.(^next) & Path
}
fun allNext [w1, w2: World] : set World {
    w2 in w1.(^next) implies ((^next).w2 - (^next).w1 - w1) else none
}
fun allPrevious [w1, w2: World] : set World {
    w2 in (^next).w1 implies ((^next).w1 - (^next).w2 - w2) else none
}
```

Listing 20 Alloy Functions to Manage the Path Reification

The Alloy function *Path* returns all possible terminal worlds (i.e. all possible paths of the structure). The second Alloy function *Path* is parameterized with a world state. It returns all paths (histories) in which a world state is at (i.e. all terminal worlds accessible from that world state). The third and fourth Alloy functions are used to facilitate the mappings from operations such as *allNext(w)*, *allPrevious(w)* and *allNext(p)*. The third Alloy function returns all next worlds from a world *w1* until a world *w2* is reached, using an open interval (i.e. neither w*1* nor *w2* are included). Similarly, the fourth Alloy function returns all previous from *w1* until *w2* is reached.

Table 6 then presents the mappings from our set of temporal OCL operators to Alloy. The *next*, *previous* and *allEndurants* are direclty mapped using the Alloy relations of *next*, *previous* and *exists*. The *allNext* and *allPrevious* are mapped to a forward and a backward Alloy transitive closure, respectively, over the Alloy relation of *next*. The *hasNext* and *hasPrevious* are mapped using the Alloy keyword *some* to check if the set of next/previous worlds are empty. The *existsIn* is mapped using the Alloy *in* operator. The operations *paths*, *allNext(w)*, *allPrevious(w)* and *allNext(p)* are mapped using the additional Alloy functions defined previously in Listing 20. Finally, *worlds* gives all worlds of a path (i.e. all previous worlds from terminal world plus the terminal world itself). It uses a backwards Alloy transitive closure over the *next* relation, uniting the result with the world *self*.

Table 6 Translation of Temporal OCL Built-In World Operators

OCL Temporal Operation	Alloy Expression	OCL Temporal Operation	Alloy Expression
`next()`	`[[self]].next`	`allNext(w: World)`	`allNext[[[self]], w]`
`previous()`	`[[self]].previous`	`allPrevious(w: World)`	`allPrevious[[[self]] , w]`
`allNext()`	`[[self]].^next`	`existsIn(w: World)`	`[[self]] in w.exists`
`allPrevious()`	`^next.[[self]]`	`allEndurants()`	`[[self]].exists`
`hasNext()`	`some [[self]].next`	`worlds()`	`^next.[[self]] + [[self]]`
`paths()`	`Path[[[self]]]`	`hasPrevious()`	`some [[self]].previous`
		`allNext(p: Path)`	`allNext[[[self]], p]`

5.4.4 Temporal Built-In Navigations

The existent translation from class diagrams already generates an Alloy function for each navigable end-point of a (static) OntoUML relationship (Section 5.2.3). This Alloy function however is bound to a specific point in time i.e. it receives as parameter, a particular world state in which the navigation must be evaluated. Temporal navigations were properly defined in previous Section 4.3 and may be indexed in time or not. In Listing 21, we define an additional Alloy function to specify a navigation at all possible world, which is not bound to a particular world. We use the running example of Figure 9 about people and marriages to demonstrate these mappings. The listing shows a temporal navigation from a specific marriage to the set of wives of that marriage at all worlds, and a temporal navigation from wife to all the marriages of that wife at all worlds.

```
fun wife [x: World.Marriage] : set World.Wife { x.(World.Mediation2) }

fun marriage [x: World.Wife] : set World.Marriage { (World.Mediation2).x }
```

Listing 21 Alloy Functions for Temporal Navigations at all Worlds

5.4.5 Historical Relationships

Finally, historical relationships are a type of dependence between entities at all worlds. The ancestry historical relationship relates people at all possible worlds, for example, my father, which is a present entity, is a descendant of my grandfather which is a wholly past entity and does not exist in the present. Listing 22 specifies the mapping from the *descendantOf* historical relationship to Alloy. The relationship is mapped as an Alloy binary self-relationship between Objects, plus four additional constraints (i.e. the Alloy fact *historical_descendantOf*) to ensure respectively (i) the correct types at the domain and range of that relationship (i.e. domain and range as class extensions at all worlds), and (ii) cardinality values at each relationship's end-point. Finally, the two end-points are also mapped as Alloy functions ensuring the navigability at all worlds.

```
sig Object {
    descendantOf: set Object
}
fact historical_descendantOf {
    descendantOf.univ in World.Person
    univ.descendantOf in World.Person
    # descendantOf.univ <= 2
    # univ.descendantOf >= 0
}
fun children [src: World.Person] : set World.Person {
    src.descendantOf
}
fun parents [tgt: World.Person] : set World.Person {
    descendantOf.tgt
}
```

Listing 22 Historical Relationships as Alloy Relation, Facts and Functions

5.5 Validating the Example Enriched with Dynamics

In the sequel, we validated our running example about people and marriages of previous Figure 9 with the addition of some of the OCL dynamic invariants as those aforementioned in this work such as the allowable phase transitions of a person's life, the continuous and transient existence of people and marriages, past derivations of ex-husbands/ex-wives, historical relationship of ancestry/descendants and a transtemporal-fact forbidding cycles of ancestry. We used the mappings specified in this chapter to translate the respective dynamic OCL invariants to Alloy.

In Figure 22 we depict a first possible world state (a past world) of that OntoUML class diagram enriched with dynamic invariants. In this world, there existed two marriages, *Property3* between husband *Object2* and wife *Object1,* and *Property4* between husband *Object3* and wife *Object0*. Both marriages *Property3* and *Property4* were established between spouses that were children. In addition to this, the husband in *Property4* was a direct descendant of his wife and of the wife from the other marriage *Property3* and the spouses (husband and wife) in *Property3* were descendants of the wife from marriage *Property4*. This is clearly not our intention for the historical ancestry relationship between people. People that are married with each other cannot be descendants of each other. Note that we have only forbidden cycles in the histor-

ical relationship and therefore this is a possible situation that may validly occur in our domain. Also, we clearly did not state anything about children possibly getting married in our domain.

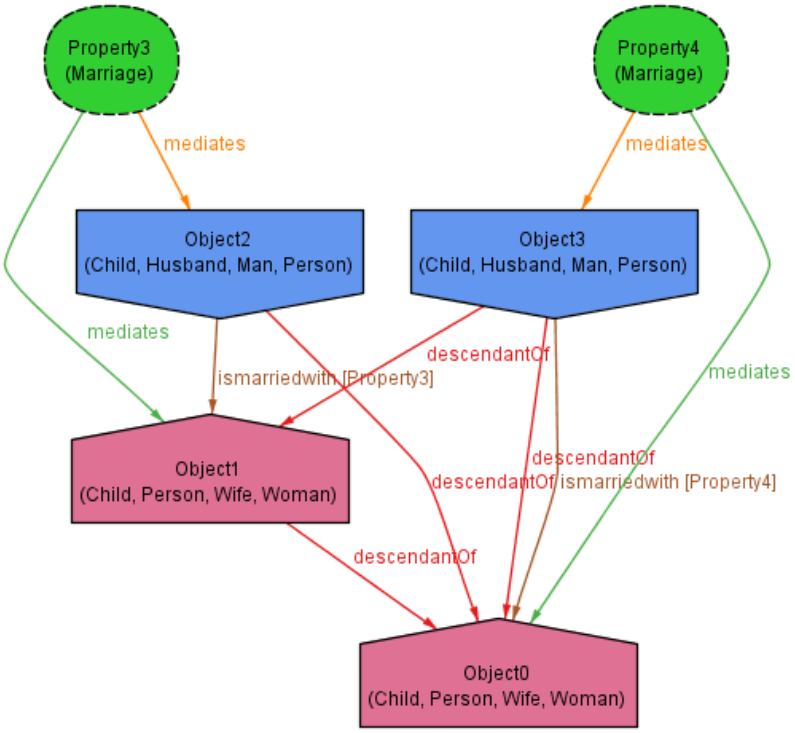

Figure 22 Marriage and Ancestry: A Past World State

In Figure 23, a present world is depicted (a next world from the past world). In this world, man *Object3* remains married with woman *Object0* through marriage *Property4* but the man *Object3* comes from being a child to being a teenager. With regard to the other marriage (*Property3*), it ceases to exist and a new marriage between man *Object2* and woman *Object1* is created. Thus, both man *Object2* and woman *Object1* are still a husband and wife but w.r.t. a different marriage (*Property1*). Additionally, they are classified as ex-husband and ex-wife as their previous marriage (*Property3*) does not exist anymore in the future.

In Figure 24, we depict a possible future world (a next world from the present world). The figure depict that the new marriage *Property1* between *Object2* and *Object1* suddenly ceases to exist as well as both spouses suddenly ceases to exist. Only husband *Object3* and wife *Object0* remain married with each other. However, their previous marriage *Property4* ceased to exist and a new marriage between then are created.

In other words, they simultaneously ceased to be married and got married again. In addition, the man, husband *Object3* comes from being a teenager to an adult whilst his wife remains a child.

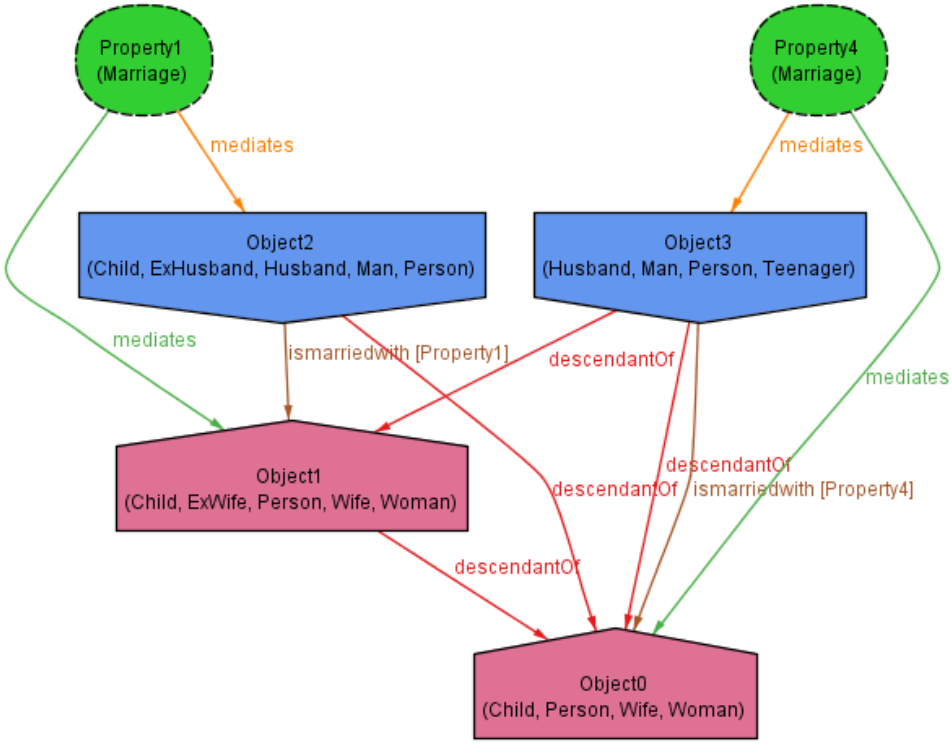

Figure 23 Marriage and Ancestry: A Present World State

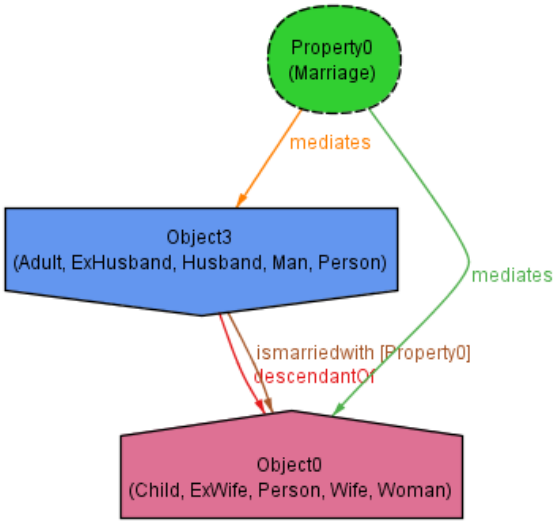

Figure 24 Marriage and Ancestry: A Future World State

The reader can notice that the former dynamic invariants such as classifications, existences, past derivations, and trans-temporal facts are all respected in this graphical

simulation which is generated by executing the Alloy specification resultant from our translations to Alloy. For example, we can note that a person is always created in the child phase, there are no cycles in the ancestry relationship and all the phase transitions are respected alongside the transient existence of the individuals. In order to generate cases wherein adult, teenagers and elders exist in time, marring and ceasing to be married with each other, it is required to increase the scope of the Alloy simulation, for instance, generating more than four states of the world, or a specific number of existing men, women, husbands, wives, and so on and so forth. Due to the lack of space and for the sake of brevity, we only executed the simulation with 4 worlds. However, the scope configuration in Alloy can be fully customized setting the Alloy run command as usually defined in Alloy [19]. Finally, it is important to mention that OCL dynamic invariants can be used not only to avoid a forbidding situation from occurring (translating them to Alloy facts) but to check if a particular desired dynamic property is already captured by the model. In other words, we follow the same approach as [22] translating OCL dynamics invariants as (i) Alloy facts, (ii) Alloy predicates (for running simulations) or (iii) Alloy assertions for model checking. We omit them here due to their straightforwardness.

5.6 Final Considerations

In this chapter, we have presented a validation approach based on automatic generation of instances of the model. We then defined a translation from Temporal OCL to Alloy building up from an OntoUML and standard static OCL translation to Alloy. Using a semantics-preserving mapping to Alloy the stakeholder can graphically visualize the possible world situations according to the model enriched with dynamics and check if indeed the conceptual model represents truthfully the subject domain. In the next chapter, we discuss the implementation of our temporal OCL extension and of the extension of the validation approach with Alloy in a tool to aid modelers to create and validate structural OntoUML conceptual models enriched with dynamics in Temporal OCL.

6 Implementation

In this chapter, we explain the implementation of this work. In particular, we first present the existing plain OCL infrastructure developed for the OntoUML structural conceptual modeling language (Section 6.1). We then present and discuss how we extend it with a support for our Temporal OCL extension (Section 6.2) pointing some important issues about the development of some of the components of the temporal infrastructure (Section 6.3, Section 6.4 and Section 6.5). Finally, we show how the entire temporal tooling is incorporated and available to the modelers in the OLED tool (Section 6.6).

6.1 Plain OCL Infrastructure for OntoUML

Figure 25 depicts the current OCL infrastructure for OntoUML [22] which is defined by a textual plain OCL editor, a plain OCL parser and an *Alloy Translator*. These three components work together in order to provide edition, syntax verification, and validation (via visual simulation and model checking) for plain OCL constraints in the context of OntoUML models. These plain OCL components are part of an OntoUML modeling tool called OLED (OntoUML Lightweight Editor) [34], a model-based environment to build, validate and implement OntoUML models.

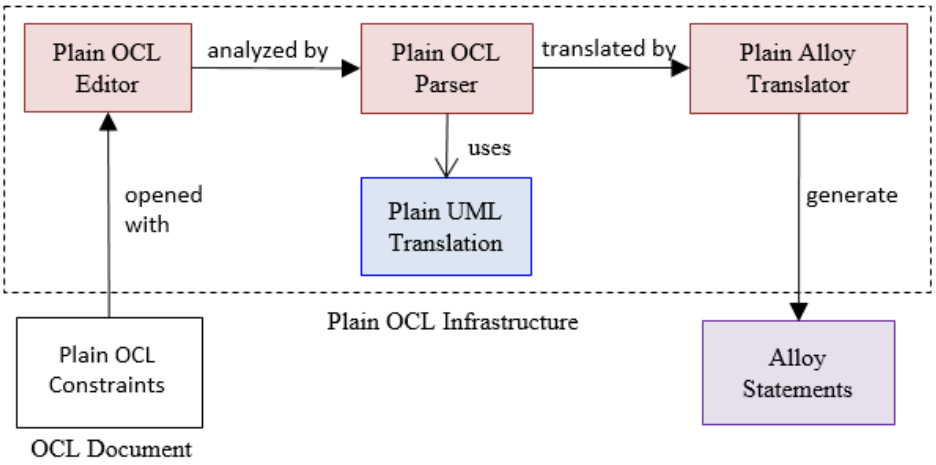

Figure 25 Plain OCL Infrastructure for OntoUML

Plain OCL Editor

The current plain OCL editor supports features such as syntax highlight, code-completion and theme customization which were implemented using three open-source Java components (projects) called *RSyntaxTextArea*[2], *AutoComplete*[3] and *TokenMaker*. These projects practically define (i) a custom Swing text area that can customize a language's syntax highlight and vocabulary through Flex configuration files; (ii) a custom Swing code-completion component that can customize a language's completions using a library of custom completion providers; and (iii) a simple software interface to help generating a .flex and .java file for the custom Swing text area highlighting the language. This plain OCL editor enable the edition of plain OCL constraints using a textual editor.

Plain OCL Parser

The current OCL parser is that of Eclipse Foundation and verifies syntactically plain OCL textual constraints according to OMG's specification. The implementation of OCL used is termed "Classic OCL" by Eclipse. It binds OCL textual constraints with UML/Ecore models, meaning that Eclipse's OCL parser tries to match all classes and navigations present in OCL expressions with the respective UML (or Ecore) model elements. If a particular type or navigation is found in an OCL expression but is not found in the UML (or Ecore) model a parser exception is thrown. Not only the parser checks the constraints with the model being enriched but especially, it checks the plain OCL textual constraints against the standard concrete syntax defined by the OMG. In order to enable the syntactic verification and analysis of these constraints with the OntoUML model, the current infrastructure defines a translation from plain OntoUML models to plain UML models (this is necessary since the current OntoUML infrastructure is not implemented as a UML profile i.e. an extension of UML, but as an independent Ecore metamodel [35]). We called this translation *Plain UML Translation*. With OntoUML models represented in terms of plain

[2] http://bobbylight.github.io/RSyntaxTextArea/

[3] https://github.com/bobbylight/autocomplete

UML, all OCL textual constraints can be syntactically checked through that background UML model using Eclipse's built-in OCL parser. The plain OCL parser thus enable the syntactical verification of plain OCL constraints.

Alloy Translator

The translation from plain OCL to Alloy uses and extends an Eclipse's built-in visitor pattern called "AbstractVisitor", defined implicitly by Eclipse to debug plain OCL constraints. When OCL constraints are being visited, each OCL construct on the constraint refers to one of the methods of the visitor pattern. For example, the type Person in the OCL expression *self.oclAsType(Person)* refers to the visitor method called *TypeExp(…)* which in turn reflects the meta-class *TypeExp* of the abstract syntax of plain OCL as defined by the OMG [5]. The dot notation on the other hand refers to the visitor method called *PropertyCallExp*(…) reflecting the meta-class *PropertyCallExp* at the OCL abstract syntax. In the existing plain OCL infrastructure, an extension of this visitor called "OCL2AlloyVisitor" is defined to develop a transformation from OCL textual constraints into Alloy textual statements. This means that when OCL constructs are being visited the Alloy visitor generate as output the respective Alloy mappings for each construct. The Alloy translator thus enable the validation of plain OCL constraints via Alloy simulation and analysis.

6.2 Implementation Extension Approach

Our approach for extending this existing plain OCL infrastructure with our Temporal OCL language encompasses three temporal extensions of the three components presented previously such as the plain OCL editor, plain OCL parser and the Alloy translator, as depicted in Figure 26, respectively.

We extend the plain OCL editor with a support for Temporal OCL's syntax highlight and code-completion; the modeler is thus able to write/edit textual temporal constraints in that editor with both feature supports.

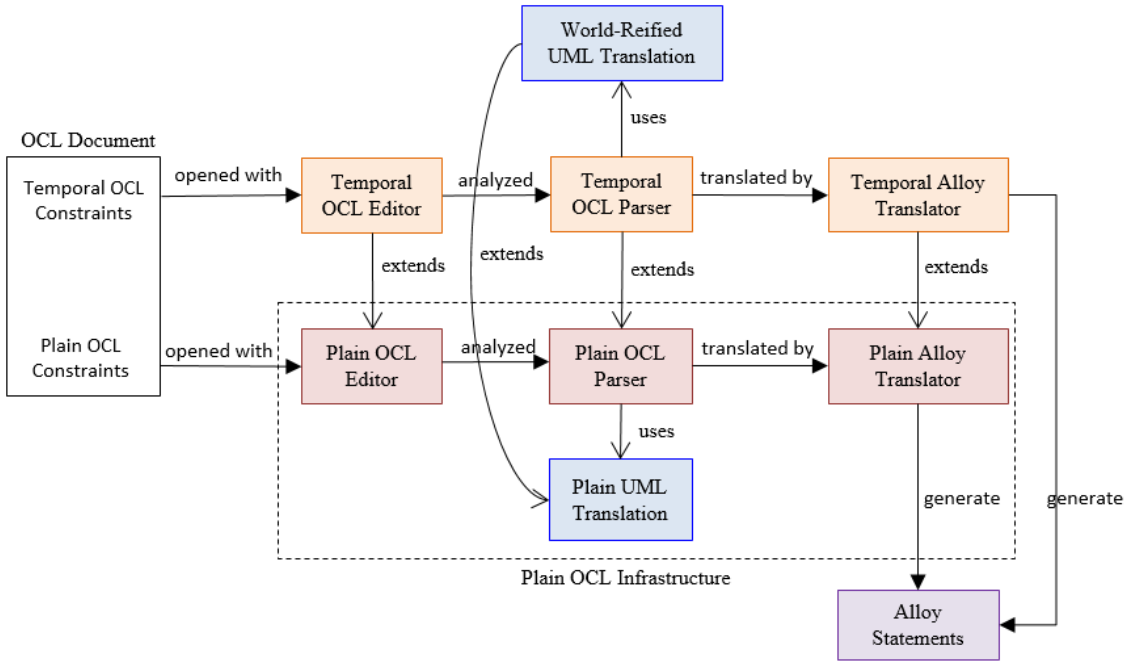

Figure 26 Temporal Extension of Existing Plain OCL Infrastructure

We extend the plain OCL parser so that the temporal textual constraints in that editor can be analyzed syntactically in accordance with our set of adjustments for OCL and the OMG's specification. This extension of the plain OCL parser (i) incorporate the few adjustments in OCL such as with type conformances and *allInstances* built-in operations, (ii) implement a syntactical analysis for our own concrete syntax in the definition of trans-temporal facts, and (iii) extend the translation to plain UML in background using our world reification approach (called here as *World-Reified UML Translation*). The result should be a temporal OCL parser that incorporates these adjustments, definitions and translation but that still uses the plain OCL parser (in other words the Eclipse's OCL parser) to syntactically verify all remaining plain OCL constructs. The temporal parser thus should receive temporal constraints in textual form (from the editor or as text documents) and returns the respective parsed constraints as Java objects to other constraint translations.

Lastly, we extend the existing transformation to Alloy to include, besides the plain OCL operators, our set of Alloy mappings from the adjustments made in OCL, our built-in world operations and built-in temporal navigations, and our definition of historical relationships and dynamic constraints. The analyzed temporal

OCL constraints are transformed to Alloy textual code and added to the resulting Alloy specification in order to be fed into the Alloy Analyzer tool.

In the sequel, we present and discuss some important points about (i) the implementation of the temporal extension of the plain OCL editor (Section 6.3), (ii) the implementation of the syntactical analysis of our set of adjustments for the built-in operations of plain OCL (Section 6.4) and finally (iii) the implementation of a translation to a world-reified model in background in plain UML (Section 6.5).

6.3 Extending the Plain OCL Editor with Temporal OCL

We extended the plain OCL editor with the support for additional constructs defined by our temporal OCL language. We defined our language's syntax highlight, vocabulary and code completion using a .flex and .java files that are generated automatically by the software called *TokenMaker*. The custom Swing text area used as our editor accepts a language's configuration through those files[4]. We thus extended the previous code-completion feature, which only supported plain OCL constructs to support our own set of code-completions such as the built-in operations for worlds, paths and endurants, plus dynamic invariants and historical relationships, as demonstrated in Figure 27.

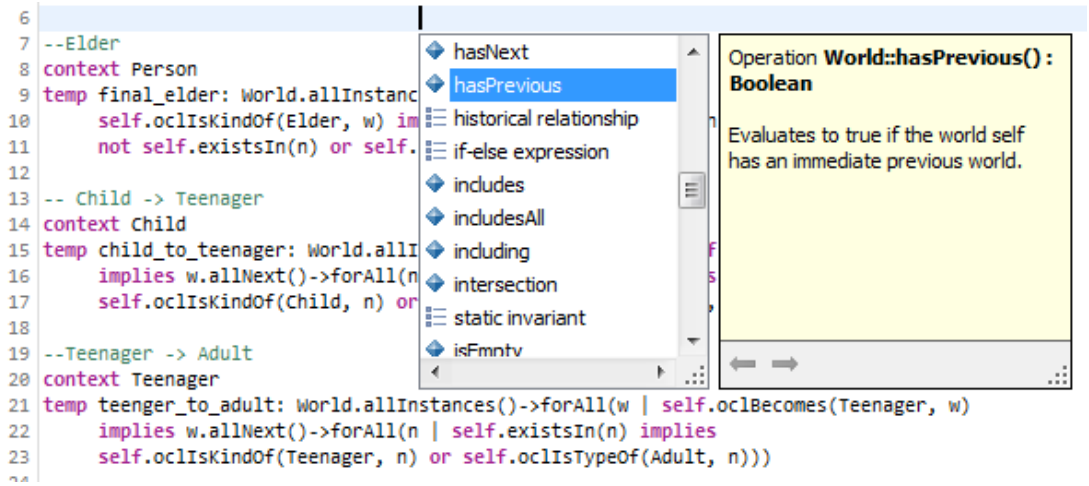

Figure 27 Code Completion Activated at the Temporal OCL Editor

[4] http://fifesoft.com/

The figure shows the code-completion feature activated in the temporal OCL editor. In the list of possible constructs that can be used are all temporal elements alongside plain OCL elements, with a proper description for each of operation, expression and constraint supported in the language. We believe that this feature may facilitate the learning process of the Temporal OCL language.

6.4 Parsing the Temporal Adjustments for Plain OCL

Eclipse's OCL language implementation called Classic OCL is currently used in the OCL infrastructure to analyze plain OCL constraints with OntoUML models (through a background translation to plain UML). It analyzes if the constraints are syntactically valid according to the concrete syntax defined by the OMG. Our Temporal OCL requires only few adjustments to plain OCL defined by OMG such as with type conformance built-in operations (i.e. *oclIsKindOf*, *oclIsTypeOf*, *oclAsType* and *oclType*) and the *allInstances* built-in operation. Only these few adjustments are required for OCL to behave as a temporal language (besides the inclusion of an entire world structure, temporal navigations and world operations as discussed in this work). In this sense, the only modification required in plain OCL is to male explicit a world parameter in the object built-in operations and *allInstances*. Classic OCL does not allow us extend and modify this OCL meta-operations as they are built-in in the language. Our solution to this was to define a (automatic) textual processing of the temporal OCL textual constraints before the Eclipse's OCL syntactical verification as executed by the Eclipse's OCL built-in parser (using our World-Reified plain UML model of background as the context to the analysis).

For example, in the parsing of the temporal OCL expression *oclIsKindOf(Child, w)* we textually process that expression by storing the world parameter *w*, passing forward only the expression *oclIsKindOf(Child)* to Eclipse's built-in OCL parser. The built-in operation *oclIsKindOf(T, w)* is not supported natively in plain OCL and therefore Eclipse's parser cannot parse it. On the other hand, *oclIsKindOf(T)* can be syntactically checked with the world-reified UML model of background using Eclipse's OCL parser. This means that our temporal parser guarantees that the world parameter introduced is always valid (i.e. it is already declared in previous expressions at

that same OCL document and does not have any invalid character as defined by the OMG). We practically simulated the parsing of Eclipse with respect to that parameter. We can access, later on, that world parameter in our temporal parser. For example, when that expression needs to be transformed to Alloy, we need to use that world parameter in the Alloy mapping generating the Alloy expression "self in w.T". The same pattern of textual processing is applied to the remaining temporal OCL built-in object operations such as *oclIsTypeOf*, *oclAsType* and *oclType* and the classifier built-in operation *allInstances*, as they all receive an additional world parameter. In Figure 28 we demonstrate the parser exception thrown at our temporal parser according to these adjustments in these OCL built-in operations. The figure shows a parsing error in which *oclIsKindOf* misses a world parameter.

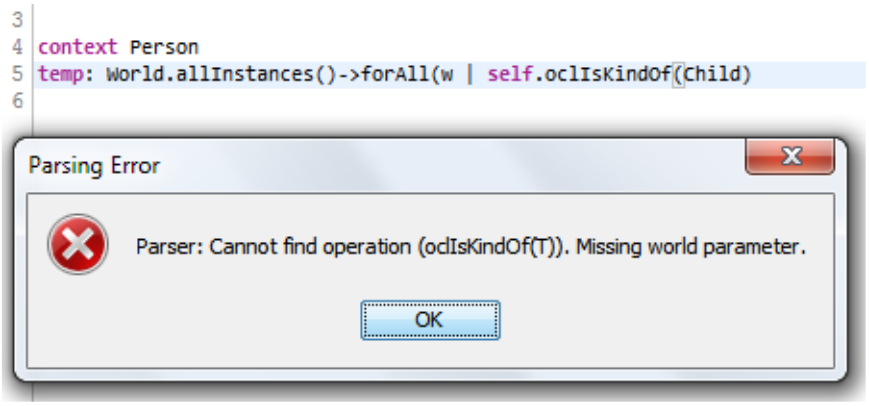

Figure 28 Parsing Exception Thrown at the Temporal OCL Parser

In addition, Temporal OCL defines two built-in operations, which are inspired by the previous built-in operation *oclIsKindOf(Child, w)*. These operations are *oclBecomes(Child, w)* and *oclCeasesToBe(Child, w)*. They receive a Classifier T (e.g. Child) and a World *w,* as parameters, and check if the object is classified (or ceased to be classified) as T at *w*. We apply the same idea of textual processing for these additional operations but the storage is performed regarding the other parameter (the type parameter). The temporal parser store the parameter *Child* passing forward only the expression *oclBecomes(w)* or *oclCeasesToBe* to Eclipse's built-in OCL parser. As far as we know, Classic OCL does not allow a new operation to be defined using the *Classifier* UML meta-class. We have thus defined *oclBecomes* and *oclCeasesToBe* in our world-reified plain UML model of background using only the world parameter such as *Endurant::oclBecomes(w: World)* and *Endurant::oclCeasesToBe(w: World)* (we did not

show these definitions implemented in OCL in previous chapter since their definition in OCL were driven by implementation concerns). In the same way, we can access that operation's type parameter with our temporal parser, for example, in the mapping to Alloy where that parameter needs to be mapped in an Alloy expression.

Finally, we used this textual processing approach to parse syntactically historical relationships defined using our concrete syntax in Temporal OCL. After parsed at an OCL document, these historical relationships are created at the world-reified plain UML model of background. They are also stored in our temporal OCL parser in order to further be mapped to the Alloy logic language.

6.5 World-Reified Model with Constraints in Background

The existing OntoUML infrastructure was developed by *Carraretto* [35] using an older version of Ecore/OCL (Eclipse Galileo[5]). The OntoUML metamodel was designed in Ecore but not implemented as an extension of UML. With regard to implementation, OntoUML is not strictly speaking a UML profile. OntoUML cannot natively benefit from Eclipse's OCL support, which is only available to UML and Ecore. The OntoUML Ecore metamodel specifies models in Eclipse's XMI format. As XMI models, OntoUML does not support OCL. In order to simulate that support as a native support, the authors in [22] defined that each OntoUML domain model should have a background UML model correspondent to orchestrate the binding between OntoUML and OCL. We extended the existing translation to a plain UML background model with the inclusion of our world reification approach. We include in the UML model a world structure, a set of temporal built-in navigations, and world, path and endurants built-in operations. This enables our temporal OCL constraints to be analyzed syntactically with that world-reified UML model for evaluation of OCL contexts, navigations and expressions. The background model is additionally enriched with several constraints to ensure that the OntoUML model semantics is preserved. The background model is a UML artifact and the constraints a separate OCL textual document such as showed in Figure 29. We depict the back-

[5] https://eclipse.org/galileo

ground model enriched with constraints for our running example about people and marriages using the Eclipse Platform. Each time a domain model is written in OntoUML and the (temporal) OCL parsing is required, we transform the OntoUML model into UML (with worlds-reified in case of Temporal OCL).

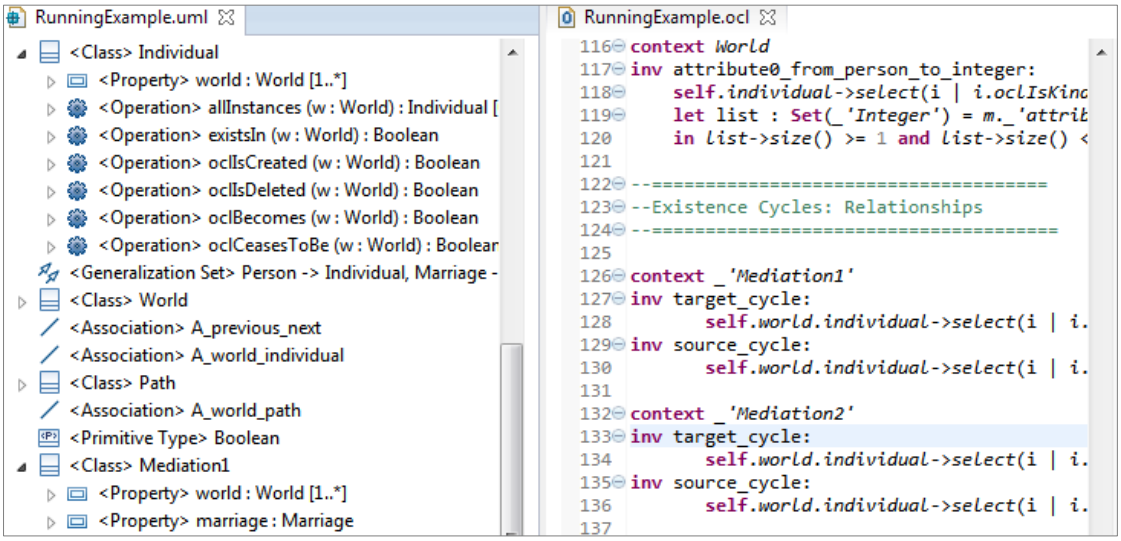

Figure 29 Automatically Generated Background Artifacts

6.6 Temporal Tooling Within OLED

We have incorporated our temporal extensions for the plain OCL editor, plain OCL parser and translation to Alloy within the OLED Tool. We depict a screenshot of the tool with regard to the Temporal OCL support in Figure 30. The figure shows our temporal OCL editor opened at the center of the tool and the project browser at the right side showing all elements pertaining to the OntoUML diagram. At the bottom there are three tabs opened. First the welcome page of the tool, second the OntoUML diagram called "Diagram0" with our running example of people and marriages and third the dynamic constraints written in Temporal OCL as discussed in this work opened in an OCL document called "Document0". It is interest to mention that the temporal OCL editor, as an extension of the plain OCL editor not only supports dynamic constraints but also plain OCL constraints. There is no need to separate them (unless by a decision of the modeler). The figure shows also the successful message displayed at the temporal parsing of the dynamic constraints thus validating our approach for the adjustments made in OCL and our world reification approach. The figure finally shoes the Alloy dialog to configure the visual

simulation. The user might translate dynamic constraints into Alloy facts, predicates (for running simulations) and assertions (for checking assertions). With this technique, the user can validate if the model is under-constrained or over-constrained according to one's domain conceptualization

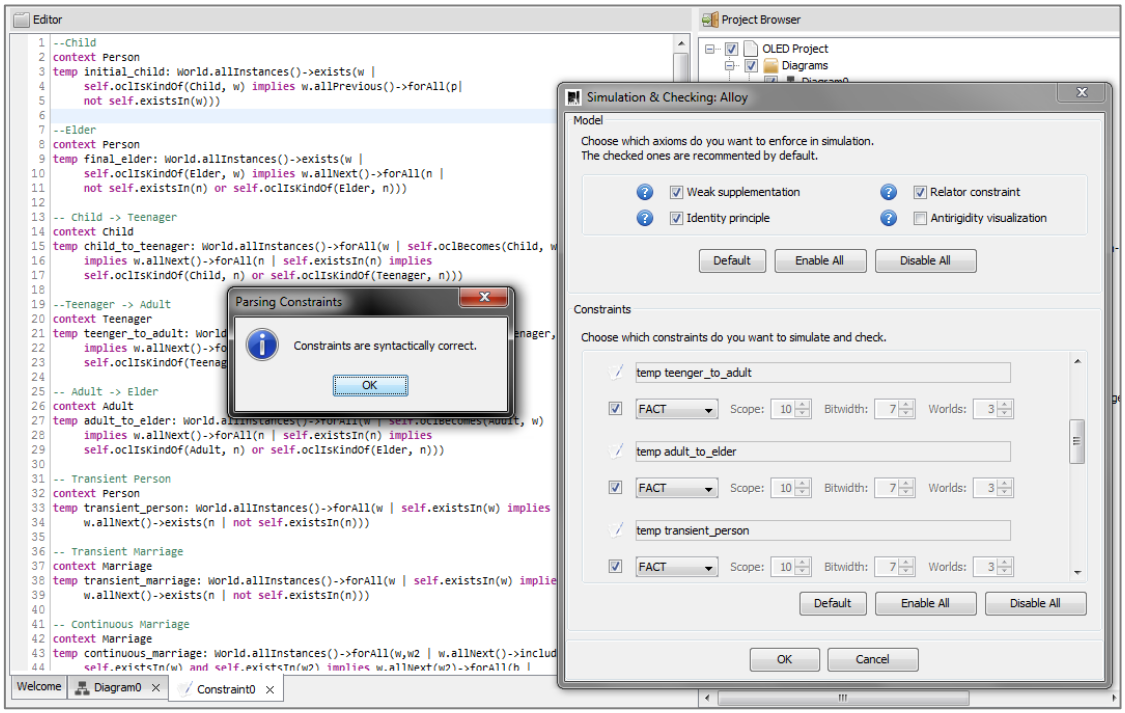

Figure 30 Temporal OCL Tooling Within OLED

6.7 Final Considerations

In this chapter, we have discussed our approach for the implementation of this work. In particular, we have extended the former plain OCL support in the OLED tool to include a support for our Temporal OCL language. We extended the plain OCL editor, the plain OCL parser (with a background translation to UML with a word reification approach) and the translation to Alloy. All the details and source code of our implementation is available at the host site of OLED [34]. In particular, we developed and extended the following projects: */br.ufes.inf.nemo.ocl*, and */br.ufes.inf.nemo.ontouml2uml*. We have thus validate our modeling approach by incorporating Temporal OCL into the OLED tool. In the next chapter, we discuss related work regarding other temporal conceptual modeling languages and other approaches that use the Alloy lightweight formal method for validation.

7 Related Work

In this chapter, we examine the state-of-the-art in temporal extensions and validation approaches of conceptual modeling languages such as OCL and UML. In particular, we discuss a temporal extension for plain UML and OCL that was proposed by Cabot et al. [12] (Section 7.1) and a set of existing temporal extensions for OCL (Section 7.2). We then discuss some related work regarding the validation of (temporal) conceptual modeling languages using a lightweight formal method (Section 7.3). Finally, we present a summary of the existing approaches on conceptual modeling with UML and OCL evaluating each set of approach with regard to some defined criteria (Section 7.4).

7.1 A Temporal Extension of plain UML and OCL

In [12], *Cabot et al.* extended both plain UML and static OCL with temporal notions in order to cope with the representation of temporal information in UML models. In particular, they extended UML with a set of temporal aspects and OCL with notational devices aiming to refer to immediate past values of model properties. They argue that using this extension, a modeler may use UML/OCL, which are primarily static modeling languages, as if they were indeed temporal modeling languages. The dynamic aspects introduced in UML by Cabot and colleagues are classified in two major sets: *durability* and *frequency* as depicted in Figure 31.

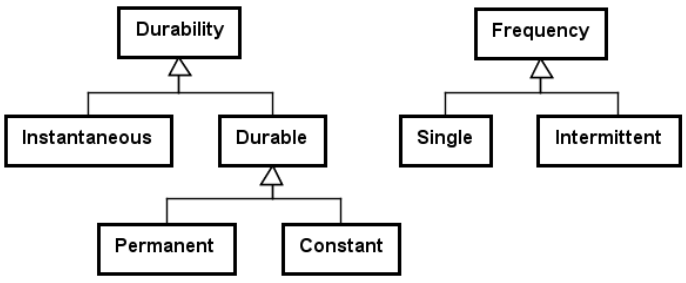

Figure 31 Temporal Aspects of Cabot's Temporal Extension of UML

Durability refers to the persistence of the instances of a UML type (a class or a relationship). These aspects can be applied not only to UML classes but to UML relationships as well. In UML terminology, a class and a relationship are both UML

types. Durability can be divided into the dynamic aspects of *Instantaneous* and *Durable*. If an instance (or relation) is classified in a single point in time not persisting to the next instant after that, the instance or relation is called *instantaneous*. If an instance or relation is classified during a certain interval, then they are called *durable*. Durable in turn may be classified as *Permanent* or *Constant*. Permanent means that once an instance or relation is classified as a type, the instance or relation will be always of that type from that point forward. Constant on the other hand states that the instance or relation will be always classified as that type. *Frequency* refers to how many times the instance or relation appears to be of a type. For example, a *Single* frequency defines that the instance or relation is classified only during a single time interval. *Intermittent* means that the instance or relation is classified during as many time intervals as desired. Using these set of dynamics as UML tagged values, the modeler can restrict the way the instances behave with time. The authors defined all six valid combinations w.r.t. these dynamic features (there are some combinations that are logically invalid together). The valid combinations are (i) *instantaneous single*, (ii) *instantaneous intermittent*, (iii) *durable single*, (iv) *durable intermittent*, (v) *permanent* and (vi) *constant*.

Since we have focused here on endurants, which are all considered continuous and non-instantaneous, we have not included the frequency *intermittent* or the distinction concerning *durability*. We thus rule out the combinations (i), (ii), (iii) and (iv), as they would not make sense in UFO. The frequency *single* is analogous to our *continuous* dynamic aspect and *permanent* and *constant* are analogous to our *permanent* and *eternal* dynamic aspects. We thus encompass combinations (v) and (vi).

With regard to the authors' temporal extension of OCL, they indexed all UML attributes and UML relationships as well as the *allInstances* built-in OCL operation with a time parameter. Any UML attribute or relationship in their approach has two implicit operations. For example, for an UML attribute called *salary*, owned by a class Employer, there are implicit operations such as *salaryAt(t: Instant)* and *salaryAtOrBefore(t: Instant)*. The former operation retrieves the salary of an employer at a time *t* whilst the latter, if no value exists at *t*, retrieves the latest value before *t*. The class *Instant* is a time instant which was reified (but the authors did not provide fur-

ther details about their time reification). Further, they also showed how these temporal and implicit OCL operations could be expanded into standard OCL expressions [12]. Our approach for extending OCL on the other hand, allows a built-in world structure and thus allows retrieval of past values at any past time as well as many other temporal operations which include a temporal version for the *allInstances* built-in OCL operation.

Finally, with regard to implementation, the authors implemented their temporal extension of plain UML and OCL in a CASE tool called Objecteering/UML[6]. This tool shows the validity of the approach providing the users the ability to graphically specify the dynamic features as UML tagged values and retrieve past values using the implicit OCL pre-defined temporal operations. Objecteering/UML however is limited to an older version of UML, only supported by another tool called ArgoUML[7]. We in contrast, implemented our OCL temporal extension in a tool called OLED which supports the OntoUML conceptual modeling language and can export OntoUML models as UML profiles as supported by the Eclipse UML2 project. OLED can also import models designed with the Enterprise Architect (EA) tool and provide a full support for both static OCL and our temporal extension of OCL with parsing, edition, code-completion and language syntax-highlight.

7.2 A Set of Existing Temporal Extensions of OCL

There have been many proposals in literature that aimed at extending OCL in order to cope with dynamics/temporal aspects of systems [10, 11, 13, 14, 15, 16, 17, 18]. *Gogolla and Ziemman*'s extension of OCL [18], named TOCL, is based on a set of (finite-) Linear Temporal Logic (LTL) operators. They formally extended the syntax and semantics of OCL invariants and pre- and post-conditions with LTL logic operators such as "always", "sometime", "next" and the concept of process types [18]. They introduced an environment's index to characterize the temporal evolution of

[6] http://www.objecteering.com

[7] http://argouml.tigris.org/

the system and its current state. They give no explanation of how the presented formal notions can be implemented.

Conrad and Turowski [13] extended OCL with Linear Temporal Logic (LTL) operators in order to specify software contracts for business components, where contracts are mainly OCL pre- and post-conditions. They used future and past operators of LTL such as "always" and "sometime", and future operators of LTL such as "until" and "before". However, they did not consider the logic operators "next" and "previous" [13, p. 162] but introduced the "initially' operator to refer to the initial state of the system/business component.

Bill et al. [10] presented an OCL extension named cOCL, based on computational tree logic (CTL). Their approach is very similar to that of Gogolla and Ziemman as they formally extended OCL's syntax and semantics with logic operators. They considered CTL operators such as "next", "weak until" and "strong until", which can be quantified either existentially or universally. Their verification framework consists of cOCL specifications and a model checker called MocOCL that can verify cOCL constraints.

Flake and Mueller [15] defined a state-oriented Real-Time extension of OCL whose semantics is given through a mapping to clocked CTL logics (CCTL). They focus on the specification of real-time systems. They extended OCL by describing a UML profile for specification of state-oriented real-time constraints demonstrating a M2 layer-based extension. Furthermore, they formalized UML State-charts and added it to (Ritcher's) object model definition as a means to complete the formal semantics of OCL, which lacked precise meaning with respect to dynamic behavior in UML models.

Differently from these approaches, we do not use tense logic operators explicitly, choosing to use reification of world states to obtain the expressiveness that would be obtained with tense operators. Extensions based on modal/tense logic operators re-quire a level of logic expertise that most modelers are not expected to have.

Distefano et al. [14] defined an object-based extension of CTL called BOTL (Object-Based Temporal Logics), their own logic formalism inspired by OCL, to define specifications of static and dynamic properties in object-oriented systems. However, they did not consider inheritance or subtyping. The mainly concern is to apply model checking approaches into object-oriented systems. They presented a mapping from part of OCL onto BOTL providing formal semantics to a large part of OCL. There is no extension of OCL by temporal operators, but a theoretical precise mapping of a part of OCL into BOTL. BOTL "looks syntactically very similar to CTL" [14] and although BOTL's concepts are defined clearly and precisely, no tool support is actually provided.

Mullins and Oarga [17] extended OCL with CTL operators and some first-order features. Their extension termed EOCL is largely inspired by BOTL [14] (but including inheritance) and based on the OCL extension framework of *Bradfield et al.* [11] which defines the language as a two-level logic, wherein the upper level is CTL extended with quantifiers and the lower level is a significant fragment of OCL. The SOCLe tool translates exactly one UML class diagram, one state-chart and one object diagram into an Abstract State Model (ASM) specification, which in turn is translated into an execution graph (an Object-Oriented Transition System-OOTS implementation) that can verify on-the-fly EOCL constraints. Their extension is briefly presented with verification issues in mind. There is no tool available at their project site[8].

Bradfield et al. [11] proposed a formalism, termed Oμ(OCL), which is a two-level logic language called Observational Mu-Calculus, where the modal mu-calculus is the upper level language and OCL is the lower-level logic language. However, Oμ(OCL) requires such understanding of temporal logics that is unrealistic to expect most developers to acquire it [11, p.2]. In order to remedy this issue, the authors suggested the design of OCL temporal templates by users, with users-own friendly syntax, and that they automatic translate from the templates into Oμ(OCL). They give no means to OCL developers to implement such templates.

[8] http://www.polymtl.ca/crac/socle/index.html

Kanso and Taha's extension of OCL [16] committed to a different approach as it was based on Dwyer's property specification patterns [28] i.e. the temporal OCL constraints are based on a set of temporal patterns rather than in a set of temporal logic operators or formalisms. Dwyer et al. created a number of mappings from the temporal and real-time property patterns to corresponding formulae in CTL, LTL and Mu-Calculus[9]. Kanso and Taha extension of OCL introduced the notion of events (such as the operation and state-change events) in the Dwyer's patterns. They introduced keywords to increase the former Dwyer patterns expressivity, for example, the "at most" or "at least" keywords, which limits the number of times that an event can happen in a given scope. They implemented the pattern-based OCL extension in an Eclipse/MDT OCL Plugin, which allows OCL temporal constraints to be defined with Ecore/UML models. However, the set of temporal patterns are not suitable to OntoUML's set of requirements, such as the initial classification dynamic aspect, usually, due to the pattern's closed/open edges of intervals

7.3 Existing Approaches on Validation of Conceptual Models Using the Alloy Lightweight Formal Method

Several approaches in the literature have aimed the analysis and validation of plain UML conceptual models and standard (static) OCL constraints e.g. HOL-OCL [7], USE [6], CD2Alloy [29], UML2Alloy [8]. In particular, a number of them [8, 9, 29, 31, 32] have used Alloy as a lightweight formal method for validating structural conceptual models written with UML/OCL. In [8], *Anastasakis et al.* present one of the first extensive approaches for automatic translation of UML+OCL models into Alloy for purposes of model verification and validation. Their tool is called UML2Alloy and although it considers both UML and OCL, it does not support several standard OCL operators while just a subset of UML is considered. *Cunha et al.* [9] extended the mappings of *Anastasakis et al.* to support UML qualified associations and dynamics of properties such as the UML read-only feature (immutability).

[9] http://patterns.projects.cis.ksu.edu/

They defined a state local signature called Time in the Alloy resulting specification to handle correctly dynamics of properties and pre- and post- conditions.

Maoz et al. [29] translated UML, particularly class diagrams, to Alloy and then from Alloy's instances back to UML object diagrams, considering both multiple inheritance and interface implementation. They use a deeper embedding strategy as not all UML concepts are translated directly to a semantically equivalent Alloy constructs. For instance, UML multiple inheritance is transformed to a combination of Alloy facts, predicates and functions. This strategy enables the support of analysis of class diagrams, checking if one class diagram is a refinement of some other class diagram [29, p.2]. Their translation is implemented fully in a prototype plugin in Eclipse called CD2Alloy, which can (optionally) hide the Alloy resulting specification from the modeler. However, the translation does not consider standard OCL. Besides, the Alloy resulting specification is difficult to read, less understandable and computationally more complex than other approaches [29].

Massoni et al. [32] proposed a transformation of only a small subset of UML (class diagrams with classes, attributes and associations) annotated with OCL standard invariants to Alloy. They specify the translation merely systematically and manually. *Kuhlmann et al.* [31] on the other hand defined a translation from UML and standard OCL to Relational Logics and a backwards translation from relational instances to UML model instances. Relational Logics is the source for the Kodkod SAT-based model instance finder used by Alloy.

None of these approaches completely supports dynamic and multiple classifications, which is essential for ontology-driven conceptual modeling with OntoUML. In fact, besides dynamic and multiple classifications, the meta-properties that characterize many of the ontological categories and relations in an ontologically well-founded language are modal in nature. As discussed in [2], the modal distinctions among object types and part-whole relations are paramount from an ontological perspective and play a fundamental role in ontology engineering and semantic interoperability efforts. Moreover, none of these approaches has fully coped with dy-

namics. The only support for dynamics is the UML read-only feature i.e. immutability, proposed by *Cunha et al.*.

There exists mainly two approaches for the validation of OntoUML structural conceptual models, one based on *Ontological Semantic Anti-Patterns* [23] and another based on *Alloy simulation* [22, 23, 30, 33]. A semantic anti-pattern [23] is a recurrent conceptual modeling decision, that although syntactically valid, it is prone to lead to domain misrepresentations, i.e., it might indicate that the model is under-constrained, over-constrained, or that an element is classified with the wrong category or that there might be elements missing from the model. In this approach, the conceptual model is checked against a catalogue of semantic anti-patterns. If an anti-pattern is found, the approach suggests a wizard to aid evaluation and if necessary, correct the anti-pattern occurrence, for instance, by adding constraints or changing an elements' category.

Benevides et al. [30] and *Braga et al.* [33] were the first to propose Alloy as a formal method of validation of OntoUML structural conceptual models. The Alloy technique used to validate OntoUML structural conceptual models is different from those based on UML because all OntoUML's modal features and the language constructs affected by them require a special treatment in Alloy. In this technique, it should be considered the dynamics implicit in OntoUML, such as the different types of rigidity and immutability, and a support for a world states structure. *Sales* [23] recently combined these former approaches in order to address several issues that hindered their usage in practice, issues involving performance, coverage, mappings and implementation. In addition, *Sales* introduced some features such as UML redefinition and UML subsetting [23] while keeping the branching world structure of *Benevides et al.* [30]. *Guerson et al.* [22] in turn extended *Sales*' approach in order to support standard OCL domain constraints in the validation with Alloy. They defined a translation from standard OCL constraints into Alloy in pace with all OntoUML's modal features and existing OntoUML mappings. However, their approach does not support richer dynamic aspects as OntoUML was still limited w.r.t. to dynamic aspects while OCL, in the context of OntoUML, was yet simply a static constraint language.

7.4 Summary of Existing Approaches

We can view all the existing approaches of conceptual modeling languages with UML and OCL (including the existing temporal extensions) into four major sets:

- The pure (plain) UML/OCL approach [4, 5] using standard UML and OCL as defined by OMG to represent static structural aspects of conceptual models (*Existing Approach 1*);
- The pure (plain) OntoUML/OCL approach [2, 5] applying a foundational ontology to evaluate and give semantics to UML class diagrams increasing its expressivity but with limited dynamic aspects such as rigidities and dependences (*Existing Approach 2*);
- The temporal UML/OCL approach of Jordi Cabot [12] which extends UML with dynamic features and OCL with devices to retrieve immediate past values (*Existing Approach 3*); and
- The temporal OCL approaches [10, 11, 13, 14, 15, 16, 17, 18] which extends OCL based on a logic language, created new logic formalisms inspired by OCL, defined an OCL extension based on a set of temporal property patterns and etc. (*Existing Approach 4*);

Table 7 presents an overview of these four major sets of approaches according to some defined criteria. With regard to documentation of the formal semantics of the respective conceptual modeling languages, plain UML and OCL are defined by the Open Management Group (OMG) [4, 5] whilst OntoUML was defined by Guizzardi in his PhD thesis [2]. The semantics of the temporal extension of plain UML and OCL developed by Jordi Cabot and colleagues [12] is not fully characterized. The authors formalized only the *instantaneous* dynamic aspect whilst the other aspects were explained only intuitively using natural language. They also presented only one type of expansion to standard OCL and two examples of implicit OCL temporal operations used to retrieve past values. We thus judged that this was sufficient to meet only *partially* the criteria of formal semantics documentation.

Table 7 Summary of Existing Approaches

Existing Approaches x Criteria		1. UML + OCL (OMG)	2. OntoUML + OCL (Guizzardi)	3. Temporal UML + OCL (Cabot et al.)	4. Temporal OCL (Extensions)
Documentation	Formal Semantics	Yes. (OMG Spec)	Yes. (Guizzardi) (PhD Thesis)	Partially. (Extension of UML/OCL)	Partially. (e.g. Kanso and Taha)
Validation	Alloy Simulation and Analysis	Yes. (CD2Alloy, UML2Alloy)	Yes. (OntoUML2Alloy)	No.	No.
Expressiveness	Dynamic Classification	No.	Yes.	No.	Not Applicable
	Multiple Static Classification	No.	Yes	No.	Not Applicable
	Richer Temporal Constraints	No.	Partially. (Only Rigidity and Dependences)	Partially. (Only Durability and Frequency)	Yes. (Logic-based Extensions)
	Time Structure	No.	No.	No.	Yes. (Linear, Branching)
	Trans-Temporal Facts	No.	Yes. (Growing Block View Theory)	No.	No.
Implementation	Tool Support	Yes. (Eclipse)	Yes. (OLED)	Yes. (Objecteering)	Partially. (e.g. Kanso and Taha)

Regarding the set of OCL temporal extensions, they were mostly defined by short publications which we judged not having a complete formal characterization of the language's semantics. Only Kanso and Taha's extension of OCL formalized their

extension recently in [16]. Some of the other extensions do not extend standard OCL e.g. BOTL, Oμ(OCL), and thus do not benefit from OMG's documentation while others lack tool support that could help us in understand more clearly the language. We thus judged that the set of existing OCL extensions achieves only *partially* the criteria of formal semantics documentation. Our modeling approach, in contrast, meet this criteria as we provide proper formalization for all the dynamics introduced in Chapter 3 and explain our OCL temporal extension in Chapter 4, which is a very similar extension to standard OCL with very few adjustments to it. We thus also benefit from standard OCL semantics and documentation.

Our second criteria concerns the validation of conceptual models created in these approaches using the Alloy logic-based language. There have been several approaches for Alloy simulation with UML [8, 9, 29] and OntoUML [22, 23] but with regard to temporal extensions, none of these approaches support Alloy simulation. Therefore, our approach meets this requirement by extending the existent OntoUML and OCL approach [22, 23] to Alloy Simulation as discussed in Chapter 5.

With respect to the expressiveness of these (temporal) conceptual modeling languages, as only OntoUML [2] supports dynamic classification (e.g. the classification of persons into life phases: child, teenager and adult; the classification of persons into roles in particular contexts) and multiple classification (e.g. the classification of person according to orthogonal classification schemes such as {healthy, sick} and {man, woman}). Although dynamic and multiple classifications are in principle supported by UML, most UML approaches that establish formal semantics and analysis/simulation do not address these features. This renders the UML approach less suitable to enable the expression of important conceptual structures that rely on dynamic and multiple classifications. Dynamic and multiple classifications are thus features of diagrammatic modeling languages such as UML and OntoUML, and as such, they are not applicable to textual constraint languages such as OCL (and extensions). Our modeling approach meets this requirement extending OntoUML and using an additional temporal OCL extension that complements OntoUML.

With respect to temporal constraints, UML is purely a static modeling language. OntoUML [2] introduces dynamic aspects but only the different types of rigidity and immutability. The temporal extension of plain UML and OCL of Jordi Cabot and colleagues [12] supports a limited number of dynamic aspects such as different types of durability and frequency as their temporal extension of OCL is only able to retrieve past values. The other existing OCL temporal extensions are (a priori) expressive enough to represent richer logic temporal constraints. This may only vary depending on the OCL extension used, according to the expressiveness of each approach, for example, Taha's extension of OCL [16] is based on a set of Dwyer's patterns and is not suitable to OntoUML's set of requirements, such as the initial classification dynamic aspect. Our modeling approach is thus able to represent richer dynamic constraints using a temporal OCL extension as discussed in this work.

With concern to the time structure embedded in these approaches. Plain UML does not support any. OntoUML reflects a notion of possible worlds from the alethic modality (the logics of necessity and possibility) with no temporal interpretation. A temporal structure of worlds is important if we want to express the behavior of model entities. The temporal extension of plain UML and OCL of *Cabot et al.* [12] does not specify any structure of time. The other existing OCL extensions define mostly linear and branching structures of time such as those from LTL and CTL [10, 15, 18]. Our modeling approach, adopts a temporal interpretation similar to CTL logics i.e. a structure of branching worlds (a tree with branches towards the future).

The OntoUML approach is able to express historical relationships and transtemporal facts [21] but only if existence is reified w.r.t. all entities of the domain (OntoUML models aligned with the growing block universe view theory). None of the other three approaches is able to represent transtemporal relationships and facts. Our modeling approach in contrast is able to support both of them, regardless of whether the model is aligned with the growing block universe theory.

Finally, with respect to tool support, there are many CASE tools to support standard UML (e.g. Eclipse, Astah, Enterprise Architect, Visual Studio, ArgoUML)

and OCL (e.g. Eclipse). From those, Eclipse[10] provides a full support to both UML and OCL, according to the OMG specification. OntoUML in turn has a tool called OLED[11] (OntoUML Lightweight Editor) which is an editor to build, validate and implement domain ontologies (i.e. ontology-based conceptual models). The temporal extension of plain UML and OCL of Jordi Cabot and colleagues is implemented into a tool called Objecteering/UML[12]. However, they still use an older version of UML (version 1.4). Most OCL temporal extensions do not provide tool support e.g. [11, 14, 18], they do not allow their respective language to be used with conceptual modeling languages such as UML or OntoUML, with the exception of Kanso and Taha's extension of OCL [16], which is implemented in an Eclipse Plugin to represent temporal properties on UML/Ecore models. We implement full support for temporal OCL in the OLED tool, including syntax verification and validation using a lightweight formal method.

[10] http://eclipse.org/eclipse/

[11] https://code.google.com/p/ontouml-lightweight-editor/

[12] http://www.objecteering.com/

8 Concluding Remarks

In this chapter, we present concluding remarks of this work. In particular, we discuss our contributions to the area (Section 8.1), limitations of our approach (Section 8.2) and future works (Section 8.3) pointing research directions in which we can address in a near future.

8.1 Contributions

Despite the recent advances in Conceptual Modeling, current approaches were not adequate to the representation of dynamic aspects in ontology-driven conceptual models, specifically, those written with OntoUML. The majority of existing approaches for dynamic aspects are concerned with temporal properties of computational systems, not with domain invariants. Further, some of them are defined as logic-based extensions, which require a level of logic-expertise of their users that we do not expect UML/OCL modelers to have. Other approaches focus on an extensions based on a set of temporal patterns, which are either restrictive or are not applicable to OntoUML. Finally, none of existing approaches is able represent facts involving historical dependences such as the so-called trans-temporal facts [21]. This research addresses these gaps supporting the expression of arbitrary dynamic invariants and historical relationships in OntoUML structural conceptual models.

This research contributes to increase the expressiveness of OntoUML proposing a simple extension that incorporates some additional dynamic invariants as part of OntoUML's modeling constructs in order to represent as accurate as possible conceptualizations about subject domains. We have formally characterized these additional dynamic aspects representing domain conceptualizations aligned with different philosophical theories about time and existence such as Presentism and the Growing Block Universe theory [21]. The constructs proposed for OntoUML reflect existence rules (permanence, transience and eternity) as well as classification dynamics (initial, final and general classifications).

This research contributes with a temporal OCL extension to cope with dynamic invariants and historical dependences in structural OntoUML conceptual models.

Our OCL extension requires only few adjustments to standard OCL; in particular, to four OCL type conformance operations and the "allInstances" operation. Our temporal OCL is expressive not only to incorporate user-defined dynamic aspects into structural conceptual models written with OntoUML but also to represent explicitly the implicit dynamic aspects of OntoUML stereotypes (i.e. rigidity, anti-rigidity, non-rigidity, semi-rigidity, immutability). We have demonstrated the expressiveness of our approach satisfying the requirements listed in Section 1.3, which include: representing existence rules (permanence, transience, eternity, and continuousness), classification rules (initial, final and general classification), derivations by past specializations [20], and specially, historical relationships and trans-temporal facts [21] (not addressed in any existing approach of conceptual modeling language).

We have developed a temporal OCL editor as an extension of the previous plain OCL editor [22]. Our OCL editor extension is embedded in the OLED tool, which is the existing model-based environment for modeling with OntoUML+OCL. Our temporal OCL editor's features include syntax verification, syntax highlight and code-completion, which are key to a productive environment for writing textual constraints.

This research contributes to facilitate the validation process developing a tool to aid modelers in checking if the dynamic constraints they have written indeed represent their intended conceptualization. The validation activity is a challenging activity since it requires trying to foresee all instantiations that can be allowed in a model. The validation of structural conceptual models enriched with plain constraints was already a hard task due to the complexity of the set of distinctions embedded in OntoUML's categories plus the new OCL constraints that can be added to the model. The validation enriched with dynamic constraints and historical relationships is thus even harder since it requires foreseeing not only a single world state deemed allowed by the model, but several states showing how entities undergo change. We contribute with a tool for the validation of dynamic OCL invariants and historical relationships using the existing validation approach based on Alloy. That is, we defined a semantics-preserving mapping from dynamic constraints and historical relationships to Alloy, in accordance with previous mappings from OntoUML+OCL [22, 23]. We

have thus incorporated our translation into the OLED tool, alongside with previous approaches, enabling the simulation/analysis of models enriched with dynamic invariants, contributing to the definition of highly accurate, ontology-based structural conceptual models written with OntoUML. In this tool, modelers can enforce a specific behavior for a certain entity of the model using our temporal OCL extension, but can also execute specific simulations by generating model instantiations as examples and checking assertions by generating model instantiations as counter-examples of desired and undesired behaviors according to a certain domain conceptualization.

Finally, this research contributes with a set of dynamic patterns for ontology-driven conceptual modeling. We have provided these dynamic aspects represented in temporal OCL as templates that can be added to OCL textual documents in the OLED tool through our code-completion feature at the editor. The code completion provides, alongside the pattern, a proper description for it. In this approach, users do not need to write again any of the dynamic rules in Temporal OCL discussed in this work. Users can simply use the templates from the code-completion feature, customizing them according to their own need and conceptualization.

8.2 Limitations

Our main objective in this research was to enable the representation of dynamic aspects in ontology-driven conceptual models. We have met this goal by proposing a simple dynamic extension for OntoUML, defining, and implementing a temporal extension for OCL. We have also provided a tool for the validation of models enriched with dynamics. However, we did not implement the OntoUML extension addressed in Chapter 3, as we did not incorporate the dynamic distinctions in the existing OntoUML infrastructure [35]. We judge that there is a trade-off regarding the amount of dynamics we can include in OntoUML's diagrammatic notation so that the resulting language can still be understandable and comprehensible. We believe further investigation is required to propose a suitable concrete syntax for OntoUML with respect to both categories of UFO and additional dynamic aspects proposed as part of our OntoUML extension. We judge that perhaps general classi-

fication rules (e.g. transitions of phases) could be better represented in a separate diagram instead of temporal OCL rules, such as for instance proposing a version of UML state-chart diagrams for OntoUML's phase transitions.

Our temporal OCL extension needs to adjust four plain OCL built-in type conformances operations and the *allInstances* built-in operation with the inclusion of a World parameter. However, we did not extend the OCL metamodel implementation to include that new *WorldType* meta-class to serve as a parameter for these new world-parameterized operations. We processed these adjusted plain OCL operations base on their textual syntax guaranteeing that they are syntactically valid when used in the context of temporal OCL expressions. A more systematic approach could lead to adjustments to the OCL metamodel in the future.

The existing validation approach with Alloy is currently limited w.r.t. the scope of simulation and analysis i.e. how many instances of each class and how many relations can be displayed at the visual simulation. An OntoUML model should not exceed fifteen to twenty classes (approximately) in order to be simulated and checked with Alloy (according to some of our own experience in practice). Our temporal interpretation assumes a world branching structure in the Alloy simulation. In this validation approach, Worlds are reified, model classes are Alloy binary relations and model relationships are Alloy ternary (and 4-ary) relations. The Alloy scope increases very rapidly using this mapping as the scope is set to each top-level Alloy signature, not to each OntoUML class represented as an Alloy binary relation. Our validation approach is thus limited w.r.t. scope since in order to simulate and check a behavior we need a considerable number of world states, endurants and their relations. Consequently, modelers are limited to simulate only parts of the models [23]. Further investigation is required to determine how to approach this sort of partial simulation.

8.3 Future Work

We plan further to investigate the application of temporal constraints on quality structures and regions of subject domains. Qualities structure and regions were left out in the scope of this research as they were recently addressed in [36] with the def-

inition of an infrastructure for qualities in OntoUML. We believe there are dynamic aspects of conceptual spaces, which needed to be specified in order to be represented as accurately as possible to a given conceptualization.

We plan to further investigate the quality of UFO-S (the foundational ontology of services [41]) and O3 (the organization foundational ontology recently developed in [40]) with regard to missing dynamic aspects using our modeling approach with temporal OCL and Alloy simulation. UFO-S and O3 were previously represented using only the limited dynamic aspects of OntoUML and the static aspects embedded in plain OCL. UFO-S and O3 were validated using the previous existing approach on Alloy simulation that dealt mostly with static aspects of phenomena. With our support for dynamic aspects, we should be able to represent dynamic aspects detected by desired and undesired behaviors in the respective foundational ontologies. We can thus contribute to enhance the quality of UFO-S and O3.

In order to further demonstrate the expressivity of our extension of OCL, we plan to compare our approach with other approaches such as (i) the temporal pattern-based OCL extension of [16], and (ii) the ontology-based behavioral specification language (OBSL) [39]. These approaches trade expressiveness for ease of use, so we expect that all of the constraints that can be expressed in these approaches can be also expressed with our temporal OCL.

OWL is an extension of RDF based on Description Logics to represent content in the context of Semantic Web. It has been used to implement reference ontologies to discover knowledge, annotate semantically its content and to publish it on the web. OWL does not support temporal aspects by nature and thus some reification approach is necessary in order to handle temporal aspects as the approach of [38]. We thus plan to investigate the representation of our reification approach and our temporal OCL extension into OWL building up from an OntoUML translation.

We plan to represent simulations scenarios [23] with our temporal OCL extension. Simulation scenarios define desired model properties as pre-defined test cases in Alloy simulation, enhancing the process of validation with Alloy making it easier and more accessible for users. We can improve even more this process if we provide

these simulations scenarios in a conceptual level (using temporal OCL) rather than in Alloy, an implementation level.

Finally, we plan to vary the temporal structure of our OCL extension in order to express temporal constraints based on other time structures such as linear structures and circular structures. We plan to generalize our infrastructure in order to allow the definition of time structures in a conceptual level, handling them at the model.

Bibliography

1. Mylopoulos, J.: Conceptual Modeling, Databases, and CASE: An Integrated View of Information Systems Development; chapter Conceptual Modeling and Telos; Wiley, Chichester (1992).
2. Guizzardi, G.: Ontological Foundations for Structural Conceptual Models. Telematica Instituut, The Netherlands (2005).
3. Halpin, T. and Morgan T.: Information modeling and relational databases. Morgan Kaufmann (2010).
4. OMG: UML Superstructure v2.4.1 (2012).
5. OMG: OCL Specification v2.4.1 (2014).
6. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Softw. Syst. Model. 4, 4, 386–398 (2005).
7. Brucker, A.D., Wolff, B.: HOL-OCL: a formal proof environment for UML/OCL. In: Fiadeiro, J.L. and Inverardi, P. (eds.) 11th International Conference on Fundamental Approaches to Software Engineering, FASE 2008. pp. 97–100 Springer Berlin Heidelberg (2008).
8. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Softw. Syst. Model. 9, 1, 69–86 (2010).
9. Cunha A., Garis A., Riesco D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. Softw. Syst. Model. (2013).
10. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL - Towards CTL-Extended OCL Model Checking. In MoDELS, 1092, pp. 13–22 (2013).
11. Bradfield, J.C., Filipe, J.K., Stevens, P.: Enriching OCL Using Observational Mu-Calculus. In FASE, 2306, 203–217 (2002).
12. Cabot, J., Olivé, A., Teniente, E.: Representing Temporal Information in UML. In UML'03, 2863, 44–59 (2003).
13. Conrad, S., and Turowski, K.: Temporal OCL Meeting Specification Demands for Business Components. In UML'01, 2185, 151–165 (2001).

14. Distefano, S., Katoen, J.P., Rensink, A.: On a temporal logic for object-based systems. In Fourth International Conference on Formal methods for open object-based distributed systems IV, 49, 305-325 (2000).
15. Flake, S., and Muller, W.: Formal semantics of static and temporal state-oriented ocl constraints. Software and System Modeling 2(3), 164–186 (2003).
16. Kanso, B., and Taha, S.: Specification of temporal properties with OCL. Science of Computer Programming 96, 527-551 (2014).
17. Mullins J. and Oarga R.: Model Checking of Extended OCL Constraints on UML Models in SOCLe. In FMOODS, 4468, 59–75 (2007).
18. Ziemann, P., and Gogolla, M.: OCL Extended with Temporal Logic. In 5th International Andrei Ershov Memorial Conference, PSI, 2890, 351-357 (2003).
19. Jackson, D.: Software Abstractions-Logic, Language, and Analysis, Revised Edition. The MIT Press (2012).
20. Olivé A., and Teniente, E.: Derived types and taxonomic constraints in conceptual modeling. Information Systems 27(6), 391–409 (2002).
21. Sider T.: Quantifiers and Temporal Ontology. Mind 115(457), 75-97 (2006).
22. Guerson, J., Almeida, J. P. A., Guizzardi, G.: Support for Domain Constraints in the Validation of Ontologically Well-Founded Conceptual Models. In 19th International Conference, EMMSAD, 302-316 (2014).
23. Sales T.P.: Ontology Validation for Managers, MSc Thesis, Federal University of Espírito Santo, UFES (2014).
24. Guizzardi G., Wagner G., Herre H.: On the foundations of UML as an ontology representation language. Engineering Knowledge in the Age of the Semantic Web, 47-62 (2004).
25. Cranefield, S., Purvis M.: UML as an ontology modelling language. Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Germany, University of Karlsruhe, 46-53, (1999).
26. Olivé, A.: Conceptual modeling of information systems. Springer Science & Business Media, (2007).

27. Guizzardi G., and Wagner G.: Using the unified foundational ontology (UFO) as a foundation for general conceptual modeling languages. Theory and Applications of Ontology: Computer Applications, 175-196 (2010).
28. Dwyer, M.B., Avrunin, G.S., Corbett J.C.: Patterns in property specifications for finite state verification. In Proceedings of the 21st International Conference on Software Programming, 411–420 (1999).
29. Maoz, S., Ringert, J., Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Whittle, J. et al. (eds.) 14th International Conference, MODELS 2011. pp. 592–607 Springer Berlin Heidelberg (2011).
30. Benevides, A.B., Guizzardi, G., Braga, B.F.B., Almeida, J.P.A.: Validating modal aspects of OntoUML conceptual models using automatically generated visual world structures. J. Univers. Comput. Sci. 16, 2904–2933 (2011).
31. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: France, R.B. et al. (eds.) 15th International Conference, MODELS 2012. pp. 415–431 Springer Berlin Heidelberg (2012).
32. Massoni, T., Gheyi, R., Borba, P.: Formal Refactoring for UML Class Diagrams. 19th Brazilian Symposium on Software Engineering (SBES). pp. 152–167 (2005).
33. Braga, B. F. B., Almeida, J. P. A., Guizzardi, G., Benevides, A. B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. Innov. Syst. Softw. Eng. 6, 1-2, 55–63 (2010).
34. OLED: OntoUML Lightweight Editor, https://code.google.com/p/ontouml-lightweight-editor (2015).
35. Carraretto, R.: A Modeling InfraStructure for OntoUML, BSc Thesis, Federal University of Espírito Santo, UFES (2010).
36. Albuquerque, A.: Ontological Foundations for Conceptual Modeling Datatypes, MSc Thesis, Federal University of Espírito Santo, UFES (2013).
37. Barcelos, P.P.F., Santos, V. A. dos, Silva, F.B., Monteiro M.E., Garcia A.S.: An Automated Transformation from OntoUML to OWL and SWRL (2013).

38. Zamborlini V.: Estudo de Mapeamento de Ontologias da Linguagem OntoUML para OWL, MSc Thesis, Federal University of Espírito Santo, UFES (2011).
39. Wiegers, R.: Behavior Specification for Ontologically Grounded Conceptual Models. MSc Thesis, University of Twente (2014).
40. Pereira D.C.: Representing Organizational Structures in Enterprise Architecture: an Ontology-based Approach. MSc Thesis, Federal University of Espírito Santo (UFES) (2015).
41. Nardi J.C.: Commitment-Based Reference Ontology for Service: Harmonizing Service Perspectives. PhD Thesis, Federal University of Espírito Santo (UFES) (2014).
42. Hughes, G. E.; Cresswell, M. J.: A Companion to Modal Logic. London: Routledge and Kegan Paul (1985).

Appendix A: Structural Layer of UFO

The Unified Foundational Ontology (UFO) [2] is a foundational ontology that provides a sound ontological basis to evaluate and give real-world semantics for conceptual modeling language's constructs such as those from UML. A foundational ontology is comprised by a set of categories (concepts) based on a number of theories from Formal Ontology, Philosophical Logics, Philosophy of Language, Linguistics and Cognitive Psychology. UFO synthesizes results from other foundational ontologies such as the Generalized Formalized Ontology (GFO), the top-level ontology underlying OntoClean and the foundational ontology DOLCE, solving a number of problematic issues regarding the coverage of these existing foundational ontologies in the development of ontological foundations for conceptual modeling languages such as UML, ORM and EER [27].

UFO consists of two compliance sets of concepts. The first deals with the ontological category of "endurants" (objects) (dubbed UFO-A) and the second with the categories of "perdurants" (events and processes) (termed UFO-B). Here, we will briefly present the first set i.e. a foundational set of concepts that persists in time called endurants and their relations.

Taxonomy of Endurant Types

Figure 32 depicts the taxonomy of endurant universals of UFO-A. Each of these types is defined by specific distinctions, which are based on various formal theories such as mereology, essentiality, identity criteria, dependences, rigidity, and inherence, among others. Our main objective here is to provide a full overview of the different entities that UFO-A categorizes.

A fundamental distinction in UFO-A is between the categories of "Particular" and "Universal". Particulars are entities that exist in reality possessing a unique identity, roughly speaking, one can think about instances. Universals, on the other hand, are patterns of features, which can be realized in a number of different particulars usually referred as concepts or types [27].

A "Monadic" universal is a universal that defines patterns of features to a single type of individual. Conversely, the "Relation" universal defines patterns of features to more than one type of individual. This reassembles the difference between concepts, such as classes, and relations, which relates two or more concepts.

Another important distinction is between "Substantial" and "Moment" universals. Moment individuals can be seen as objectified properties of other individuals, which inhere in those individuals. For example, the age of an individual John is a property of John. The same holds for John's headache, which inheres in John. This creates a chain of existential dependence between moments that terminates in a substantial individual, which does not inhere in any other individual. Therefore, a moment universal defines patterns of features to moment individuals (in which the inherence dependence holds).

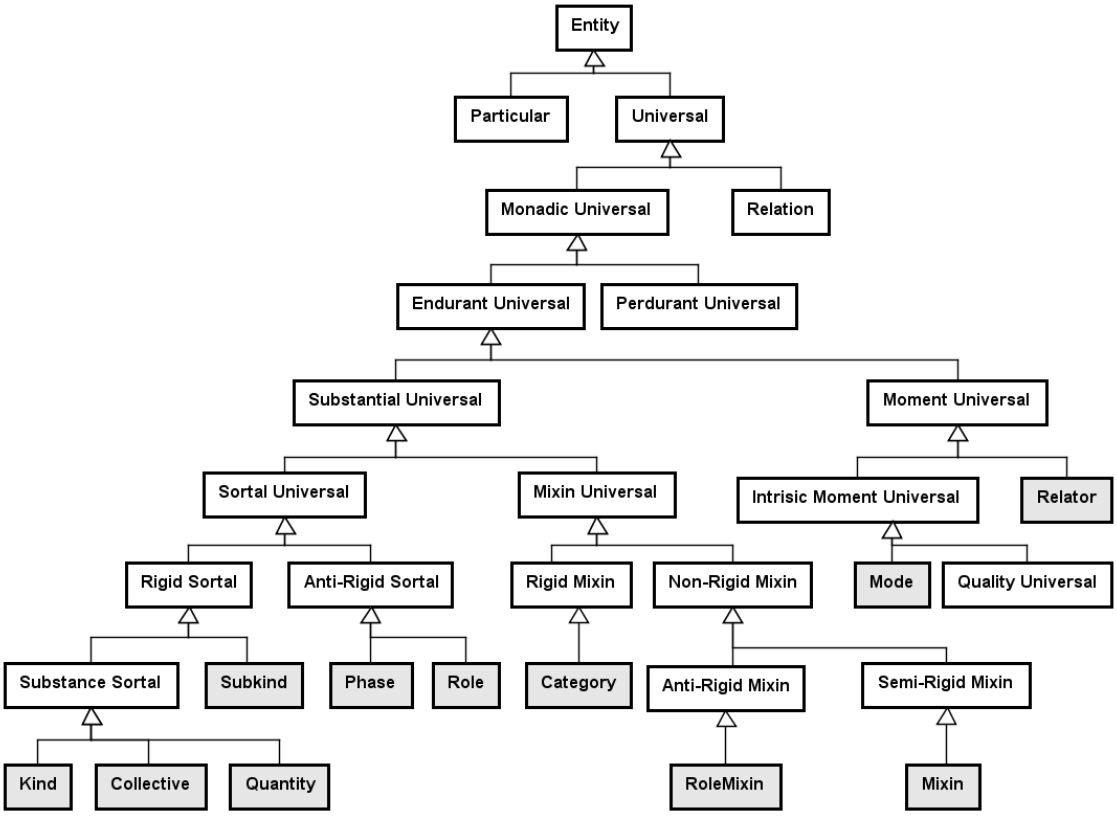

Figure 32 UFO-A Taxonomy of Endurant Types

"Sortal" universals define patterns of features to individuals with the same identity criteria and *Mixin* universals to those with different identities. "Rigid" universals in turn define that its instances will be of that specific type while they exist. For in-

stance, for an individual John to be instance of a rigid sortal universal named Person means that John will be a person while he exists. "Anti-Rigid" universals on the other hand define that there will be a time in which their instances are not of that type. For example, for John to be instance of an anti-rigid sortal universal named Student means that he will be a student in a time and cease to be student at another time. "Semi-Rigid" universals define that, part of its instances will be rigid instances, while others anti-rigid instances. For example, Seatable is a mixin universal because rigid individuals such as a chair are always seatable (considering that it cannot be broken) while a crate (which can be broken) is not i.e. there are times in which a crate is a solid crate and thus seatable and others in which it is broken and not seatable [2, p.113]. Moreover, a "Substance Sortal" universal, also referred as "Ultimate Sortal" universal, defines that its individuals pursues an identity criteria while Subkind universals only inherit the criteria from other substance sortal universals they must specialize.

"Intrinsic" moment universals such as a "Quality" defines that its instances are objectified properties that can be measured, for instance, the John's age or weight. Contrariwise, "Mode" universals define that its instances are objectified properties that cannot be measured such as for example John's headache. Finally, "Relator" universals define that their instances are a composition of objectified properties that inheres in more than one individual. For instance, the marriage between Abraham and Sarah is composed by certain externally dependent modes (intrinsic moments) of both Abraham and Sarah named qua-individuals. A qua-individual i.e. the individual qua-Abraham or qua-Sarah, exemplify all the properties that an individual has in the scope of a certain material relationship [2].

Taxonomy of Relational Types

Figure 33 depicts the taxonomy of relational universals of UFO-A. "Formal" relations hold between two or more entities without any further intervening individual. Conversely, "Material" relations require another individual to intervene in the relation called relators; these relators induce material relations.

"Basic" and "Comparative" Formal relations comprise formal relations. Basic Formal relations are types of existential dependence relations such as "Mediations", "Characterizations" and "Derivations" whilst Comparative Formal relations are founded in qualities, which are intrinsic to the relation's relata [2]. Characterizations define the inherence relation between intrinsic moments and their bearers (substantials), e.g., a Symptom characterizes what means to be a sick person. Mediations define the intermediation between a relator and its inhered individuals e.g. a marriage mediates both Abraham and Sarah. Derivations define the relator from which the material relation it is induced by e.g. the material relation "is married with" between Abraham and Sarah is derived by their particular marriage.

Finally, a "Meronymic" relation is a part-whole relation and is comprised by four types of relations. "MemberOf" are relations that hold between functional complexes [2] and collectives e.g. John is a member of a Band. "SubQuantityOf" relations hold between quantities e.g. alcohol composes wine. "SubCollectionOf" relations hold between collectives e.g. the collection of male individuals in a crowd is part of that crowd. "ComponentOf" relation holds between functional complexes e.g. a heart is part of a person.

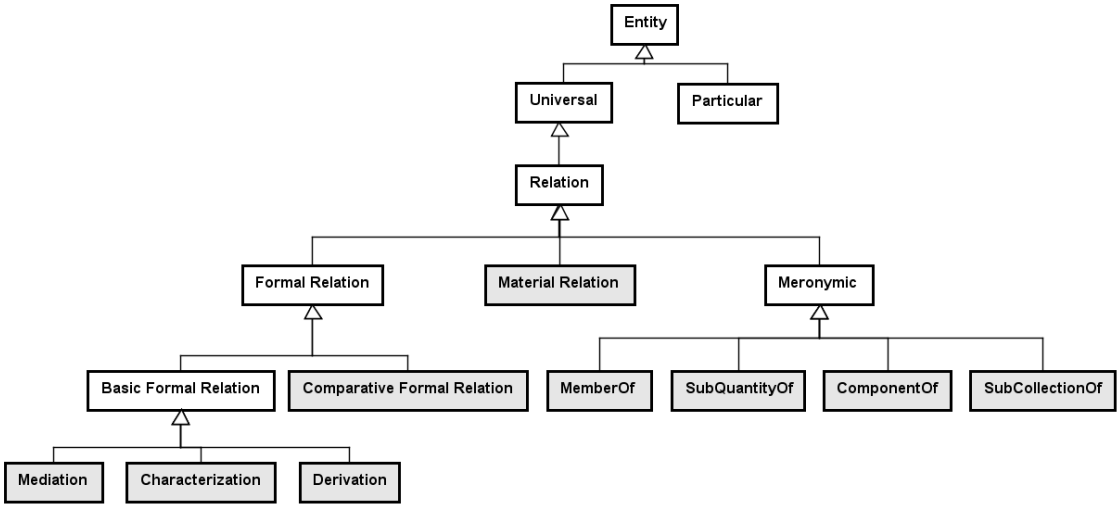

Figure 33 UFO-A Taxonomy of Relational Types

In the sequel, we formally characterize the dynamics already captured by these ontological categories, such as rigidity, anti-rigidity, non-rigidity, semi-rigidity, existential and specific dependences, essentiality, inseparability and finally, immutability.

Rigidity

Rigidity is one of the main meta-properties of UFO and distinguishes different types of endurants. There are endurant types that are rigid, non-rigid, anti-rigid, and semi-rigid. Sortal universals such as a Kind, Quantity, Collective and Subkind, Moment universals such as a Mode, Quality and Relator, and Mixin universals such as a Category, are all types of rigid universals.

A rigid universal defines that all of its individuals will continue to be so as long as they exist [2, p. 42, 101]. In other words, a universal G is rigid iff, for all G's individuals (individuals belonging to the extension of G), if they exist in a world *w*, then they must belong to G's extension in that world, as formalized in Axiom 4.

Axiom 4 Rigidity

$$\text{rigid}(G) \stackrel{\text{def}}{=} \forall x \in \text{ext}(G), \forall w \in W, \text{existsIn}(x, w) \rightarrow x \in \text{ext}(G, w)$$

Non-rigidity on the other hand states that at least one of its individuals will not continue to be so [2, p.101]. In other words, a universal G is non-rigid iff, for some of G's individuals (individuals belonging to the extension of G), there will be at least one world *w* in which they exist but do not belong to G's extension, as formalized in Axiom 5.

Axiom 5 Non-Rigidity

$$\text{nonrigid}(G) \stackrel{\text{def}}{=} \exists x \in \text{ext}(G), \exists w \in W, x \notin \text{ext}(G, w) \text{and existsIn}(x, w)$$

Non-rigidity is divided in two types: Anti-Rigidity and Semi-Rigidity. An anti-rigid universal defines that for all its individuals there will be a world in which they do not continue to be so [2, p.102]. Sortal universals such as a Role and Phase, and Mixin universals such as Role-Mixin are all types of anti-rigid universals. In other words, a universal G is anti-rigid iff, for all G's individuals (individuals belonging to the extension of G), there will be at least one world *w* in which they exist but do not belong to G's extension, as formalized in Axiom 6. Notice that anti-rigidity is a specific case of non-rigidity. In other words, non-rigidity constitutes a weaker constraint than what is imposed by anti-rigidity [2, p.102].

Axiom 6 Anti-Rigidity

$$\text{antirigid}(G) \stackrel{\text{def}}{=} \forall x \in \text{ext}(G), \exists w \in W, x \notin \text{ext}(G, w) \text{and existsIn}(x, w)$$

Mixins in turn are the only semi-rigid universals. A universal is semi-rigid iff it is non-rigid but not anti-rigid [2, p.102] as formalized in Axiom 7. Therefore, a universal G is semi-rigid iff

Axiom 7 Semi-Rigidity

$$\text{semirigid}(G) \stackrel{\text{def}}{=} \text{nonrigid}(G) \text{and not antirigid}(G)$$

Dependences

Dependences are another main meta-property of UFO and distinguish different types of relationships. Dependence, in its more general form, is a relationship that holds between two universals X and Y stating that necessarily, whenever *x* (individual of X) exists, *y* (individual of Y) must also exist, without specifying which times each entity must exist. In other words, if *x* exists at some world state, then *y* exists at some world state. The world which *y* must exist may be prior to, coincident with or even subsequent to the world state in which *x* exists. Here we name of instantaneous dependence the type of dependence that holds in a coincident world state, i.e., for any world state *w* at which *x* exists, *y* must also exist in *w*. For the sake of simplicity, from this point forward the term instantaneous dependence is only referred as dependence.

A dependency is classified into "rigid specific" or "specific" dependence. A rigid specific dependence states that if an individual *x* exists at world state w, then *y* must also exist in w i.e. Existential Dependence as formalized in Axiom 8. A Specific Dependence on the other hand states that if an individual *x* exists as an instance of a universal G at a world w, then y must also exist in w, as formalized in Axiom 9.

Axiom 8 Existential Dependence

$$\text{ed}(x, y, w) \stackrel{\text{def}}{=} \text{existsIn}(x, w) \rightarrow \text{existsIn}(y, w)$$

Axiom 9 Specific Dependence

$$\mathrm{sd}(x, y, w) \stackrel{\text{def}}{=} x \in \mathrm{ext}(G, w) \rightarrow \mathrm{existsIn}(y, w)$$

Let the predicate partOf(y, x, w) denote the parthood relation in which an individual *y* is part of an individual *x*. A parthood relation between individuals y and x holds at world w if both y and x exist at w and are connected through a parthood relation.

An Essential Part means that a whole is existentially dependent on its part i.e. whenever the whole exists at a world, its part must also exist at that world. For instance, a person only exists in a world if a brain also exists at that world and was related to that person via parthood relation, which means that this very brain is essential to the existence of that person. Formally, we can state essentiality of parts as in Axiom 10.

Axiom 10 Essential Part

$$\mathrm{esp}(y, x) \stackrel{\text{def}}{=} \forall w \in W, \mathrm{existsIn}(x, w) \rightarrow \mathrm{existsIn}(y, w) \text{ and } \mathrm{partOf}(y, x, w)$$

An Inseparable Whole means that it is the part that is existentially dependent on the whole i.e. whenever the part exists at a world, its whole must also exists at that world. For example, a brain only exists at a world if a person also exists at that world and is related to that brain via parthood relation, which means that this very person is inseparable from that brain, as formalized in Axiom 11. Essentiality and inseparability are all types of existential dependence. An essential part is always a rigid universal as well as an inseparable whole [2].

Axiom 11 Inseparable Whole

$$\mathrm{isw}(y, x) \stackrel{\text{def}}{=} \forall w \in W, \mathrm{existsIn}(y, w) \rightarrow \mathrm{existsIn}(x, w) \text{ and } \mathrm{partOf}(y, x, w)$$

Essential parts and inseparable wholes must be rigid universals [2] but in the case where they are anti-rigid universals, the parts are called immutable parts and the wholes immutable wholes, respectively.

An Immutable Part means that the whole is specifically dependent on the part i.e. whenever the whole instantiate the anti-rigid universal G at a world, its part must

also exist at that world as formalized by Axiom 12. If the whole ceases to instantiate G, then the dependence no longer holds.

Axiom 12 Immutable Part

$$\text{imp}(y, x) \stackrel{\text{def}}{\equiv} \forall w \in W, x \in \text{ext}(G, w) \rightarrow \text{existsIn}(y, w) \text{ and } \text{partOf}(y, x, w)$$

An Immutable Whole on the other hand is just the other way around, meaning that the part that is specifically dependent on the whole i.e. whenever the part instantiate the anti-rigid universal G at a world, the whole must also exist at that world as formalized by Axiom 13. Immutability of parts and wholes are types of specific dependence. Thus, an immutable part is always an anti-rigid universal as well as an immutable whole [2].

Axiom 13 Immutable Whole

$$\text{imw}(x, y) \stackrel{\text{def}}{\equiv} \forall w \in W, y \in \text{ext}(G, w) \rightarrow \text{existsIn}(x, w) \text{ and } \text{partOf}(y, x, w)$$

Immutability

From an equivalent logic point of view, essentiality and inseparability state that an individual *x*, which is a part or a whole, will always be connected to the same whole or part, respectively, at any time (world state) that it exists. This dynamic aspect is called here of Immutability and in UML is defined using the readOnly UML meta-property. The readOnly meta-property defines that a UML an association end-point (or attribute) cannot be updated once assigned. This means that their values cannot change.

> "isReadOnly: Boolean - if true, the attribute may only be read, and not written. The default value is false." "If a navigable property is marked as read-only, then it cannot be updated once it has been assigned an initial value." [4, p.125, 129]

Essentiality and inseparability should thus imply that the respective ends of the parthood relations are readOnly by default. In other words, if a part is essential, this means that the property that leads the whole to its part is immutable i.e. readOnly (a whole cannot change its part), and if the whole is inseparable, this means that the

property that leads the part to its whole is readOnly (the part cannot change its whole).

Analogously, immutable parts and wholes also state immutability. This means that an individual *x*, which is a part or a whole, will always be connected to the same whole or part, respectively, while instantiating the anti-rigid universal. In this manner, the property that leads the whole to its immutable part must also be set as readOnly and the property, which leads the part to its immutable whole must also be readOnly by default.

Therefore, let *x* be an individual which is instance of the rigid universal G and the expression "x.P(w)" be a property of the individual *x* at world *w*, we can formalize immutability as formalized in Axiom 14.

Axiom 14 Immutability

$$\text{immutable}(P, G) \stackrel{\text{def}}{=} \forall w \in W, \forall x \in \text{ext}(G, w),$$

$$\forall w' \in W, x \in \text{ext}(G, w') \rightarrow x.P(w') = x.P(w)$$

Lastly, dependency binary relationships in UFO are Characterizations, Mediations and Derivations relationships, and all of them stand for relations of existential dependence. This means that they also define immutability on their target-end point. Characterizations are always readOnly by default on the characterized side and mediations on the mediated side [2, p.334-336]. Moreover, derivation relationships between relators and material relationships are also readOnly on the material relationship side [2, p.337]. The semantics applied to the UML readOnly meta-property will depend on the rigidity of the universal from/of the property.

Appendix B: Constraints to the Reified Model

Here we list all constraints (of our running example) that are (automatically) added to the world-reified plain UML model of background so that the OntoUML model semantics is preserved in the reification step. Listing 23 exemplifies the first set of constraints enforcing actual multiplicity constraints using plain OCL on the world-reified background model. It specifies actual multiplicity cardinalities of the original OntoUML mediation relationship between relator Marriage and role Wife on the word-reified UML model.

```
context World
inv marriage_mediates_one_wife_at_a_time:
    self.endurant->select(i | i.oclIsKindOf(Marriage))->forAll(m |
    m.mediates_marriage_wife->select(r | r.world = self)->size() = 1)
inv wife_is_mediated_by_one_marriage_at_a_time:
    self.endurant->select(i| i.oclIsKindOf(Wife))->forAll(h |
    h.mediates_marriage_wife->select(r | r.world = self)->size() = 1)
```

Listing 23 World Reified Model: Current Multiplicity Cardinalities

The first OCL constraint states that for every world (self), for all marriages at that world, the number of "mediates_Marriage_Wife" linked (at that world) to that marriage is equal to one. Conversely, the second OCL constraint states that a wife is mediated by exactly one marriage at a particular world. The same pattern is applied to the other mediation between Marriage and Husband, in fact, to every OntoUML relationship, with the exception of the OntoUML derivation relationships, which we assume to be non-navigable.

The next set of additional constraints capture the fact that relationships, relators and relata co-exist in all worlds in which they exist. In other words, a constraint ensuring the cycle between the reified relationship (e.g. mediates_Marriage_Wife), the elements they connect (e.g. Wife) and the worlds in which they exist (e.g. World), such as described in Listing 24. The same pattern is applied to the other mediation

between Marriage and Husband (i.e. to every OntoUML relationship except for OntoUML derivations as they are not translated).

```
context mediates_Marriage_Wife
inv wife_exists_same_world_as_mediation:
self.world.endurant->select(i |i.oclIsKindOf(Wife))->includes(self.wife)
inv marriage_exists_same_world_as_mediation:
self.world.endurant->select(i |i.oclIsKindOf(Marriage))->includes(self.marriage)
```

Listing 24 World Reified Model: Existence Cycles

The world-reified model also needs constraints to ensure the immutability on the mediated side from the OntoUML mediation, as described in Listing 25. For example, the first OCL invariant states that for every world *self*, for every marriage at that world, for every subsequent world *n*, the wife related to that marriage in *n* is the same as in *self*. Analogously, the same holds for the immutability of husbands.

```
context World
inv immutable_wife:
self.endurant->select(I | i.oclIsKindOf(Marriage))->forAll(m |
self->asSet()->closure(next)->asSet()->forAll(n |
m.oclAsType(Marriage).mediates_marriage_wife->select(r | r.world = n).wife =
m.oclAsType(Marriage).mediates_marriage_wife->select(r | r.world = self).wife))
inv immutable_husband:
self.endurant->select(I | i.oclIsKindOf(Marriage))->forAll(m |
self->asSet()->closure(next)->asSet()->forAll(n |
m.oclAsType(Marriage).mediates_marriage_husband->select(r|r.world=n).husband =
m.oclAsType(Marriage).mediates_marriage_husband->select(r|r.world=self).husband))
```

Listing 25 World Reified Model: Immutability of Relata

Finally, the world-reified model needs to reflect the Set type as the default collection type of original OntoUML relationships. This means that by default, in mediations, no two relations (at an instance level) are allowable between the same instances.

Therefore we need to ensure that no two reified mediations have the same world, domain and range elements such as described in Listing 26.

```
context World
inv no_duplicated_mediations_between_marriage_and_wife:
    not self.mediates_Marriage_Wife->exists(m1, m2 |
    m1<>m2 and m1.marriage = m2.marriage and m1.wife = m2.wife)
inv no_duplicated_mediations_between_marriage_and_husband:
    not self.mediates_Marriage_Wife->exists(m1, m2 |
    m1<>m2 and m1.marriage = m2.marriage and m1.husband = m2.husband)
```

Listing 26 World Reified Model: Mediation's Set Type

Lastly, although we only demonstrated the world reification of mediation relationships, the same holds for all the other OntoUML relationships. We were restricted to our running example which only used two mediations. The only exception regards OntoUML material relationships, which may have duplicates at an instance level (the default collection type are Bag types). For this reason, uniqueness constraints are not applied to material relationships.

Appendix C: Alloy Language and Analysis

Alloy [19] is a declarative and first-order logic based language to describe and explore structures. Alloy models (often called *specifications*) can be viewed as a set of constraints (axioms) that describe (implicitly) a set of structures. A model is comprised basically by declaration of signatures, relations, facts, predicates, assertions and functions. The Alloy tool supports a solver responsible for finding structures that satisfy the specification i.e. the constraints implied by the Alloy specification. The Alloy analyzer (how the tool is called) uses predicates to explore the model, generating sample structures in conformity with the model, and assertions to check properties of the model, generating counter-examples of it. The result is displayed graphically to the user. Alloy structures are composed by atoms and the relations between these atoms:

Atoms and Relations

An atom is a primitive entity which is indivisible (it cannot be broken into little pieces), immutable (its properties does not change with time), and non-interpretable (it does not have built-in properties such as the numbers). Few things in reality are atomics, therefore, in order to create structures that are divisible, mutable and interpretable; relations are introduced. A relation is a structure that relates atoms. It consists of a set of tuples, where each tuple is a sequence of atoms. For example, in Figure 34 we have the atoms A2 e B4 and a relation between them, i.e. the tuple (A2, B4). The relation "r" is therefore the set of tuples r = {(A2, B4), (A1, B4), (A3, B1), (A3, B2), (A3, B3)}.

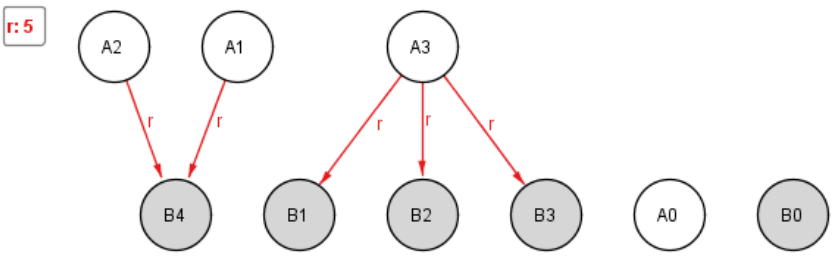

Figure 34 Alloy Atoms and Relations

Signatures and Fields

A set of atoms is declared through a signature declaration. For instance, the declaration "sig A {}" declares a set of atoms named "A". A signature is more than just a set of atoms because it can include relations declarations. The relations are declared as fields in the signatures. For example, the declaration "sig A {r: set B}" introduces a relations "r" whose domain is the set "A" and range the set "B", as showed in previous Figure 34.

Facts, Functions, Predicates and Assertions

Alloy structures are accompanied of properties i.e. restrictions and assertions about the structure itself. These properties are organized into paragraphs. The paragraphs can contain four types of properties: facts, functions, predicates and assertions.

Facts specify restrictions that must always hold in any circumstance. For instance, in Figure 34, we could specify a restriction stating that the set of atoms "A" cannot be an empty set. In this manner, at any possible instantiation (structure) generated according to the declarations in the model, the set "A" will always be a set with the minimum 1 atom. In Alloy, this can be represented as "fact {some A}".

Functions specify expressions to be reused in different parts of the model (similarly to the methods/functions in programming languages). For example, in Figure 34 we could specify a function named "getBs" that given an atom from the set "A" i.e. A1, it would return all atoms from the set "B" related to A1 through relation "r". In Alloy, this would be represented as "fun getBs [x: A]: set B {x.r}".

Predicates specify restrictions that can be used in different parts of the model. For instance, in Figure 34 we could specify a predicate named "only2Bs" defining that an atom of the set "A" (e.g. A1) is related to exactly two atoms of the set "B" through relation "r". This would be represent in Alloy as "pred only2Bs [x: A] {# x.B = 2}". We could thus use this predicate to state that all atoms of the set "A" satisfy this restriction i.e. using this predicate inside a fact such as "fact {all x: A | only2Bs[x]}".

Assertions specify properties that we desire to be valid from the facts of the model. The Analyzer can then check the assertions. If an assertion is invalid from the facts, or a failure of the project was exposed or there have been an error from the formulation of the assertion. For instance, we could specify an assertion named "notEmptyBs" to state that the set "B" will always be a non-empty set, such as "assert notEmptyBs {some B}". If, according to the facts of the model, the analyzer finds an example in which this assertion does not hold, the tool will show this case as a counter-example. Otherwise, the assertion can be valid, or invalid.

Commands and Scopes

An Alloy model is comprised by basically sets and relations (forming the structure of the model) and facts, predicates, assertions and functions (specifying the (un-) desired properties of the model). The tool then searches for (counter-) examples through Alloy commands. In particular, using the Alloy "run" command, specified over predicates, and the Alloy "check" command, specified over assertions.

In case of *running simulation* (i.e. using the run command), the tool search for examples that should be in conformity with the predicate inside the run command and all the facts from the model. An example is a scenario in which both, facts and that predicate are valid. For example, the execution of the predicate "only2Bs" would be defined as "run {only2Bs} for 5", where the number 5 specifies the scope of the command i.e. all top-level signatures will have at most 5 atoms.

In case of *checking an assertion*, the analysis considers the negation of the assertion being checked, and all the facts from the model. A counter-example is a scenario in which the assertion fails from the facts of the model. For instance, the checking of the assertion "notEmptyBs" would be defined as "check notEmpty2Bs for 5". Through a scope definition i.e. the maximum number of atoms of each top-level signature of the model, the tool provides a possible instantiation, in visual form, that satisfies the model.

Simulation and Analysis

It is impossible to guarantee where an assertion is valid since it requires to cover the entire space of solution. Instead, the analysis in Alloy is based on the instance finding, which is an attempt to find a refutation checking an assertion against a huge number of test cases (a tiny model with only four relations in Alloy can have a space of solution over billions of test cases) [19, p.141, 142]. The analysis executed by the Alloy tool is based on the SAT (boolean satisfiability) technology. The analyzer translates the Alloy restrictions to boolean restrictions which are given to an efficient SAT solver. This solver can examine spaces over a hundred of bits (i.e., 10^60 cases or more) [19, Preface]. The analysis of an assertion finishes when the first instance is found. If none instance is found, it is still possible that an instance exists in a greater test case than considered [19, p.141, 142]. The size of the test case can be increase by changing the value of the scope. A scope determines the maximum number of atoms of each top-level signature of the model. [19, p.130]. It is true that the assertion is checked against a finite number of test cases that occupies only a small portion of all the possible space of cases. However, the analysis tends to be more effective to find specification problems. The search for an instance that satisfies the invalid assertion is realized exhaustively inside the tiny set of test cases defined by the scope. The small scope hypothesis states that the majority of problems in specifications have tiny counter-examples, i.e., if an assertion is invalid, then it has probably a counter-example with a small scope among all the test cases considered [19, p.143].