Building Correct Taxonomies with a Well-Founded Graph Grammar

Jeferson O. Batista[1], João Paulo A. Almeida[1],
Eduardo Zambon[1], and Giancarlo Guizzardi[1,2]

[1] Federal University of Espírito Santo, Vitória, Brazil
[2] Free University of Bozen-Bolzano, Italy
jeferson.batista@aluno.ufes.br, jpalmeida@ieee.org,
zambon@inf.ufes.br, gguizzardi@unibz.it

Abstract. Taxonomies play a central role in conceptual domain modeling having a direct impact in areas such as knowledge representation, ontology engineering, software engineering, as well as in knowledge organization in information sciences. Despite their key role, there is in the literature little guidance on how to build high-quality taxonomies, with notable exceptions such as the OntoClean methodology, and the ontology-driven conceptual modeling language OntoUML. These techniques take into account the ontological meta-properties of types to establish well-founded rules for forming taxonomic structures. In this paper, we show how to leverage on the formal rules underlying these techniques to build taxonomies which are *correct by construction*. We define a set of correctness-preserving operations to systematically introduce types and subtyping relations into taxonomic structures. To validate our proposal, we formalize these operations as a graph grammar. Moreover, to demonstrate our claim of correctness by construction, we use automatic verification techniques over the grammar language to show that: (i) all taxonomies produced by the grammar rules are correct; and (ii) the rules can generate all correct taxonomies.

Keywords: Conceptual modeling · Taxonomies · Graph grammar

1 Introduction

Taxonomies are structures connecting types via *subtyping* (i.e., type specialization) relations. They are fundamental for conceptual domain modeling and have a central organizing role in areas such as knowledge representation, ontology engineering, object-oriented modeling, as well as in knowledge organization in information sciences (e.g., in the construction of vocabularies and other lexical resources). Despite their key role in all these areas, there is in the literature little guidance on how to build high-quality taxonomies.

A notable exception is OntoClean [4]. OntoClean was a pioneering methodology that provided a number of guidelines for diagnosing and repairing taxonomic relations that were inconsistent from an ontological point of view. These guidelines were grounded on a number of *formal meta-properties*, i.e., properties characterizing types. Derived from these meta-properties, the methodology would offer a number of formal rules governing

how types characterized by different meta-properties could be associated to each other in well-formed taxonomies.

OntoClean has been sucessfully employed to evaluate and suggest repairs to several important resources (e.g., WordNet [13]). However, being a methodology, it does not offer a representation mechanism for building taxonomies according to its prescribed rules. Also with the intention of addressing that problem, in [8], the authors (including one of OntoClean's original authors) proposed a UML profile with modeling distinctions based on an extension of OntoClean's meta-properties and rules. That profile would later become the basis of the OntoUML modeling language [5], incorporating syntactic rules to prevent the construction of incorrect taxonomies in conceptual models. In [6], the language has its full formal semantics defined in terms of a (proved-consistent) ontological theory, and its abstract syntax defined in terms of a metamodel. In particular, the latter is an extension of the UML 2.0 metamodel, redesigned to reflect the ontological distinctions and axiomatization put forth by that theory. These distinctions and constraints, in turn, have influenced other prominent modeling languages, e.g., ORM [9].

As argued in [15], instead of leveraging on this axiomatization by proposing methodological rules (as in OntoClean) or semantically-motivated syntactical constraints (as in the OntoUML metamodel), a representation system based on this ontological theory could employ a more productive strategy. It could leverage on that fact that the theory's formal constraints impose a correspondence between each particular type of type (characterized by those ontological meta-properties) and certain modeling *structures* (or modeling *patterns*). In other words, a representation system grounded on this ontological theory is a *pattern language*, i.e., a system whose basic building blocks are not low-granularity primitives such as types and relations but higher-granularity patterns formed by types and relations. A MOF-based metamodel (such as UML's) simply isn't capable of naturally capturing this fundamental aspect of such a representation system.

In [17], some of us have proposed a first attempt to formalize a representation system based on that ontological theory as a true Pattern Grammar, i.e., as a Graph Grammar with transformation rules capturing these patterns and their possible relations. Hence, this paper can be seen as an extension of that work. On one hand, it is focused of types and taxonomic relations. On the other hand, it extends that original work in providing a complete set of graph transformation rules. Moreover, we use automatic verification techniques over the grammar state space (language) to show the correctness of the taxonomies produced by the grammar and the capability of the grammar to generate all correct taxonomies.

This work contributes to the foundations of rigorous conceptual modeling by identifying the set of rules that should be considered as primitives in the design of correct taxonomies. Moreover, it does that in a metamodel-independent way, so the results presented here can be incorporated into different modeling languages (again, e.g., ORM) as well as different tools used by different communities (e.g., as a modeling plugin to Semantic Web tools such as Protégé[3]).

The remainder of this paper is structured as follows. In Section 2, we review the ontological foundations used in this work. In particular, we present a number of ontological meta-properties, a typology of types derived from them, and the formal constraints

[3] `https://protege.stanford.edu/`

governing the subtyping relations between these types. In Section 3, we present the graph transformation grammar with operations that take into account the distinctions and constraints discussed in Section 2. In Section 4, we discuss the formal verification of the grammar. Finally, Section 5 presents some concluding remarks.

2 Ontological Foundations

In this section, we present some ontological distinctions that are the basis for the remainder of this paper. These notions and the constraints governing their definitions and relations correspond to a fragment of the foundational ontology underlying OntoUML, and which incorporates and extends the theory of types underling OntoClean [5, 7]. For an in depth discussion, philosophical justification, empirical support, and full formal characterization of these notions, one should refer to [5, 6].

Types represent properties that are shared by a set of possible instances. The set of properties shared by those instances is termed the *intension* of type; the set of instances that share those properties (i.e., the instances of that type) is termed the *extension* of that type. Types can change their extension across different circumstances, either because things come in and out of existence, or because things can acquire and lose some of those properties captured in the intension of that type.

Taxonomic structures capture *subtyping* relations among types, both from *intensional* and *extensional* points of view. In other words, subtyping is thus a relation between types that govern the relation between the possible instances of those types. So, if type B is a subtype of A then we have that: (i) it is necessarily the case that all instances of B are instances of A, i.e., in all possible circumstances, the extension of B subsets the extension of A; and (ii) all properties captured by the *intension* of A are included in the *intension* of type B, *i.e.*, B's are A's and, therefore, B's have all properties that are properties defined for type A.

Suppose all instances that exist in a domain of interest are *endurants* [6]. Endurants roughly correspond to what we call *objects* in ordinary language, i.e., things that (in contrast to occurrences, events) endure in time changing their properties while maintaining their identity. Examples include you, each author of this paper, Mick Jagger, the Moon, the Federal University of Espírito Santo.

Every endurant in our domain belongs to one **Kind**. In other words, central to any domain of interest we will have a number of object kinds, i.e., the genuine fundamental types of objects that exist in that domain. The term "kind" is meant here in a strong technical sense, i.e., by a kind, we mean a type capturing *essential* properties of the things it classifies. In other words, the objects classified by that kind could not possibly exist without being of that specific kind [6].

Kinds tessellate the possible space of objects in that domain, i.e., all objects belong to exactly one kind and do so necessarily. Typical examples of kinds include Person, Organization, and Car. We can, however, have other static subdivisions (or subtypes) of a kind. These are naturally termed **Subkinds**. As an example, the kind 'Person' can be specialized in the (biological) subkinds 'Man' and 'Woman'.

Endurant kinds and subkinds represent essential properties of objects. They are examples of **Rigid Types** [6]). Rigid types are those types that classify their instances

necessarily, i.e., their instances must instantiate them in every possible circumstance in which they exist. We have, however, types that represent *contingent* or *accidental* properties of endurants termed **Anti-Rigid Types** [6]). For example, in the way that 'being a living person' captures a cluster of contingent properties of a person, that 'being a puppy' captures a cluster of contingent properties of a dog, or that 'being a husband' captures a cluster of contingent properties of a man participating in a marriage.

Kinds, subkinds, and the anti-rigid types specializing them are categories of endurant **Sortals**. In the philosophical literature, a sortal is a type that provides a uniform principle of identity, persistence, and individuation for its instances [6]. To put it simply, a sortal is either a kind (e.g., 'Person') or a specialization of a kind (e.g., 'Student', 'Teenager', 'Woman'), i.e., it is either a type representing the essence of what things are or a subclassification applied to the entities that "have that same type of essence", be it rigid, i.e., a **Subkind**, or anti-rigid, i.e., an **Anti-Rigid Sortal**.

In general, types that represent properties shared by entities of *multiple kinds* are termed **Non-Sortals**, i.e., non-sortals are types whose extension possibly intersect with the extension of more than one kind. Non-sortals too can also be further classified depending on whether the properties captured in their intension are essential (i.e., rigid) properties or not.

Now, before we proceed, we should notice that the logical negation of rigidity is not anti-rigidity but *non-rigidity*. If being rigid for a type A means that all instances of A are necessarily instances of A, the negation of that (i.e., non-rigidity) is that there is at least one instance of A that can cease to be an instance of A; anti-rigidity is much stronger than that, it means that all instances of A can cease to be instances of A, i.e., A's intension describes properties that are contingent for all its instances. Finally, we call a type A *semi-rigid* iff it is non-rigid but not anti-rigid, i.e., if it describes properties that are essential to some of its instances but contingent to some other instances. Because non-sortal types are dispersive [11], i.e., they represent properties that behave in very different ways with respect to instances of different kinds, among non-sortal types, we have: those that describe properties that apply *necessarily to the instances of all kinds* it classifies (i.e., Rigid Non-Sortals, which are termed **Categories**); those that describe properties that apply *contingently to the instances of all kinds* it classifies (**Anti-Rigid Non-Sortals**); those that describe properties that apply *necessarily to the instances of some of the kinds it classifies* but that also apply *contingently to the instances of some other kinds it classifies* (i.e., Semi-Rigid Non-Sortals, termed **Mixins**). An example of a category is 'Physical Object' representing properties of all kinds of entities that have masses and spatial extensions (e.g., people, cars, watches, building); an example of a anti-rigid non-sortal is 'Customer' representing contingent properties for all its instances (i.e., no customer is necessarily a customer), which can be of the kinds 'Person' and 'Organization'; an example of a mixin is the 'Insurable Item', which describe properties that are essential to entities of given kinds (e.g., suppose that cars are necessarily insured) but which are contingent to things of other kinds (e.g., houses can be insured but they are not necessarily insured).

Figure 1 represents this typology of endurant types generated by the possible values of these two properties. As always, UML arrows connect subtypes to their supertypes (the arrowhead pointing to the supertype). Two subtyping relations joined in their ar-

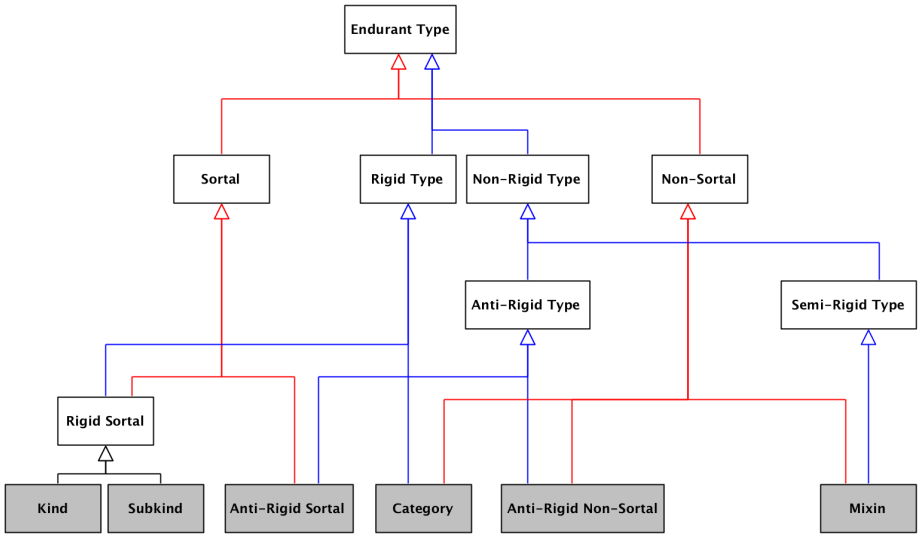


Fig. 1. A taxonomy for Endurant Types.

rowheads form a *generalization set*, which here we assume to tessellate the extension of the supertype (pointed to by the joint arrowhead), i.e., these are disjoint and complete generalization sets. The red subtyping relations and generalization sets here represent an inheritance line created by the *sortality* meta-property, i.e., all endurant types are either sortals (i.e., either kinds or specializations thereof) or non-sortals (crossing the boundaries of multiple kinds) but not both. Finally, the blue subtyping relations and generalization sets here represent an inheritance line created by the *rigidity* meta-property, i.e., all endurant types are either rigid (i.e., essentially classifying all their instances), anti-rigid (i.e., contingently classifying all their instances), or semi-rigid (essentially classifying some of their instances, and contingently classifying others). As a result of the combination of these two meta-properties, we have the following six (exhausting and mutually disjoint) types of types (i.e., meta-types): **Kinds**, **Subkinds**, **Anti-Rigid Sortals**, **Categories**, **Anti-Rigid Non-Sortals**, and **Mixins** (in grey in Figure 1).

The ontological meta-properties that characterized these different types of types also impose constraints on how they can be combined to form taxonomic structures [6]. As we have already seen, since kinds tessellate our domain and, because all sortals are either kinds or specializations thereof, we have both that: no kinds can specialize another kind; every sortal that is not a kind specializes a unique kind. In other words, every sortal hierarchy has a unique kind at the top. Moreover, from these, we have that any type that is a supertype of a kind must be a non-sortal. But also that, given that every specialization of a kind is a sortal, non-sortals cannot specialize sortals. Finally, given the formal definitions of rigidity (including anti-rigidity), it just follows logically that anti-rigid types (sortals or not) cannot be supertypes of semi-rigid and rigid types (sortals or not) (see proof in [6]). For example, if we determine that ‘Customer’ applies contingently

to persons in the scope of business relationships, then a taxonomy in which a rigid type 'Person' specializes an anti-rigid type 'Customer' is logically incorrect. Intuitively, a person will be at the same time required through the specialization to be statically classified as a 'Customer' while at the same time, being defined dynamically classified as a 'Customer', in virtue of the definition of that type. So, either: (i) the definition of 'Customer' should be revised to capture only essential properties, becoming a rigid type and thus solving the incorrect specialization problem; or (ii) 'Customer' should be an anti-rigid specialization of the rigid type 'Person', inverting the direction of the original (but incorrect) taxonomic relation.

3 Graph Transformation Rules to Build Taxonomies

Graph transformation (or *graph rewriting*) [10] has been advocated as a flexible formalism, suitable for modeling systems with dynamic configurations or states. This flexibility is achieved by the fact that the underlying data structure, that of graphs, is capable of capturing a broad variation of systems. Some areas where graph transformation is being applied include visual modeling of systems, the formal specification of model transformations, and the definition of graph languages, to name a few [3, 16].

The core concept of graph transformation is the rule-based modification of graphs, where each application of a rule leads to a graph transformation step. A transformation rule specifies both the necessary preconditions for its application and the rule effect (modifications) on a *host graph*. The modified graph produced by a rule application is the result of the transformation.

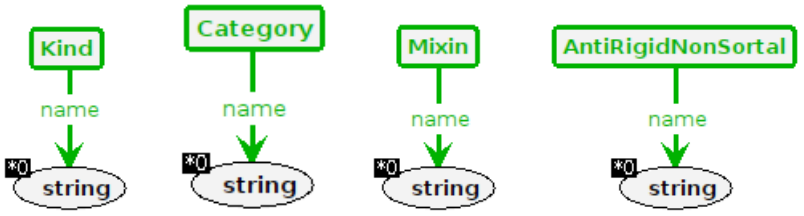
In this work, we use graph transformations to formally model the operations for the construction of a taxonomy. A set of graph transformation rules can be seen as a declarative specification of how the construction can evolve from an initial state, represented by an initial (empty) host graph. This combination of a rule set plus an initial graph is called a *graph grammar*, and the (possibly infinite) set of graphs reachable from the initial graph constitute the *grammar language*.

Our main contribution in this paper is to formally define a graph grammar that, starting from an empty taxonomy, allow us to build any (and only) correct taxonomies. To put this more precisely: in the area of formal verification, statements about a system are usually split between *correctness* and *completeness* properties. The correctness of a modeled system ensures that only desirable models are possible. In our setting, this means that only correct taxonomies can be part of the grammar language. On the other hand, *completeness* ensures that if a desirable system configuration can exist "in the real world", then a corresponding model is reachable in the formalization. In our setting, this means that any correct taxonomy can be created using the proposed graph grammar.

The grammar described in this section was created with GROOVE [3], a graph transformation tool suitable for the (partial) enumeration of a grammar language, which the tool calls the *state space exploration* of the graph grammar.

3.1 Introducing New Types

We start by defining transformation rules to introduce a new type in the taxonomy. Types for four of the leaf ontological metatypes given in Fig. 1 can be introduced in



(a) new-kind (b) new-category (c) new-mixin (d) new-antirigid-nonsortal

Fig. 2. Transformation rules to introduce an independent type.

the taxonomy without being related with a previously introduced type: these include all **Kinds** and all the non-sortals: **Categories**, **Mixins** and **Anti-Rigid Non-Sortals**.

Fig. 2 shows the four rules that introduce independent types, using the GROOVE visual notation for presenting rules. Each rule is formed only by a green box representing the type that will be *created* during rule application. A type has an ontological metatype (the label inside the box) and a name attribute. The "string" ellipses in Fig. 2 are the tool notation to indicate that the name must be provided (perhaps by the tool user) upon the type creation. No rule in Fig. 2 have preconditions. Therefore, types for these four ontological metatypes can be introduced without requiring the existence of other types or relations in the taxonomy.

3.2 Introducing Dependent Types

In contrast to non-sortals and kinds, **Subkinds** and **Anti-Rigid Sortals** have preconditions upon their introduction.

In the case of **Subkinds**, their introduction requires the existence of a previous sortal, from which the subkind will inherit a principle of identity. In addition, this sortal must be rigid, to respect the ontological principle that a rigid type cannot specialize an anti-rigid one. These preconditions for the introduction of a new **Subkind** are captured in the rule shown in Fig. 3. The *existing* **Rigid Sortal** is shown as a black box in the figure. The green "subClassOf" arrow states that a new direct subtyping relation will be introduced in the model.

Fig. 3. Transformation rule to introduce a **Subkind** type.

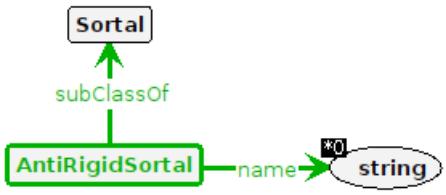


Fig. 4. Transformation rule to introduce an **Anti-Rigid Sortal** type.

In the case of an **Anti-Rigid Sortal** type, the only precondition is the existence of a previous sortal, from which the newly introduced Anti-Rigid Sortal will inherit a principle of identity. This rule is shown in Fig. 4. Differently from a Subkind, an Anti-Rigid Sortal can specialize any Sortal (and not only Rigid ones).

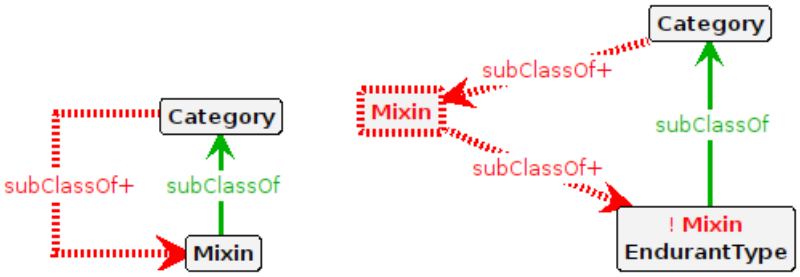
3.3 Introducing Specializations for Existing Non-Sortal Types

Having defined rules for the introduction of types, we proceed with rules to insert subtyping relations between two types already present in the model. We start with **Category** and **Mixin** specializations, as both of these ontological metatypes have meta-properties that allow their types to be specialized in any **Endurant Type**, without breaking formal ontology principles.

Fig. 5(a) shows a rule that creates a subtyping relation between an existing **Category** supertype and an existing **Mixin** subtype. The red arrow in the figure prevents the introduction of a circularity in the relations. Red elements in GROOVE rules indicate *forbidden* patterns, *i.e.*, elements that, if present, prevent the rule application. The label "subClassOf+" indicates direct or indirect subtyping. Circularity of specializations may be tolerated in taxonomies structured with improper specialization relations, such as `rdfs:subClassOf` in the Semantic Web. A consequence of circular specializations in that case is that mutually specializing classes become equivalent, and hence, should have the same ontological nature. Because of this, we rule out cases of circularity involving types of different metatypes such as a **Category** and a **Mixin**.

The forbidden pattern in Fig. 5(a) is not sufficient to prevent any circular subtyping relation. This occurs because while a **Mixin** can specialize a **Category**, the opposite relation is also possible. Therefore, in order to avoid any circularity, we separate the specialization of a **Category** by mixins and non-mixins Endurant Types. This second case is shown in Fig. 5(b). The forbidden pattern in this figure prevents cycles of subtyping relations involving Categories, Mixins and other Endurant Types. The red label "!Mixin" in Fig. 5(b) indicates that the existing **Endurant Type** cannot be a **Mixin**.

Analogously, we created two additional transformation rules to define how the specialization of a **Mixin** type can be made. These rules are not shown here due to their similarities to the ones in Fig. 5. Finally, the rule depicted in Fig. 6 allows the specialization of an **Anti-Rigid Non-Sortal** by another **Anti-Rigid Type**.



(a) add-category-specialization-in-mixin

(b) add-category-specialization

Fig. 5. Transformation rules to specialize a **Category**.

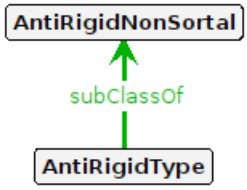


Fig. 6. Transformation rule to specialize an **Anti-Rigid Non-Sortal** type.

3.4 Introducing Generalizations for Existing Sortal Types

Kind types appear on the top of **Sortal** types hierarchies because kinds provide a principle of identity for all their instances. By definition, kinds cannot specialize other kinds. Therefore, they can only specialize **Non-Sortal** types, more specifically **Categories** and **Mixins**. These specializations can already be constructed with the rules presented in Section 3.3.

Subkind types, on the other hand, carry a principle of identity from their supertypes and, ultimately, from exactly one **Kind** type. The rule shown in Fig. 7 properly captures this restriction. If there are distinct (as defined by the *not equal* red dashed edge) **Sub-Kind** and **Rigid Sortal** types that carry a principle of identity from the same **Kind**, then a direct subtyping relation can be created between the two. The black edges with labels "subClassOf*" and "subClassOf+" indicate that, for the rule to be applied, a specialization relation from the new super-type and from the **Subkind** to the same **Kind** must already be present, or at least that the new (direct) super-type of the **Subkind** is its own **Kind**. Subkinds can also specialize any rigid or semi-rigid non-sortal, but these cases are already covered by the rules presented in Section 3.3. A similar construction for **Anti-Rigid Sortal** types can be seen in Fig. 8.

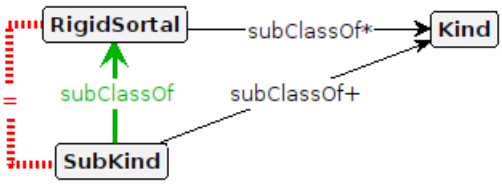


Fig. 7. Transformation rule to generalize a **Sub-Kind** type.

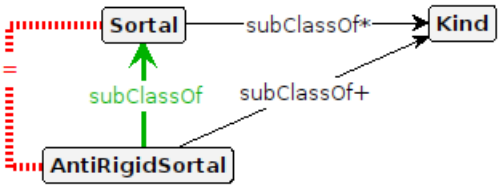


Fig. 8. Transformation rule to generalize an **Anti-Rigid Sortal** type.

4 Formal Verification

We use the GROOVE graph transformation tool to carry out a formal verification of the graph grammar presented in Section 3. To do so, we employ verification conditions in GROOVE, which formally define the ontological restrictions described in Section 2, and allow us to perform an analysis over any given taxonomy (graph state model). We then use the state space exploration functionality of the tool to check that all states (taxonomies) satisfy the restrictions.

As stated in Section 3, our objective with the verification is two-fold: to demonstrate the *correctness* and *completeness* of the proposed graph grammar. Correctness ensures that the grammar rules only produce correct taxonomies, *i.e.*, those that do not invalidate well-formedness constraints. Completeness ensures that any and all correct taxonomies can be produced by a sequence of rule applications.

A *graph condition* in GROOVE is represented diagrammatically in the same way as transformation rules, albeit without creator (green) elements. A graph condition is satisfied by a taxonomy model if all reader (black) elements of the condition are present in the model, and all forbidden (red) elements are absent.

Fig. 9 shows our first graph condition, capturing the restriction that **Kinds** must appear at the top of sortal hierarchies, hence not specializing another **Sortal**. It is important to note that restrictions are stated *positively* but are checked *negatively*. Thus, the condition in Fig. 9 characterizes an undesired model violation (a **Kind** specializing a **Sortal**), and therefore, by verifying that such condition never occurs in any taxonomy model, we can determine the grammar well-foundness. This same rationale holds for all other conditions shown in this section.

Fig. 10 formalizes a second restrictive condition, stating that a **Sortal** cannot inherit its principle of identity from more than one **Kind**. A third condition, shown in Fig. 11,

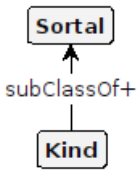


Fig. 9. Restrictive condition of a **Kind** specializing another **Sortal**.

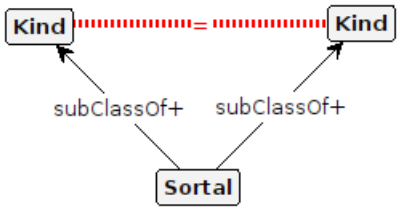


Fig. 10. Restrictive condition of a **Sortal** with more than one **Kind**.

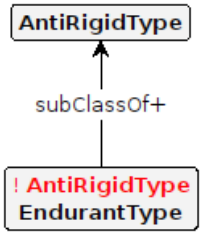


Fig. 11. Restrictive condition of a rigid or semi-rigid type specializing an anti-rigid one.

captures the situation in which the *rigidity* meta-property is contradicted, that is, when a rigid or semi-rigid type specializes an anti-rigid one. Similarly, the fourth restrictive condition, depicted in Fig. 12, represents the situation in which the *sortality* meta-property is contradicted, that is, when a **Non-Sortal** type specializes a **Sortal** one.

To specify a fifth restrictive condition, we consider that all **Sortals** ultimately should specialize (or be) a **Kind**, from which they inherit a principle of identity. The violating situation, in which a **Sortal** does not specializes a **Kind**, is shown in Fig. 13.

A final restriction is that any two types instantiating different ontological metatypes cannot have a mutual (circular) subtyping relation between them. We represent such restriction with 15 graph conditions, one for each pair of different ontological metatypes. Fig. 14 shows the graph condition for the pair **Category** and **Mixin**. The remaining 14 conditions all have the same structure, and thus are not shown.

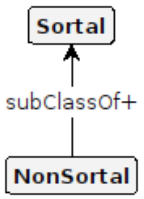


Fig. 12. Restrictive condition of a **Non-Sortal** type specializing a **Sortal** one.

Fig. 13. Restrictive condition of a **Sortal** without a **Kind**.

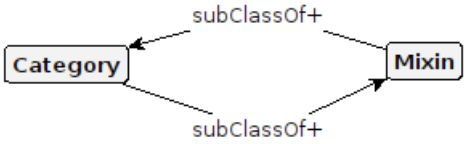


Fig. 14. An example of a condition with two equivalent types of different ontological metatypes.

4.1 Verifying Correctness

The first step in verifying the correctness of the graph grammar proposed is to enumerate its language, i.e., construct all possible taxonomies reachable by any sequence of rule applications. Subsequently, the graph conditions just presented are checked against these constructed taxonomies. If any model triggers one or more graph conditions, then we know the model violates some ontological restrictions, and therefore it is incorrect. Consequently, the goal of the correctness analysis is to verify that no taxonomy in the language is incorrect. To perform the grammar state space exploration we use the GROOVE Generator, a command-line tool designed for this task. Details of GROOVE usage can be found at the tool manual[4], and additional case studies that illustrate the tool functionalities are presented in [3].

A major caveat in the first step above is that the grammar language is *infinite*, thus preventing a complete enumeration in a finite amount of time. To cope with this situation, we need to perform a *bounded* exploration with the GROOVE tool. In this setting, our bound N is the maximum number of types present in a taxonomy. When performing the

[4] Available at `https://sourceforge.net/projects/groove/`

# types (N)	Produced taxonomies	Incorrect taxonomies
1	4	0
2	24	0
3	223	0
4	3,865	0
5	146,882	0
6	?	?

Table 1. Results of correctness analysis.

# types (N)	All taxonomies	Incorrect taxonomies	Correct taxonomies
1	6	2	4
2	78	54	24
3	2,456	2,233	223
4	228,588	224,723	3,865
5	?	?	?

Table 2. Results of completeness analysis.

exploration, the tool managed to generate a total of 150,998 taxonomies up to a bound $N = 5$, with a breakdown of this total per bound value shown in Table 1. The table also shows that our correctness goal was validated (at least up to $N = 5$), with no taxonomies being flagged as incorrect by the graph conditions.

Given the inherently exponential growth of the number of possible taxonomies with respect to bound N, it was not possible to continue the exploration for $N = 6$ and beyond due to memory limitations (the execution was halted after several million models partially produced.) This *state space explosion* is a common problem for all explicit state model checkers, such as GROOVE [3].

To support that the correctness results in Table 1 are significant, we rely on the *small scope hypothesis*, which basically claims most design errors can be found in small counterexamples [2]. Experimental results suggest that exhaustive testing within a small finite domain does indeed catch all type system errors in practice [14], and many case studies using the tool Alloy have confirmed the hypothesis by performing an analysis in a variety of scopes and showing, retrospectively, that a small scope would have sufficed to find all the bugs discovered [12].

4.2 Verifying Completeness

The verification described in the previous section assures that all taxonomies produced are correct, but does nothing to persuade us that any and all possible correct taxonomies can be produced. To provide this kind of assurance is the goal of the completeness verification described below.

To perform the completeness analysis we need to consider not only correct taxonomies but also the incorrect ones. To this end, we developed another, completely permissible, graph grammar that allows the creation of both correct and incorrect models. The grammar is quite simple, with six rules for the unrestricted creation of the leaf

types of types in Fig. 1, and one rule allowing the introduction of a subtyping relation between any two endurant types.

The results of the exploration with this new permissible grammar are presented in Table 2. As expected, the rate of growth in this scenario is even steeper, given that more models can be produced. The tool was able to perform a bounded exploration up to $N = 4$, with larger bounds exceeding the available memory. The second column of Table 2 lists all taxonomies created with the new grammar, both correct and incorrect. We again use the graph conditions to flag violations of ontology restrictions in the models. If a taxonomy triggers *any* of the graph conditions, then it is considered incorrect. Conversely, if *no* graph condition is triggered by a model, then it certainly describes a correct taxonomy. The last two columns in the table summarize this classification.

The completeness goal can be verified by a comparison between the **Correct taxonomies** column of Table 2 and the **Produced taxonomies** column of Table 1. It can be seen immediately that all values up to $N = 4$ match. Given that the permissible grammar produces all possible models (correct and incorrect), this allows us to conclude that the taxonomy grammar of Section 3 produces *all* correct taxonomies, and *only* the correct ones. To strengthen this validation claim we once again rely on the small scope hypothesis: although the completeness result is not formally proven for models of arbitrary size, the bounded values shown provide strong evidence that such result holds. Also, the bound limit could be pushed (at least a bit) further with additional computational resources and time.

5 Final Considerations

In this paper, we propose a systematic approach for building ontologically well-founded and logically consistent taxonomic structures. We do that by leveraging on a *typology of endurant types*. This typology, in turn, is derived from an ontological theory that is part of the Unified Foundational Ontology (UFO) [7], and which underlies the Ontology-Driven Conceptual Modeling language OntoUML [6].

The original theory puts forth a number of ontological distinctions based on formal meta-properties. As a result of the logical characterization of these meta-properties, we have that certain structures (patterns) are imposed on the language primitives representing these distinctions [15]. We have identified a set of primitive operations on taxonomic structures that, not only guarantees the correctness of the generated taxonomies, but also is capable of driving the construction of any correct taxonomy. This forms the basis for the systematic design of such structures at a higher level of abstraction.

Given the limitations of metamodels as a mechanism for representing a language's abstract syntax, these structures were not treated as first-class citizens before and have remained hidden in the abstract syntax of the original OntoUML proposal [5]. This paper addresses this exact problem. By leveraging on that theory, and propose a *pattern grammar* (graph transformation grammar) that embeds these distinctions and that guarantees *by design* the construction of taxonomic structures that abide by the formal constraints governing their relations. The work proposed here advances the work initiated in [17]. For example, by employing the state exploration mechanism supported by GROOVE, we managed to detect important omissions in the rule set proposed in that original work.

Another important aspect is that our proposal captures the representation consequences of that ontology theory in a way that is metamodel-independent. For this reason, these results can be carried out to other languages and platforms. In particular, we are currently developing a plugin for Protégé that, among other things, implements the primitive operations proposed in this paper. This plugin is intended to be used in tandem with the gUFO ontology (a lightweight implementation of UFO) [1]. In that implementation, these operations take the form of ontology patterns to be applied, to support its users in modeling consistent Semantic Web ontologies.

Acknowledgments

This research is partly funded by Brazilian funding agencies CNPq (grants numbers 312123/2017-5 and 407235/2017-5) and CAPES (Finance Code 001 and grant number 23038.028816/2016-41).

References

1. Almeida, J.P.A., Guizzardi, G., Falbo, R.A., Sales, T.P.: gUFO: a lightweight implementation of the Unified Foundational Ontology (UFO) (2019)
2. Gammaitoni, L., Kelsen, P., Ma, Q.: Agile validation of model transformations using compound f-alloy specifications. Science of Computer Programming **162**, 55–75 (2018)
3. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. International journal on software tools for technology transfer **14**(1), 15–40 (2012)
4. Guarino, N., Welty, C.A.: An overview of OntoClean. In: Handbook on ontologies, pp. 151–171. Springer (2004)
5. Guizzardi, G.: Ontological foundations for structural conceptual models. No. 15 in Telematica Institute Fundamental Research Series, University of Twente (2005)
6. Guizzardi, G., Fonseca, C.M., Benevides, A.B., Almeida, J.P.A., Porello, D., Sales, T.P.: Endurant types in ontology-driven conceptual modeling: Towards OntoUML 2.0. In: International Conference on Conceptual Modeling. pp. 136–150. Springer (2018)
7. Guizzardi, G., Wagner, G., Almeida, J.P.A., Guizzardi, R.S.: Towards ontological foundations for conceptual modeling: The unified foundational ontology (UFO) story. Applied ontology **10**(3-4), 259–271 (2015)
8. Guizzardi, G., Wagner, G., Guarino, N., van Sinderen, M.: An ontologically well-founded profile for UML conceptual models. In: International Conference on Advanced Information Systems Engineering. pp. 112–126. Springer (2004)
9. Halpin, T., Morgan, T.: Information modeling and relational databases. Morgan Kaufmann (2010)
10. Heckel, R.: Graph transformation in a nutshell. Electronic notes in theoretical computer science **148**(1), 187–198 (2006)
11. Hirsch, E.: The concept of identity. Oxford University Press (1992)
12. Jackson, D.: Alloy: a language and tool for exploring software designs. Communications of the ACM **62**(9), 66–76 (2019)
13. Oltramari, A., Gangemi, A., Guarino, N., Masolo, C.: Restructuring WordNet's top-level: The OntoClean approach. LREC2002, Las Palmas, Spain **49** (2002)
14. Roberson, M., Harries, M., Darga, P.T., Boyapati, C.: Efficient software model checking of soundness of type systems. ACM Sigplan Notices **43**(10), 493–504 (2008)

15. Ruy, F.B., Guizzardi, G., Falbo, R.A., Reginato, C.C., Santos, V.A.: From reference ontologies to ontology patterns and back. Data & Knowledge Engineering **109**, 41–69 (2017)
16. Zambon, E.: Abstract Graph Transformation – Theory and Practice. Centre for Telematics and Information Technology, University of Twente (2013)
17. Zambon, E., Guizzardi, G.: Formal definition of a general ontology pattern language using a graph grammar. In: 2017 Federated Conference on Computer Science and Information Systems (FedCSIS). pp. 1–10. IEEE (2017)