FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	OntoUML Models Verification for the OpenPonk platform
Student:	Marek Bělohoubek
Supervisor:	Ing. Robert Pergl, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

- Acquaint yourself with the OpenPonk conceptual modelling platform.
- Acquaint yourself with Unified Foundational Ontology and the OntoUML language with the respect to meta-model rules.
- Design and implement OntoUML model syntax checking for the OpenPonk platform.
- Test your solution using unit tests, demonstrate it on a case study.
- Document and discuss your solution.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 13, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

OntoUML Models Verification for the OpenPonk platform

Marek Bělohoubek

Department of Software Engineering
Supervisor: doc. Ing. Robert Pergl, Ph.D.

May 14, 2019

Acknowledgements

I would like to thank doc. Ing. Robert Pergl, Ph.D., my supervisor, for guiding me through creation of this work, and Peter Uhnák for providing me crucial information about OpenPonk when it was most needed.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 14, 2019 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Bělohoubek, Marek. *OntoUML Models Verification for the OpenPonk platform.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato práce se zaměřuje na vytvoření nového frameworku, který bude použit pro verifikaci OntoUML modelů na platformě OpenPonk. Pro to je nutné nejdříve popsat a analyzovat jazyk OntoUML a platformu OpenPonk. Na základě této analýzy jsou následně vytvořeny dva návrhy verifikačního frameworku. Dle lepšího z nich je framework následně implementován. Veškerý nový kód je zdokumentovván a jsou pro něj vytvořeny testy. Na závěr je celý framework demonstrován na referenčním modelu.

Klíčová slova Verifikace, OntoUML, OpenPonk, Pharo, Konceptuální model, Unified foundation ontology, Ontologický model.

Abstract

This work focuses on creation of new verification framework that will be used for verification of OntoUML model on OpenPonk platform. It is necessary to describe and analyse OntoUML language and OpenPonk platform. Using this analysis two designs are created for the verification framework. The better one is then selected and implemented. All new code is documented and unit test are created for him. Finally entire framework is demonstrated on reference model.

Keywords Verification, OntoUML, OpenPonk, Pharo, Conceptual model, Unified foundation ontology, Ontological model.

Contents

Introduction **1**

Goals **3**

I Review **5**

1 OntoUML **7**
1.1 Basic concepts 7
1.2 Entity stereotypes 10
1.3 Relation stereotypes 15

2 OpenPonk **19**
2.1 Pharo 19
2.2 Data model 20

II Analysis and design **21**

3 Analysis **23**
3.1 AllowedSupertype & AllowedSubtype 23
3.2 RelationSource & RelationTarget 23
3.3 IdentityRequired 24
3.4 CharacterizationDependency 24
3.5 PartOfGeneralizationSet 24
3.6 RoleMediationDependency 24
3.7 RelatorMediationDependency 24

4 Initial design **25**
4.1 DesignChecker 25

4.2 OntoUML DesignChecker 27
4.3 Problems 30

5 Final design **31**

IIIImplementation, documentation and testing **39**

6 Reference model **41**
6.1 Reference model design 41
6.2 AllowedSupertype & AllowedSubtype model 41
6.3 RelationSource model 41
6.4 RelationEnd model 43
6.5 Characterized model 44
6.6 PartOfGeneralizationSet model 44
6.7 IdentityRequired model 44
6.8 RoleMediationDependency model 45
6.9 RelatorMediationDependency model 46

7 Documentation and testing **47**
7.1 Documentation 47
7.2 Testing 48

Conclusion **49**

Bibliography **51**

A Acronyms **53**

B Contents of enclosed CD **55**

List of Figures

1.1 Simplified model of all OntoUML stereotypes for entities 11

4.1 Initial design diagram 26

5.1 Final design diagram 32

6.1 Complete reference model 42
6.2 Reference model for *AllowedSupertype* and *AllowedSubtype* rules . . 43
6.3 Reference model for *RelationSource* rule 43
6.4 Reference model for *RelationEnd* rule 43
6.5 Reference model for *Characterized* rule 44
6.6 Reference model for *PartOfGeneralizationSet* rule 44
6.7 Reference model for *IdentityRequired* rule 45
6.8 Reference model for *RoleMediationDependency* rule 45
6.9 Reference model for *RelatorMediationDependency* rule 46

7.1 Documentation for `VerificationResults` class 47
7.2 Unit tests for `IdentityRequiredVerification` class 48

Introduction

With increasing popularity of OntoUML language, demand for tools that make conceptual modelling easier increases as well. One of this tools is OpenPonk platform developed using Pharo programming language by Czech Technical University in Prague.

This platform allows quick and easy creation of conceptual models, but it didn't contain any framework for verifying those models. With incoming preparations to use OpenPonk for educational purposes, this became significant problem.

Thus it was decided to create new framework, that will allow easy verification of conceptual models created by rules of OntoUML language.

This thesis is structured to three main parts. First part is called "Review" and it introduces reader to basic concepts of OntoUML language, and current state of the OpenPonk platform.

Second part is called "Analysis and Design" and it focuses on analysis of OntoUML rules and describes design of two versions of verification framework, that were designed during work on this thesis.

Third part is called "Implementation, documentation and testing". It focuses on tests performed on reference model and mentions both unit test and documentation.

Goals

Main goal of this bachelors thesis is to analyse, design, implement, test, and document automatic verification for OntoUML Models on the OpenPonk platform.

First the analytical part consists of analysis of OntoUML language (identification of all entities and association, including all of their constraints), and analysis of OpenPonk platform.

Second it is necessary to design new verification framework that will allow user to perform verification on any OntoUML model. This framework has to be easily modifiable and expandable, because both OntoUML language and OpenPonk are being constantly developed and changed.

Third the implementation consist of verification framework itself (including verification for all OntoUML entities and relations that can be validated), and user interface that allows user to start and view results of the verification.

Fourth the newly implemented code will have unit tests created for it, and entire verification framework will be tested on reference model.

Finally documentation of the verification framework will be be part of the newly implemented code, because of the nature of both OpenPonk platform and Pharo programming language, that allow users to customise modelling environment to their needs.

Part I

Review

Chapter 1

OntoUML

This chapter introduces reader to the basics of OntoUML language. It starts with basic concepts and rules. Then it continues with describing all OntoUML stereotypes for entities and it ends with explaining all OntoUML stereotypes for relations.

Information contained in this chapter were obtained from following sources[1, 2, 3, 4, 5].

1.1 Basic concepts

This section describes basic concepts and rules of OntoUML language. Those are: modal logic, identity principle, generalization, rigidity, sortals, non-sortals and aspects.

1.1.1 Modal logic

OntoUML language is build upon modal logic, which expands predicate logic, by adding modality to it. This allows us to model reality much more accurately in relation to *worlds*, that are defined by space and time.

Because modal logic is expansion of predicate logic, it carries over all logical operations (negation, conjunction, disjunction, implication and equivalence) and quantifiers (existential and universal). It also adds two new modal operators:

possibility operator $\Diamond statement$: *statement* is true in at least one world

necessity operator $\Box statement$: *statement* is true in all worlds

Predicate and modal operations and quantifiers can be combined together, thus making the model more accurate. For example "All green apples are not edible, is true in at least one world" can be represented like this:

$$\Diamond(\forall apple) : Green(apple) \implies \neg Edible(apple)$$

1.1.2 Identity principle

Identity is one of the fundamental ontological principles and every entity mapped by conceptual model has to have unique identity. This allows us to identify the object at any time during its existence.

Even though we can intuitively feel that particular entity has its own identity, it can be very hard to determine what exactly defines this identity. For example statue can be identified it by its shape, but if we then cut it in half, then by our definition of identity, we will create two new entities and the original statues identity will be lost.

This example shows that defining identity for entities in particular domain can be one of the most difficult task when working on conceptual model, because it will have severe consequences for the entire model.

1.1.3 Generalization

One of many things that OntoUML inherits from UML is generalisation. It is used to indicate that one entity is subtype of another entity. This means that subtype inherits all variables, methods and relations from supertype.

Multiple generalizations originating from the same supertype can be joined into single generalization set. Each set has two additional properties called *disjoint* (all subtypes in the set are mutually exclusive) and *complete* (set contains all possible subtypes for generalised supertype). This allows further specialisation of generalisation set.

1.1.4 Rigidity principle

Rigidity is ontological meta-property, that defines mutability of the type, i.e., it defines if instances of type are connected to the type in all worlds or not.

Here are all four types of rigidity:

1.1.4.1 Rigidity

"*Type T is* ***rigid (R+)****, if it applies to its instances (necessarily) in all worlds. In another words entity doesn't change its type in any world.*"[5](translated by author)

$$R+(T) = \Box(\forall x)(T(x) \implies \Box(T(x)))$$

1.1.4.2 Anti-Rigidity

"*Type T is* ***anti-rigid (R−)****, if its possible, that every instance of entity that this type applies to in one world, is not applied by this type in another (possible) world.*"[5](translated by author)

$$R-(T) = \Box(\forall x)(T(x) \implies \Diamond(\neg T(x)))$$

1.1.5 Non-Rigidity

Because anti-rigidity isn't logical negation of rigidity (it is stronger statement), non-rigidity is defined as its logical negation, i.e., type T is **non-rigid (NR)**, if at least one instance that this type applies to in one world, is not applied by this type in another (possible) world.

$$NR(T) = \Diamond(\exists x)(T(x) \wedge \Diamond(\neg T(x)))$$

1.1.5.1 Semi-Rigidity

"*Special case is* ***semi-rigid (R∼)*** *type, that is rigid for some of its instances and non-rigid for others.*"[5](translated by author)

$$SR(T) = (\exists x, y)(x \neq y)((T(x) \implies \Box(T(x))) \wedge (\Diamond T(y) \wedge \Diamond(\neg T(y))))$$

1.1.5.2 Rigidity principle and generalization

Rigidity principle of the type has great impact on generalization as it determines possible supertypes of the type. Here is list of possible supertypes for all rigidity types:

- Allowed supertypes for *rigid* type are: *rigid*, *semi-rigid*.
- Allowed supertypes for *anti-rigid* type are: *rigid*, *anti-rigid*, *semi-rigid*.
- Allowed supertypes for *non-rigid* type are: *rigid*, *anti-rigid*, *non-rigid*, *semi-rigid*.
- Allowed supertype for *semi-rigid* is: *semi-rigid*.

1.1.6 Sortals and Non-Sortals

Sortals have identity principle and provide it to other entities. *Sortals* are used to represent entities from modelled domain.

Non-sortals don't have identity and thus can't provide it to other entities. *Non-sortals* are used to represent features and properties that are common for several entities.

Sortals and non-*sortals* have to follow this rules for inheritance:

- *Sortal* can be subtype of another *sortal* or *non-sortal*.
- *Non-sortal* can be subtype only of another *non-sortal*.

1.1.7 Aspects

Aspects are existentially dependent entities. Object on which the aspect depends is called bearer and his aspects both begin and end their existence at the same time as him.

Aspects are used to map properties of entities. Those properties can be either structured/measurable (*quality*), or non-structured/non-measurable (*mode*).

Mapping property as aspect is used to highlight existence of the property, or if the property requires some special handling.

1.2 Entity stereotypes

This section focuses on OntoUML entity stereotypes. First it shows their structure on simplified model and then it lists all entity stereotypes.

Figure 1.1 shows simplified model of stereotype structure. This model has been created using information from [2, 3].

1.2.0.1 Kind

Kind supplies principle of identity for its instances and it can be further specified by other rigid subtypes.

Type	*Rigid Sortal*
Allowed supertypes	*Category, Mixin*
Allowed subtypes	*Subkind, Phase, Role*
Constraints	*Kind* has no additional constraints.
Examples	Car, tree, card, person.

1.2.0.2 Subkind

Subkind is used to specify other *rigid sortal* types. It is the only *rigid sortal* that requires identity.

Type	*Rigid Sortal*
Allowed supertypes	*Kind, Subkind, Collective, Quantity, Relator, Category, Mixin*
Allowed subtypes	*Subkind, Phase, Role*

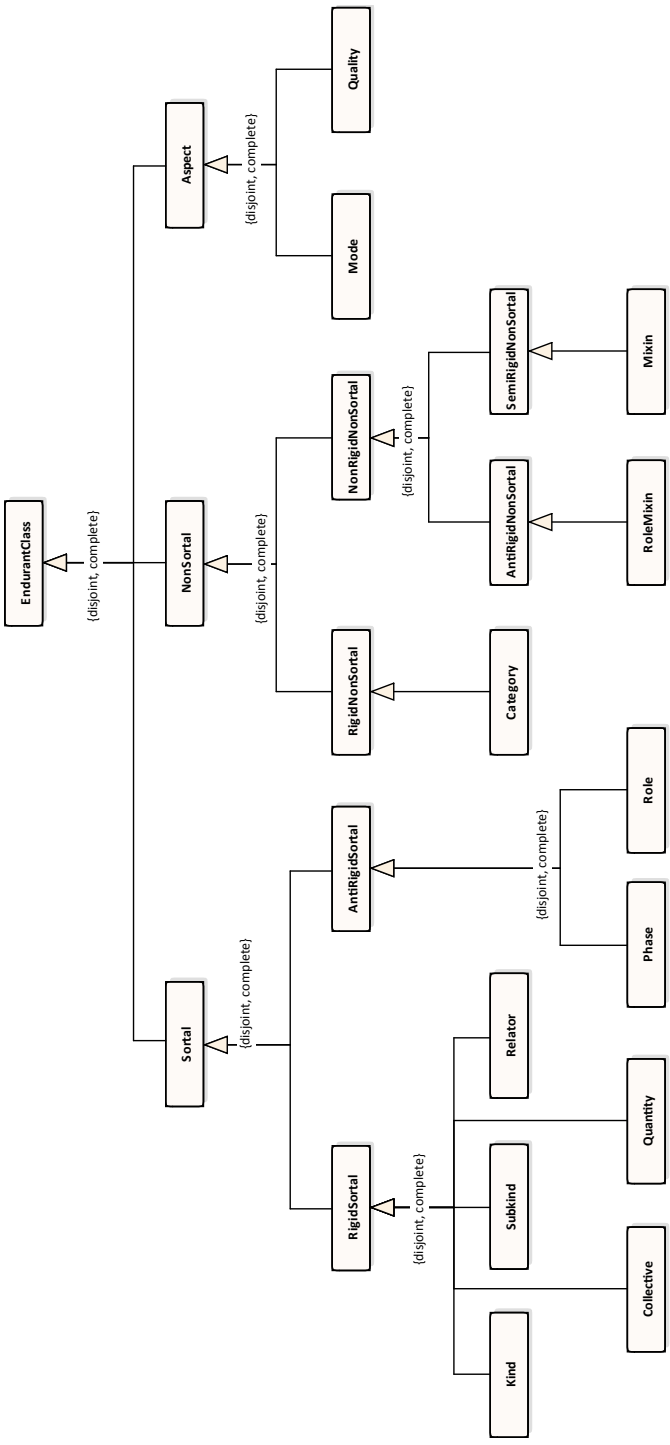

Figure 1.1: Simplified model of all OntoUML stereotypes for entities [2, 3](edited by author)

Constraints	*Subkind* requires exactly one identity principle from its supertypes. Identity provider type doesn't have to be direct supertype, but it has to exist and there has to be only one identity provided to the *subkind*.
Examples	Man and woman (as *subkinds* of person), oak (as *subkind* of tree), Model T (as *subkind* of car)

1.2.0.3 Collective

Collective represents homogeneous internal structure, this means that for whole all parts are perceived as equal.

Type	*Rigid Sortal*
Allowed supertypes	*Category, Mixin*
Allowed subtypes	*Subkind, Phase, Role*
Constraints	*Collective* has no additional constraints.
Examples	Forest, array, group of tourists.

1.2.0.4 Quantity

Quantity represents uncountable objects such as sand, oil and water in maximally topologically connected amount.

Type	*Rigid Sortal*
Allowed supertypes	*Category, Mixin*
Allowed subtypes	*Subkind, Phase, Role*
Constraints	*Quantity* can have only other quantites as parts.
Examples	Sand, oil, water, dirt.

1.2.0.5 Relator

Relator is used to model "truth makers", objects that are created through *material* relation and that guarantee its existence.

Type	*Rigid Sortal*
Allowed supertypes	*Category, Mixin*
Allowed subtypes	*Subkind, Phase, Role*

Constraints *Relator* must be part of at least one mediation and total multiplicity on the other ends of all connected mediations have to be at least two.

Examples Marriage, order, contract.

1.2.0.6 Phase

Phase represents state of the object that can change in the time.

Type *Anti-rigid Sortal*

Allowed supertypes *Kind, Subkind, Collective, Quantity, Relator, Phase, Mixin*

Allowed subtypes *Phase, Role*

Constraints *Phase* requires exactly one identity principle from its supertypes. Identity providing type doesn't have to be direct supertype, but it has to exist and there has to be only one identity provided to the *phase*.

Phase has to be part of generalization set that is both covering and disjoint.

Examples Baby, child, teenager, adult and senior.

1.2.0.7 Role

Role is used to specialise entities in relational context.

Type *Anti-rigid Sortal*

Allowed supertypes *Kind, Subkind, Collective, Phase, Quantity, Relator, Role, RoleMixin, Mixin*

Allowed subtypes *Role*

Constraints *Role* requires exactly one identity principle from its supertypes. Identity providing type doesn't have to be direct supertype, but it has to exist and there has to be only one identity provided to the *role*.

Role is relationally depended, this means that the role has to be part of at least one relation.

Examples Student, teacher and headmaster (as *roles* of person)

1.2.0.8 Category

Category is used to aggregate property (or properties) that is essential for existence of multiple entities with different identities.

Type	*Rigid Non-sortal*
Allowed supertypes	*Category, Mixin*
Allowed subtypes	*Kind, Subkind, Collective, Quantity, Relator, Category*
Constraints	*Category* has no additional constraints.
Examples	Vehicle, object, living thing.

1.2.0.9 RoleMixin

RoleMixin aggregates multiple roles (with different identities) into single object.

Type	*Anti-rigid Non-sortal*
Allowed supertypes	*Mixin, RoleMixin*
Allowed subtypes	*Role, RoleMixin*
Constraints	*RoleMixin* has no additional constraints.
Examples	*Customer* (for *roles* "CompanyCustomer" and "IndividualCustomer").

1.2.0.10 Mixin

Mixin represent property that is essential for existence of some objects but optional for others.

Type	*Semi-rigid Non-sortal*
Allowed supertypes	*Mixin*
Allowed subtypes	*Subkind, Kind, Collective, Quantity, Category, Mixin, Role, Phase, RoleMixin, Relator*
Constraints	*Mixin* has no additional constraints.
Examples	Luxury goods (for *phase* RarePainting and *kind* Diamond).

1.2.0.11 Mode

Mode represents non-structured property of entity, that it characterizes.

Type *Aspect*

Allowed supertypes None

Allowed subtypes None

Constraints *Mode* has to be part of at least one *characterization* and multiplicity on the other end must be always equal to one.

Examples Mood, desire, ability.

1.2.0.12 Quality

Quality represents structured property of entity, that it characterizes.

Type *Aspect*

Allowed supertypes None

Allowed subtypes None

Constraints *Quality* has to be part of at least one *characterization* and multiplicity on the other end must be always equal to one.

Examples Weight, length, maximal speed, price.

1.3 Relation stereotypes

Here is list of all relation stereotypes:

1.3.0.1 Formal

This relation can be reduced to direct comparison between two entities. The name *Formal* is short for *Domain Comparative Formal Relation*.

Allowed source types All

Allowed end types All

Examples Faster-than, stronger-than, younger-than.

1.3.0.2 Material

It represents relations, that have material structure. Each *material* relation creates *relator*, which also acts as "truth maker" of the relation.

Allowed sources All

Allowed ends All

Examples Wife and husband are married, creditor and debitor have signed contract.

1.3.0.3 Mediation

Mediations are defined between entites, that are connected with *material* relation, and relator that acts as "truth marker" of this relation.

Allowed sources *Relator*

Allowed ends All

Examples Contract (*relator*) is signed by creditor and debitor, marriage exist between wife and husband.

1.3.0.4 Characterization

Characterization is relation between type and its property.

Allowed sources All

Allowed ends *Mode, Quality*

Examples Mood characterizes person, top speed characterizes car, width characterizes parcel.

1.3.0.5 Derivation

Derivation is used for highlighting connection between *relator* and *material* relation in which it acts as „truth marker". *Derivation* currently isn't fully implemented in OpenPonk.

Allowed sources *Relator*

Allowed ends *Material* relation

Examples Marriage (*relator*) is connected to marriedWith (*material* relation), contract (*relator*) is connected to contractedBy (*material* relation).

1.3.0.6 Structuration

Structuration is used for creating advanced structure for quality. *Structuration* is currently not implemented in OpenPonk.

Allowed sources	*Quality*
Allowed ends	*Quality, Mode*
Examples	Gram and pound structure weight, inch and meter structure height.

1.3.0.7 Part-Whole

It represents relation between whole and its part. OntoUML distinguish parts on more granular level than UML. It distinguishes between shared parts that can be part of multiple wholes and exclusive parts that can be part of single whole in any world.

This type of relation is not currently implemented in OpenPonk.

Allowed sources	All
Allowed ends	All
Examples	Shared part: person is part of club. Essential part: engine is part of car, tree is part of forest.

1.3.0.8 ComponentOf

"*Component of represents parthood relation between two complexes.*"[3]

Allowed sources	*Kind, Subkind, Phase, Role, Category, Role-mixin, Mixin, Mode, Quality*
Allowed ends	*Kind, Subkind, Phase, Role, Category, Role-mixin, Mixin, Mode, Quality*
Examples	Harddisk is part of computer, wheels are part of motorbike.

1.3.0.9 Containment

Containment represents relation between container and its contents.

Allowed sources	All
Allowed ends	*Quantity*
Examples	Milk in bottle, sand in bucket, fuel in fuel tank.

1.3.0.10 MemberOf

MemberOf represent relation between *collective* (as whole) and functional complex or *collective* (as part).

Allowed sources	*Collective*
Allowed ends	*Collective*, Functional complex
Examples	Band member is part of band, card is part of card deck, tree is part of forest.

1.3.0.11 SubCollectionOf

SubCollectionOf represent relation between collection (*collective*) and its subcollection (*collective*).

Allowed sources	*Collective*
Allowed ends	*Collective*
Examples	Collection of pawns is part of chess figures, collection of oaks is part of forest, collection of aces is part of deck.

1.3.0.12 SubQuantityOf

SubQuantityOf represent relation between two *quantities*.

Allowed sources	*Quantity*
Allowed ends	*Quantity*
Examples	Sugar is part of candy, hydrogen is part of water, alcohol is part of beer.

Chapter 2

OpenPonk

This section gives reader basic information about OpenPonk platform (downloadable from [6]). It starts with basics of Pharo language (downloadable from [7]) on which OpenPonk is build, then it continues with data model of OpenPonk platform and it ends witch brief description of its editor.

2.1 Pharo

"*Pharo is a pure object-oriented programming language and a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one).*"[8]

This subsection uses informations from following sources [8, 9, 10].

Pharo is open-source dialect of SmallTalk programming language. It is purely object-oriented dynamically typed language. Object-orientation means that every single element from basic types to UI (User Interface) elements is an object. And dynamically typed language stands for language that checks object types only during runtime.

One of its advantages is compact syntax, that follows its straightforward rules. As its authors say, this syntax can be fitted on single postcard as you can see here[10]. I would strongly recommend it for anyone that would like to read or create code in Pharo.

Similarly to SmallTalk, Pharo is also contained within its own live environment that acts both as its IDE and its OS. Since the environment is live, it allows direct feedback to changes in code.

For example you can look at all instances that exist in the environment, inspect them and send them messages. Not only that but you can write programs that can access and edit their own source code during runtime.

2.2 Data model

This subsection focuses on most important classes in the OpenPonk data model. It starts with class representing conceptual model itself, continues with classes corresponding with entities, relations and generalisations, and ends with classes implementing stereotypes.

2.2.1 OPUMLElement

`OPUMLElement` is parent of most classes from data model, including class `OPUMLModel`. It defines unified interface for all its subtypes, most importantly its two initialization methods `initializeSharedGeneralizations` and `initializeDirectGeneralizations` that apply composition.

2.2.2 OPUMLModel

`OPUMLModel` represents the data part of the conceptual model itself. It holds all other entities, relations, generalizations…

For purposes of verification framework is crucial its `packagedElements` method, that returns collection of all elements stored by this model.

2.2.3 OPUMLClass

`OPUMLClass` represents entity in the conceptual model. By applying OntoUML stereotypes to this class we get all OntoUML entities.

2.2.4 OPUMLAssociation

`OPUMLAssoiation` represents relation in the conceptual model. By applying OntoUML stereotypes to this class we get most of OntoUML relations (OpenPonk doesn't fully implement all relations at the moment).

2.2.5 OPUMLGeneralization

`OPUMLGeneralization` represents generalization in the conceptual model. It contains references to all generalization sets which its part of.

It also contains references to both ends of the generalization that can be accessed through methods: `general` (supertype) and `specific` (subtype).

2.2.6 OPUMLGeneralizationSet

`OPUMLGeneralizationSet` represents single generalization set in the model. It contains references to all generalizations that are part of it.

Its two most important properties are accessible by methods named after them: `isCovering` (generalizations cover all possible options) and `isDisjoint` (generalizations are mutually exclusive).

Part II

Analysis and design

Chapter 3

Analysis

Previous chapters described OntoUML and OpenPonk. Notably for OntoUML they contained lists of all stereotypes for entities and relations, including their constraints and examples.

This chapter focuses on those constraints, because they (together with basic OntoUML rules) will have to be transformed into verifications.

First I will focus on general constraints that are common for multiple entities/relations and then continue with more specific ones. Those constraint will be presented as verification rules and please note that I have chosen their names myself, since they were created as part of this analysis, therefore there is no official terminology.

Analysis itself won't be delving into OpenPonk much, because OpenPonk is main subject of design oriented chapters.

3.1 AllowedSupertype & AllowedSubtype

First two verification rules are directly connected to entity stereotypes and generalization. Every stereotype has its own sets of possible subtypes and supertypes.

This means that every generalization must be verified, once for the subtype and once for the supertype. Even though it could be done by single rule, it was decided to split them into two, to highlight importance of this rule.

3.2 RelationSource & RelationTarget

As their names suggest, these two rules are associated with relations and entities on their ends.

Similarly to entity stereotypes, relation stereotypes define set of allowed source types and set of allowed target types.

Note that unlike *AllowedStereotype* and *AllowedSubtype* rules, *RelationSource* and *RelationTarget* aren't parts of single rule, i.e., one particular relation can be follow *RelationSource* and still break *RelationTarget*.

3.3 IdentityRequired

Some entity stereotypes require identity, that must be provided by one of its supertypes. Thus we need to verify that exactly one supertype (direct or indirect) for entity requiring identity is identity provider and that there are no loops in the generalisation structure.

Here is list of all identity providers according to official ontouml.org portal[3]: *kind*, *collective*, *quantity*, *mode*, *quality*.

3.4 CharacterizationDependency

Stereotypes *mode* and *quality* relationally dependant on characterization relation. This relation has to have multiplicity on the opposite end (opposite to *mode*/*quality*) equal exactly to one.

3.5 PartOfGeneralizationSet

Stereotype *phase* requires by its definition to be part of disjoint and complete generalisation set. This means that all phases are mutually exclusive (disjoint) and they cover all possible options (complete).

3.6 RoleMediationDependency

This rule is specific for stereotype *role*. It states that each role has to be connected directly or indirectly (only if directly connected supertype is stereotyped as *role*/*rolemixin*) through at least one mediation to *relator*.

Lower bound of multiplicity on end opposite to *role*/*rolemixin* has to be at least one.

3.7 RelatorMediationDependency

This rule is specific for stereotype *relator*. It is similar to rule *RoleMediationDependency*, because it requires that each *relator* is part of at least one relation stereotyped as *mediation*.

Unlike *RoleMediationDependency*, this rule requires that total count of all multipl icities on *non-relator* ends cannot be lower than two.

Chapter 4

Initial design

Verification of the conceptual mode should check, that all entities and relations in this model are created according to OntoUML rules. To fulfil this requirement new verification framework had to be designed and implemented.

This chapter contains first of two possible designs for this framework that were created during my work on this framework.

Initial design of verification framework was inspired by solutions that I have seen and worked on, mainly by tool called DesignChecker that is part of commercial software xTractor[11]. I know this tool very well, since I was contributing to its development.

Figure 4.1 shows UML diagram summarising changes in this design.

4.1 DesignChecker

Goals of DesignChecker are same as goals of the verification framework: verify elements (rules for xTractor, entities for OpenPonk) and their relations, identify any violations of the OntoUML rules and inform user about them.

This tool has verification methods as part of elements and relations. Those verifications are connected directly to the properties of verified objects through their setters. This way any time property value gets changed, its verification is automatically called and its result is saved in the object.

Results themselves are represented as warning/error messages and they contain reference to the verified object, name of the verified property and short text summarizing the problem.

In addition to the verification functions themselves, every object that can be verified has methods called HasErrors and HasWarnings. Those methods used to check if the object has any error/warning message stored in results collection.

DesignChecker also contains verification controller, that holds reference for the verified model and provides interface for verifying all elements and

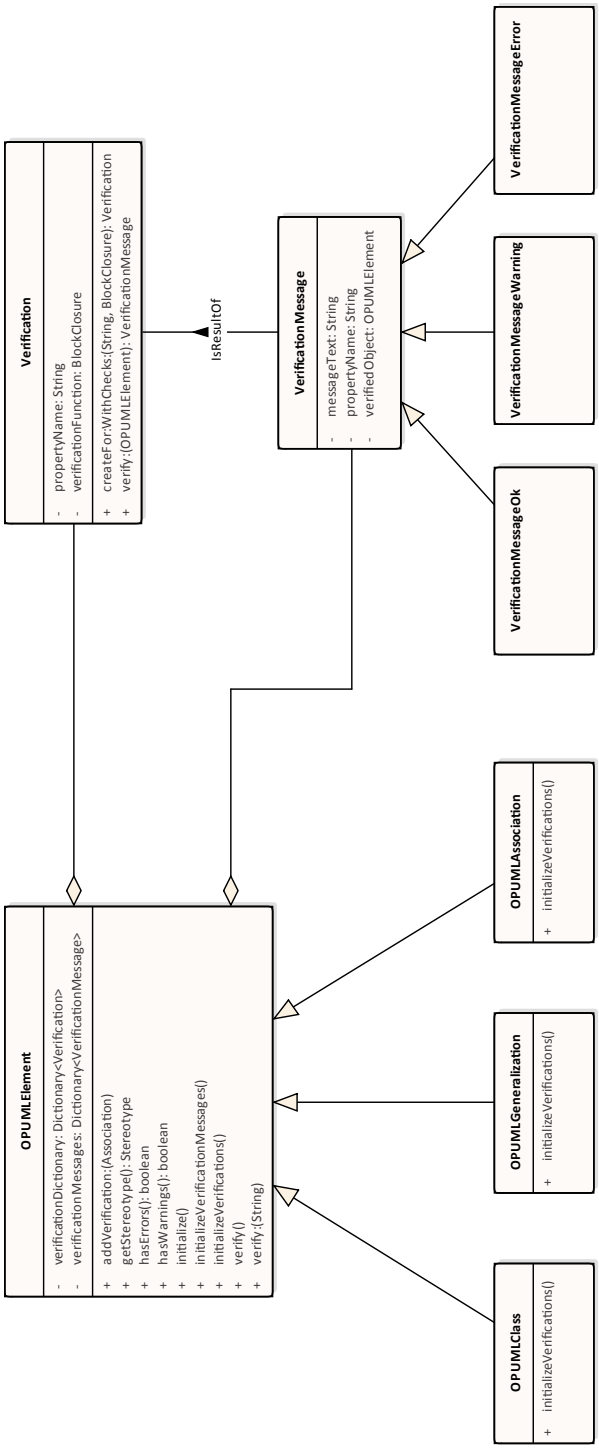

Figure 4.1: Initial design diagram

relations in the model at once.

Designing verification framework similarly to the DesignChecker tool will allow instant verification of the object, because each tim the model changes appropriate verification will be triggered.

4.2 OntoUML DesignChecker

Similarly to solutions which inspired this design of verification framework, main of the responsibility lies on verified classes. In this case all those classes are subtypes of `OPUMLElement` (they will be responsible for starting and collecting results of verifications), stereotype classes and `OPUMLGeneralization`. All classes mentioned before will contain verification functions themselves.

Next five subsections describe changes that need to be done to the existing classes and new classes that need to be added to new `OPUMLVerifications` package.

First two subsections are focused on classes in `OPUMLVerifications` packages. Those classes are `Verification` and `VerificationMessage` (with its subclasses `VerificationMessageError`, `VerificationMessageWarning` and `VerificationMessageOk`).

Remaining three subsections describe all changes done to `OPUMLElement`, `OPUMLGeneralization` and stereotype classes.

4.2.1 Verification

This class encapsulates each verification function and gives them all unified interface. I will first explain variables, that this class uses to store necessary values and then continue with its methods.

4.2.1.1 Verification variables

Here is list of `Verification` variables:

`propertyName` *Instance variable*, contains name of verified property.

`verificationFunction` *Instance variable*, contains function responsible for verification itself, usually in form of `BlockClosure` (lambda function).

4.2.1.2 Verification methods

Here is list of `Verification` methods (excluding simple getters and setters for instance variables):

`createfor:withChecks:` *Class method*, that acts as constructor. It takes two parameters property name and verification function,

and returns new instance with parameters saved in its variables.

`verify:` *Instance method*, takes verified element (entity, relation or generalization) as parameter, applies stored `verificationFunction` on it and returns result.

4.2.2 VerificationMessage

This class is used to represent both positive and negative results of verifications. It stores reference to the verified object, name of verified property and short text that describes broken rule (if there is one).

`VerificationMessage` doesn't have any special methods, apart from getters, setters and constructor that sets all three instance variables.

`VerificationMessage` has three subclasses: `VerificationMessageError`, `VerificationMessageWarning` and `VerificationMessageOk`. Those subclasses are used to represent result and severity of the problem (Warning, Error), in case of negative result.

4.2.3 Changes in OPUMLElement

To implement new verification framework there will be changes to both variables and methods of `OPUMLElement` class.

In addition to changes listed bellow, any subclass of `OPUMLElement` that wishes one or more of its properties to be validated, needs to add validation trigger to setter of those properties.

4.2.3.1 Changes to OPUMLElement variables

`verificationDictionary` *New instance variable*, that contains dictionary with property names as keys and verifications that can be applied to this element as values.

`verificationMessages` *New instance variable*, that contains dictionary with property names as keys and verification results (that are represented by `VerificationMessage`) as values.

4.2.3.2 Changes to OPUMLElement methods

Here is list of all changed and newly created methods (excluding simple getters and setters for new variables):

`verify` *New instance method*, that applies all verifications from `verificationDictionary` on this element and saves all results into `verificationMessages` variable.

`verify:` *New instance method*, that takes property name as parameter. It searches its `verificationDictionary` for verification associated with property name from parameter, applies it to this element and save the result into `verificationMessages`.

`getStereotype` *New instance method*, that returns first stereotype that has been applied to this element.

`addVerification:` *New instance method*, that takes association (key and value) and saves it into `verificationDictionary`.

`initialize` *Edited instance method*. First it calls itself from supertype and after that it calls `initializeVerifications` and `initializeVerificationMessages`, that set initial values for variables `verificationDictionary` and `verificationMessages`.

`initializeVerifications` *New instance method*, that creates dictionary and saves it into `verificationDictionary` variable.

This method should be overridden in all subclasses that wish to be verified, and add code that will fill `verificationDictionary` with all necessary verifications.

`initializeVerificationMessages` *New instance method*, that creates new dictionary and saves it into `verificationMessages` variable.

`hasErrors` *New instance method*, returns if its instance variable `verificationMessages` contains at least one instance of `VerificationMessageError`.

`hasWarnings` *New instance method*, returns if its instance variable `verificationMessages` contains at least one instance of `VerificationMessageWarning`.

4.2.4 Changes in OPUMLGeneralization

Because generalisations aren't stereotyped in OntoUML, it is necessary to implement their verifications straight into them.

For this reason initializeVerification needs to be overridden to first call itself from supertype and then call `initializeGeneralizationVerifications` method. This is new method, that will provide `OPUMLGeneralization` with all necessary verification functions.

4.2.5 Changes in stereotype classes

Stereotype classes are responsible for most of the verification functions themselves, therefore they need to implement functions for verifying subtypes, supertypes, relations sources and ends…

Most of the functions will be implemented as class methods since they will get all necessary information from elements (entities, relations or generalizations) calling them.

4.3 Problems

This design fulfilled all of the initial requirements and at the start of the implementation it seemed, that everything worked as intended. Prototype didn't show any flaws in the design and so it was decided to proceed with implementation using this design.

However shortly after first problems appeared. Not all property values were changed using their setters (for example relation ends are stored in collection, so when changing one or more ends, only getter is used). Thus it was decided to remove automatic triggers from all properties and add new interface for starting the verifications manually.

Next problem was flat structure of OntoUML stereotypes (OpenPonk prefers composition over inheritance), that lead to duplicated code. This was resolved by using Pharo traits (stateless classes containing only functional methods).

Even with those problems, it appeared that verification framework designed like this would need only few changes to work. But when I sent it to main programmer of the OpenPonk platform Peter Uhnák to ask if it can be implemented like this, I got response, that was the final nail in the proverbial coffin of this design.

All stereotypes of OntoUML entities and relations with significant part of other existing classes that would have to be edited are generated code and as such, all changes done to them would be lost at the next update of the platform.

Because of this it was decided to completely redesign the framework and remove dependencies to the data model. This would also fulfil the new requirements for modularity and for content separation that were added by Peter Uhnák.

Chapter 5

Final design

Final design was inspired by verification/validation model[4] and it was consulted with Peter Uhnák, to prevent any code being placed in the generated classes.

Verification framework is divided into these packages:

- `OntoUML-VerificationControllers`,
- `OntoUML-VerificationResults`,
- `OntoUML-VerificationMessages`,
- `OntoUML-VerificationTraits`,
- `OntoUML-Verifications`,
- `OntoUML-VerificationTests`.

Every package contains both functional code and unit tests for this code. Only exceptions are `OntoUML-Verifications` that contains only verification classes and `OntoUML-VerificationTest` that contains test for those classes. This split was done to improve implementation of `VerificationController`.

Figure 5.1 shows UML model of this diagram. This diagram focuses on general design of the verification classes and thus it is slightly simplified as it doesn't contain all Verification and StereotypeVerification subclasses.

5.0.0.1 OntoUML-VerificationControllers

This package contains `VerificationController` class (and its tests), that is used as main entrance to the verification framework. Through it all verifications are started and their results are gained.

Here is list of its methods:

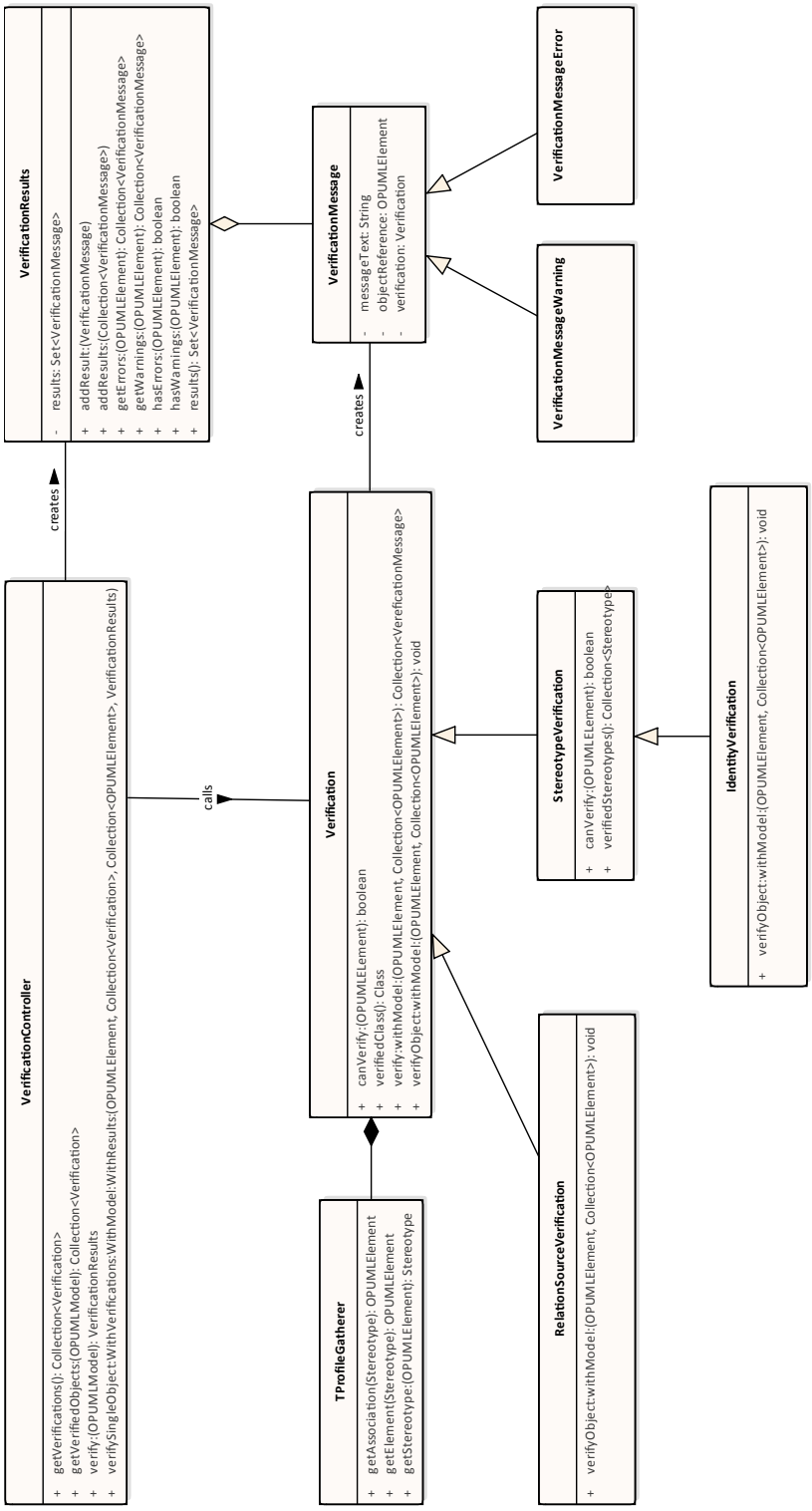

Figure 5.1: Final design diagram

`verify:`	*Instance method*, that takes `OPUMLModel` (data model) as parameter. First it creates new instance of `VerificationResults` and calls `getVerifications` to load all implemented verifications. Then it uses `getVerifiedObjects` to gain collection of all `OPUMLElements` (entities, relations and generalizations) from the model. And finally it calls `verifySingleObject:` on every single `OPUMLElement` from last step and returns filled instance of `VerificationResults`.
`verifySingleObject:`	*Instance method*, its full name is in footnote[1]. It takes these parameters: verified element (entity, relation or generalization), collection of verifications, collection of all verifiable objects from model and instance of `VerificationResults`. It applies all verifications to verified object, gathers results and saves them into `VerificationResults` from parameter.
`getVerifications`	*Instance method*, that returns collection containing all verifications classes from `OntoUML-Verifications` package.
`getVerifiedObjects`	*Instance method*, that takes verified mode as parameter and returns collection of all `OPUMLElements` (entities, relations and generalizations) from the model.

I would like to add few more details to the getVerifications method and the way it gathers all verifications from `OntoUML-Verifications` package.

One of the defining features of Pharo is the ability to access and edit its source code during runtime. This is used in `getVerifications` method to load all classes directly from their source codes in `OntoUML-Verifications` package. Verifications classes in resulting collection can have messages send to them and therefore can be instantiated and used like any other classes.

Designing verification framework in this way allows simple and very quick creation of new verifications. Each time new verification needs to be added all that user has to do is to implement this class in `OntoUML-Verifications` package and the rest will be done automatically.

Due to those design choices I have opted to split verifications and their test into two packages to make both loading and creation new verifications faster.

[1] `verifySingleObject:WithVerifications:WithModel:WithResults:`

Here you can see code from prototype implementation that was used to check if this design would be possible to implement.

```
getVerifications
    ^(RPackageOrganizer
        defaultPackageNamed:'OntoUML-Verifications')
            definedClasses.
```

5.0.0.2 OntoUML-VerificationResults

This package contains `VerificationResults` class. This class stores results of all verifications applied on one model and allows access to those results.

Here is list of its methods:

`addResult:` *Instance method*, that takes verification result (represented by `VerificationMessage`) as parameter and stores it to internal collection.

`addResults:` *Instance method*, that takes collection of verification results as parameter and adds those results into internal collection.

`results:` *Instance method*, that returns contents of internal collection.

`hasErrors:` *Instance method*, that takes `OPUMLElement` (entity, relation or generalization) as parameter and returns if there is at least one `VerificationMessageError` with reference to `OPUMLElement` from parameter in internal collection.

`hasWarnings:` *Instance method*, that takes `OPUMLElement` (entity, relation or generalization) as parameter and returns if there is at least one `VerificationMessageWarning` with reference to `OPUMLElement` from parameter in internal collection.

`getErrors:` *Instance method*, that selects and returns all instances of `VerificationMessageError` stored inside internal collection.

`getWarnings:` *Instance method*, that selects and returns all instances of `VerificationMessageWarning` stored inside internal collection.

5.0.0.3 OntoUML-VerificationMessages

This package contains `VerificationMessage` class and its two subclasses `VerificationMessageError` and `VerificationMessageWarning`, that represent (negative) results of verifications.

`VerificationMessage` stores reference to verified object, reference to instance of the verification function and short description of the OntoUML rule that was broken.

Neither `VerificationMessageError` nor `VerificationMessageWarning` add or override any methods right now, because their main goal right now is representing severity of the problem. This was done in preparation for future implementation of verification UI.

5.0.0.4 OntoUML-VerificationTraits

This package contains trait `TProfileGatherer`, that provides utility methods for working with stereotypes. Here is list of those methods:

`getStereotype:` *Class method*, that takes `OPUMLElement` (entity or relation) as parameter and returns its stereotype.

`getElement:` *Class method*, that takes stereotype as parameter and returns `OPUMLClass` (entity) that it is applied to.

`getAssociation:` *Class method*, that takes stereotype as parameter and returns `OPUMLAssociation` (relation) that it is applied to.

This trait is commonly used through classes in verification framework, because most conditions check for stereotype of the entity/relation. I have opted for this solution, because as was said earlier both stereotype classes and `OPUMLElement` are generated code and therefore can't have verifications implemented straight into them.

5.0.0.5 OntoUML-Verifications

This package contains classes that are responsible for the verification itself. There are two abstract classes: `Verification` and `StereotypeVerification` that define interface provided by all verification classes. Other verification classes are named by the rules they verify.

`Verification` is abstract class, that acts as supertype for all other classes in this package and that defines basic interface. It consists of these methods:

`verify:` *Class method*, that has these parameters: verified object and collection of verifiable object from the model.

First it calls `canVerify:` to check if it can be applied to verified element (entity, relation or generalization). If not it returns empty collection, otherwise it creates new instance and returns result of `verifyObject:withModel:`.

`canVerify:` *Class method*, that takes verified object as parameter. It checks if class of the object from parameter and result of verifiedClass are same and returns the result.

`verifiedClass` *Class method*, that returns class that can be verified by this verification.

`verifyObject:withModel:` *Instance method*, that has these parameters: verified object and collection of verifiable object from the model.

Subclasses are responsible to override this method and add code implementing the rule check itself.

Second abstract class in the package is `StereotypeVerification`. This class expands interface defined by `Verification` class with checks for stereotypes of verified object.

Here is list of added/changed methods:

`canVerify:` *Class method*, that has been overridden. It first calls original version of `canVerify:` from its supertype, then it looks if `verifiedStereotypes` contains stereotype of verified object. In the end it applies logical and between values gained by those first two steps and returns the result.

`verifiedStereotypes` *Class method*, that returns collection of stereotypes that can be verified.

All other classes in this package are subclasses of `Verification` (either directly or as subclasses of `StereotypeVerification`). They have to override methods `verifyObject:withModel:`, `verifiedClass` and classes that have `StereotypeVerification`) as parent, have to override `verifiedStereotypes` too.

5.0.0.6 OntoUML-VerificationTests

This package contains only tests for classes from `OntoUML-Verifications` package. Those tests are named after the classes they test (`VerificationTest` for `Verification`…).

They have very similar structure as verifications themselves with one exception. On the top of the test hierarchy is `AbstractVerificationTest` instead of `VerificationTest`, and it acts as parent for all other tests.

This change had to be done due to implementation of `Verification` class. `VerificationController` calls all verifications on every object in the model and since `Verification` is abstract, its `verifiedClass` method returns *nill*. This prevents `Verification` from being applied to elements in the model.

This behaviour is suppressed by its subclasses, because they need to override `verifiedClass` method. Therefore `VerificationTest` needs to handle most tests differently and using it as parent for all other classes would cause massive duplication of code.

Part III

Implementation, documentation and testing

Chapter 6

Reference model

This chapter demonstrates newly implemented verification framework on reference model. First it explains the model itself and then continues with describing every single broken rule.

6.1 Reference model design

As you can see on Figure 6.1 reference model consist of multiple smaller submodels. Each of those models break single OntoUML rule (with exception of AllowedSupertype and AllowedSubtype rules). Together they cover all verified rules.

Following sections focus on those submodels and explain what rule was broken, and what should user do to repair it.

6.2 AllowedSupertype & AllowedSubtype model

Figure 6.2 shows *kind* "Tree" as subtype of *subkind* "Oak". This breaks both *AllowedSupertype* and *AllowedSubtype* rules, as we intuitively feel that "Oak" is more specific that tree and not the other way around.

Fixing this problem is fairly straightforward. All that needs to be done, is to reverse the generalization.

6.3 RelationSource model

Figure 6.3 shows *kind* "Bottle" as subquantity of *quantity* "Vine". This breaks *RelationSource* rule, because relation *subQuantityOf* has to have *quantity* as source.

This problem is result of using incorrect relation, and should be resolved by removing *subQuantitiyOf* relation and replacing it with *containment* relation.

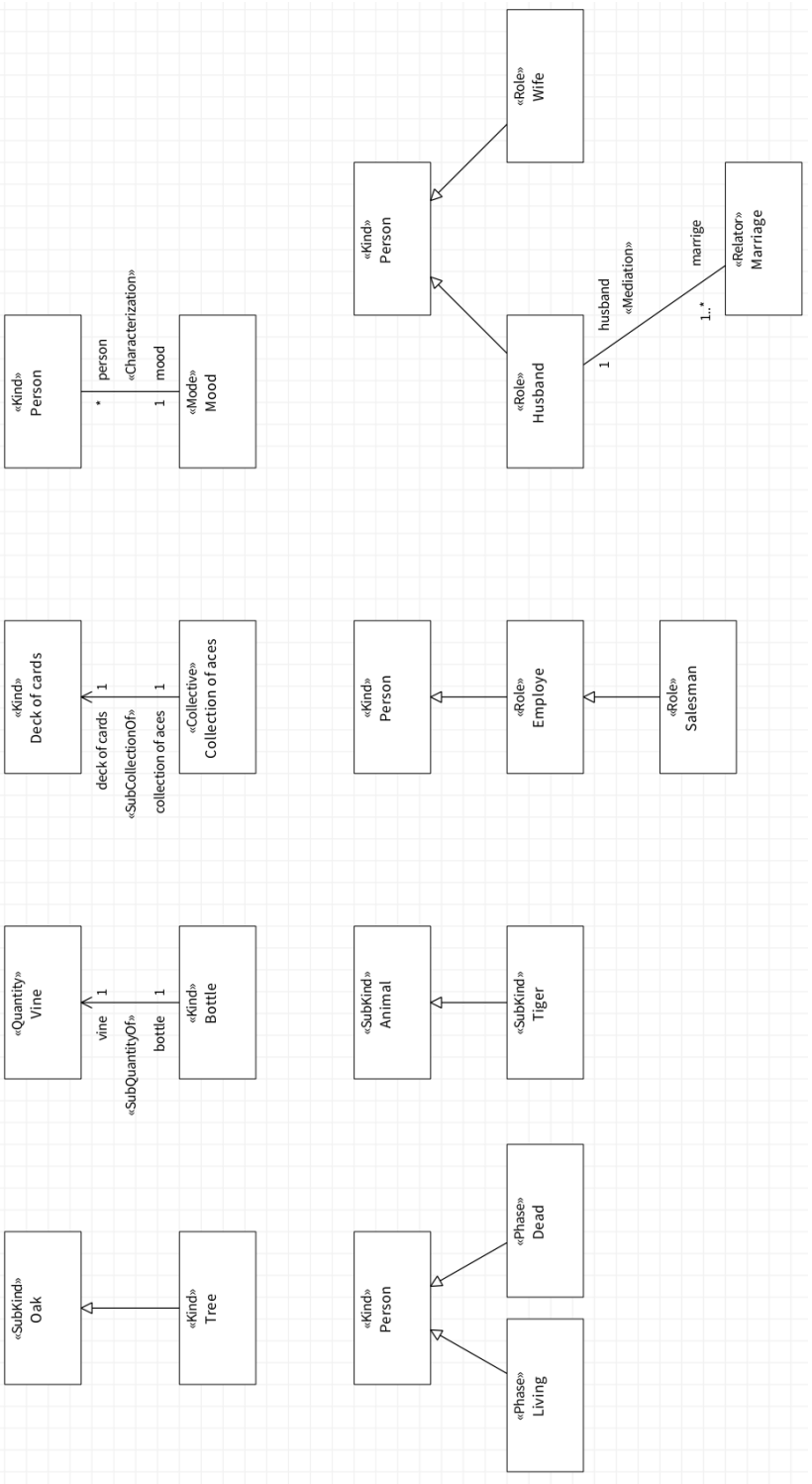

Figure 6.1: Complete reference model

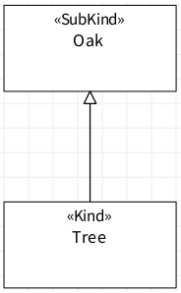

Figure 6.2: Reference model for *AllowedSupertype* and *AllowedSubtype* rules

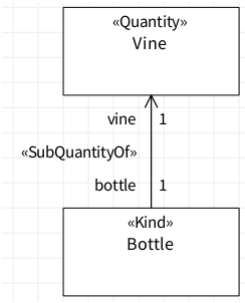

Figure 6.3: Reference model for *RelationSource* rule

6.4 RelationEnd model

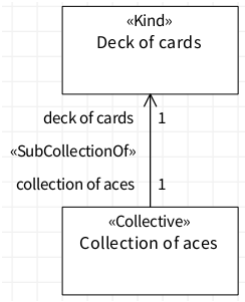

Figure 6.4: Reference model for *RelationEnd* rule

Figure 6.4 shows *Collective* "Collection of aces" as subcollection of *kind* "Deck of cards". This breaks *RelationEnd* rule, because relation *subCollectionOf* has to have *collective* as source.

Source of this problem is incorrectly selected stereotype for "Deck of cards", that should be stereotyped as *collective*.

6.5 Characterized model

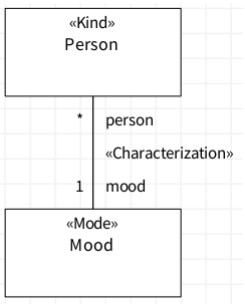

Figure 6.5: Reference model for *Characterized* rule

Figure 6.5 shows *mode* "Mood" as characterizing *kind* "Person". This seems like valid model, but closer inspection shows that *Characterized* rule has been broken.

Since *mode* is relationally dependent stereotype, cardinality of *characterization* on *kind* "Person" end has to be equal to one.

6.6 PartOfGeneralizationSet model

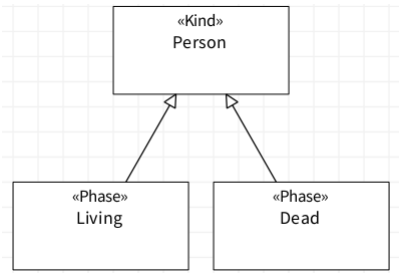

Figure 6.6: Reference model for *PartOfGeneralizationSet* rule

Figure 6.6 shows *kind* "Person" with its *phases* "Living" and "Dead". Rule that was broken here is *PartOfGeneralizationSet*.

To fix this problem user should create new generalisation set that will include both *phases* and that will be both disjoint and complete.

6.7 IdentityRequired model

Figure 6.7 shows *subkind* "Tiger" and its supertype *subkind* "Animal". There are no problems with *AllowedSupertype* and *AllowedSubtype* rules here, but

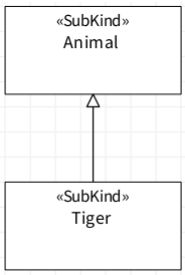

Figure 6.7: Reference model for *IdentityRequired* rule

since there isn't any entity that would provide them identity *IdentityRequired* rule is broken.

User should either change stereotype of "Animal" to *kind*, or add new entity that would provide identity of both *subkinds*.

6.8 RoleMediationDependency model

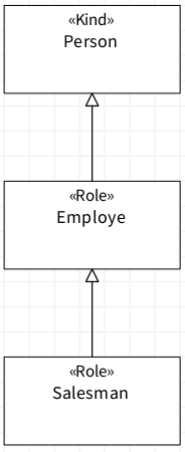

Figure 6.8: Reference model for *RoleMediationDependency* rule

Figure 6.8 shows *kind* "Person" its *role* "Employee" that is further specialised by *role* "Salesman". Since "Employee" (nor "Salesman") isn't part of any *Mediation*, *RoleMediationDependency* is broken.

Fix for this rule is bit more complicated. User will have to create another *role* called "Manager", *relator* called "Contract" and create *mediations* between "Manager", "Contract" and "Employee".

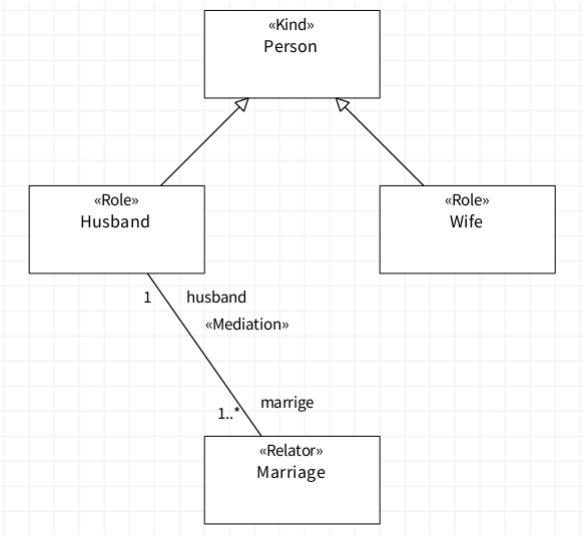

Figure 6.9: Reference model for *RelatorMediationDependency* rule

6.9 RelatorMediationDependency model

Figure 6.9 shows *kind* "Person", *relator* "Marriage" and *roles* "Husband" and "Wife". Rule that was broken here is *RelatorMediationDependency*.

This problem is caused by missing *mediation* between *relator* "Marriage" and *role* "Wife". Adding it will resolve the problem.

Chapter 7

Documentation and testing

This chapter contains information about documentation and tests for the verification framework.

7.1 Documentation

Due to nature of both OpenPonk and Pharo, it was decided to put the documentation directly to the documented code.

As you can see on the figure 7.1 each class commentary includes information about the class, its variables and its responsibilities. In addition class comments, all methods contain description explaining their purpose.

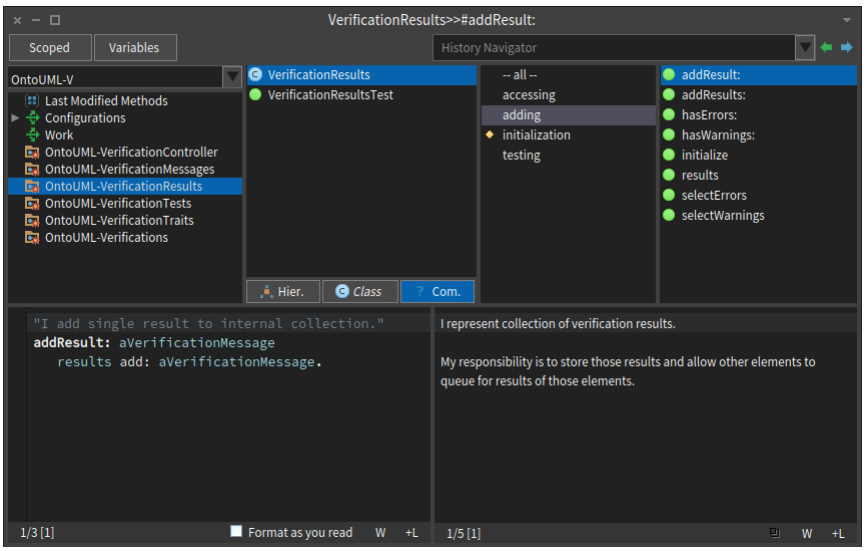

Figure 7.1: Documentation for `VerificationResults` class

7.2 Testing

Final part of verification framework are its of own unit tests. They are necessary for maintaining integrity of the framework during future updates and checking that current implementation works as intended.

Since unit test should be easy to use and quick to resolve, it was necessary to use unit test framework built in to Pharo. Fortunately this framework is very easy to use and creating new unit test was quick process for most classes.

Only exception from this were tests for `Verification` and its subclasses that are located in `OntoUML-VerificationTest`. I have run into few problems with testing subclasses of `Verification`, because it provides interface that cannot be directly checked in its tests, but that needs to be tested in its subclasses, thus leading to some duplicated code.

Figure 7.2 shows tests for `IdentityRequiredVerification` class. Those tests are located in `IdentityRequiredVerificationTest` class as its methods.

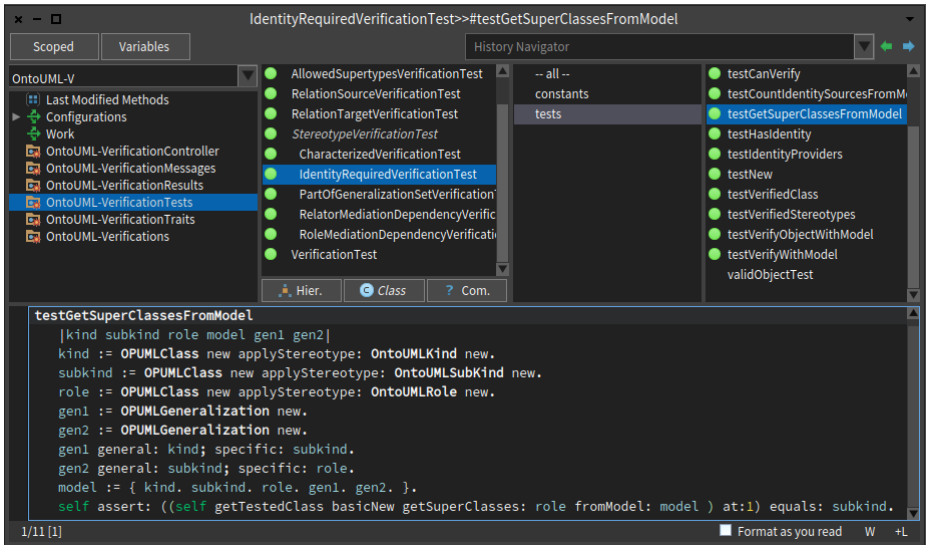

Figure 7.2: Unit tests for `IdentityRequiredVerification` class

As you can see there are only two prerequisites for creating unit tests:

- Test class has to be subclass of `TestCase` and its name has to be same as the tested class with “Test” attached to its end.
- Names of test methods have to start with “test” and then continue with name of tested method.

Fulfilling these two prerequisites links tests to methods of the tested class displaying their results like semaphore lights next to tested class and its methods and allowing user to run them by simply clicking in those results.

Conclusion

This thesis focused on analysis, design implementation and testing of new verification framework for OntoUML models, that was created for OpenPonk platform.

At the start I have focused on analysis of OntoUML language and OpenPonk platforms. I have explained basic principles of OntoUML and then listed all of its entities and relations.

Then I continued with designing verification framework. At the start I have identified rules and constraints that need to be verified. With those rules in mind, I have designed new verification framework and provided explanation and reasoning for each of its parts.

After that I have implemented the framework in the OpenPonk platform, with strong focus on possibility of future development of the framework.

Unit tests were created for all newly implemented code and I have created reference model containing at least one object breaking at least one verified rule.

In the end there was short mention of the documentation. This is part of the attached implementation.

With this I have fulfilled all goals set for this thesis and prepared foundations for future projects attached to verification. Both OntoUML language and OpenkPonk platform are constantly developed and thus there will be constant need to maintain current verification rules and add new ones based on new specifications. There is also anti-pattern domain that deals with searching for potential mistakes based on the patterns in the conceptual model.

Bibliography

1. GUIZZARDI, Giancarlo. *Ontological foundations for structural conceptual models*. Telematica Instituut / CTIT, 2005. ISBN 90-75176-81-3. PhD thesis. University of Twente.
2. GUIZZARDI, Giancarlo; FONSECA, Claudenir M.; BENEVIDES, Alessander Botti; ALMEIDA, João Paulo A.; PORELLO, Daniele; SALES, Tiago Prince. Endurant Types in Ontology-Driven Conceptual Modeling: Towards OntoUML 2.0. In: TRUJILLO, Juan C.; DAVIS, Karen C.; DU, Xiaoyong; LI, Zhanhuai; LING, Tok Wang; LI, Guoliang; LEE, Mong Li (eds.). *Conceptual Modeling*. Cham: Springer International Publishing, 2018, pp. 136–150. ISBN 978-3-030-00847-5.
3. ONTOUML COMMUNITY. *OntoUML community portal* [online]. 2017 [visited on 2019-05-09]. Available from: `ontouml.org`.
4. BENEVIDES, Alessander Botti. *A Model-Based graphical editor for supporting the creation, verification and validation of OntoUML conceptual models*. 2010. Master's thesis. Universidade Federal do Espírito Santo.
5. KRÁL, Ondřej. *Ontologická analýza změnového řízení ICT projektu*. 2018. Bachelor's thesis. České vysoké učení technické.
6. CENTRE FOR CONCEPTUAL MODELLING AND IMPLEMENTATION. *OpenPonk*. 2010. Version 1.0.0. Available also from: `https://openponk.org/`.
7. PHARO COMMUNITY. *Pharo*. 2019. Version 7.0. Available also from: `https://pharo.org/download`.
8. PHARO COMUNITY. *pharo.org* [online] [visited on 2019-05-10]. Available from: `https://pharo.org/`.
9. PHARO COMMUNITY. *Pharo wiki* [online] [visited on 2019-05-10]. Available from: `https://github.com/pharo-open-documentation/pharo-wiki`.

10. PHARO COMMUNITY. *PharoCheatSheet* [online] [visited on 2019-05-10]. Available from: `http://files.pharo.org/media/pharoCheatSheet.pdf`.

11. X-CENTER CZ. *xTractor*. 2019. Version 1.2.5. Available also from: `https://www.x-center.eu/cs/`.

Appendix A

Acronyms

IDE	Integrated development environment
OS	Operating System
UI	User interface

Appendix B

Contents of enclosed CD

- `readme.txt` ..... The file with CD contents description
- `EXE` ..... The directory with executables
  - `OpenPonk` ..... Directory with OpenPonk and verification framework
  - `ReferenceModel.opp` ..... Reference model created in OpenPonk
- `TEXT` ..... The directory of source codes
  - `BT_Assignment.pdf` ..... Assignment of bachelors thesis
  - `BT_Bělohoubek_Marek_2019.pdf` ..... Bachelors thesis in PDF format
  - `BT_Latex` ..... Directory with latex source files for the thesis
    - `Figures` ..... Directory with all used figures
      - `ModelDiagrams` ..... Directory with all model diagrams
      - `OntoUML` ..... Directory with OntoUML diagram
      - `OpenPonk` ..... Directory with screenshots from OpenPonk
      - `ReferenceModel` ..... Directory with figures for ReferenceModel