Großer Beleg

Ontology Composition using a Role Modeling Approach

submitted by

Michael Pradel

born 10.03.1983 in Jena

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisor: MSc. Jakob Henriksson
Professor: Dr. rer. nat. habil. Uwe Aßmann

Submitted September 20, 2007

Abstract

There is an increasing use of ontologies to represent domain knowledge, however, no notion of ontology components has so far been established. This lack hampers partial reuse of ontologies and makes a reuse-oriented engineering approach difficult. We propose to enable component-based ontology engineering by merging ontologies with role modeling, a well-known modeling paradigm from the object-oriented software community with focus on object collaborations. Ontological role models provide an intuitive reuse unit (ontology components) and allow for a clearer and more natural way of modeling ontologies.

Zusammenfassung

Obwohl die Verbreitung von Ontologien zur Darstellung von Wissen kontinuierlich zunimmt, hat sich bis heute keine klare Definition von Ontologiekomponenten herauskristallisiert. Dies erschwert die Wiederverwendung von Teilen existierender Ontologien, so dass sich ein auf Wiederverwendung basierender Ontologieentwicklungsprozess als kompliziert erweist. In dieser Arbeit wird gezeigt, wie komponentenbasierte Ontologieentwicklung durch die Zusammenführung von Ontologien mit Rollenmodellierung, einem in der objekt-orientierten Softwareentwicklung wohlbekannten Modellierungsparadigma, ermöglicht wird. Ontologische Rollenmodelle stellen eine intuitive, wiederverwendbare Einheit dar (Ontologiekomponenten), und ermöglichen eine verständlichere und natürlichere Modellierung.

Contents

1 Introduction

Ontology = knowledge representation model

An ontology is a model used to capture and represent knowledge. It describes the concepts of a domain and relations between them. In addition, ontologies allow for reasoning, that is, infer knowledge that has not explicitly been given. Ontologies are used in artificial intelligence, software engineering, and biomedical informatics as a form of knowledge representation of the world or a part of it. The most recent use of ontologies is for the semantic web, where ontologies enhance web content with semantics. That is, they provide meaning of information aiming at better interoperability of web content.

Poor modeling

Ontologies as used today have a number of fundamental problems that have not yet been entirely resolved by research and which hamper an even more extensive use of them. First, ontologies do not provide all modeling concepts necessary to represent the world. Imagine for instance a person appearing in several independent contexts, e.g. in his family, at work, and in a sports club. How can we model his properties being a parent, researcher, and basketball player? What about properties valid in all contexts, like name or birthday? In this example, we are dealing with two different ways of abstraction. One that is bound to the identity of this person, and another depending on the context in which the person appears. These two abstractions are called *natural types* and *role types* [Sow84]. However, current ontology languages support only one of them, namely natural types represented by classes. Thus, it is impossible to distinguish classifying properties and those belonging to a certain role. Consequently, ontology engineers are forced to intermingle different aspects into one class description, and hence, cannot separate concerns.

Few reuse

Second, although ontologies allow for reuse when applied to other domains [Ass05], reuse during the ontology engineering process itself is difficult. Up to now, there is no well-established notion of a reuse unit in ontologies. The Web Ontology Language OWL [DS04] provides only very basic reuse mechanisms. In software engineering, the notion of modules and components has largely enabled reuse of existing software when building new ones. However, ontology engineering is still far away from a component-based approach.

Role modeling in OO

Role modeling is a well-known concept from object-oriented software engineering [Ree96]. The main argument is that today's class-oriented approach mixes the notions of natural types and role types. Roles provide an abstraction of a set of objects focusing on object collaborations instead on their inherent properties. Although role modeling has not yet found its way into mainstream programming languages, it has been successfully applied in different domains, for example framework development [Rie00, RG98, ADN+03, Ste00].

More natural modeling

This work introduces roles into ontologies. Making roles an ontological primitive (as for instance called for in [Ste05]) provides a number of benefits for ontology engineering. First, distinguishing role types from classes leads to more natural modeling. Reconsider the above example of a person and

its roles in different contexts. A role-based approach allows to model that a person can play one or more of the mentioned roles. Furthermore, role types can often be reused in different contexts, for instance the parent-child relationship of persons may reoccur when modeling operating system processes or tree-based data structures.

Separation of concerns

Second, roles for better separation of different aspects of a domain than a purely classes-based approach. Role types explicitly split a class definition in parts, each describing the properties of instances of that class in different contexts.

Ontological components

Third, ontological role models provide an interesting reuse abstraction—ontological components. We will show how general role models can be (re)used to describe completely different domains. Ontological role models thus enable reuse-centered ontology engineering.

Roles in ontologies

In this work, we merge the concept of roles with ontologies on a conceptual level and address issues to consider when transferring role modeling from object-orientation. Furthermore, we propose an extension of the Web Ontology Language OWL enabling modeling of ontologies based on roles. However, we argue that no additional expressiveness is needed to support roles. Instead, we provide a translational semantics, mapping our extended syntax to standard OWL. Moreover, we present a methodology for creating role-based ontologies.

Prototype

In order to validate our approach, we realized a prototypical implementation based on the Reuseware composition framework [HAJZ07] allowing to model ontologies with roles and translate them into standard OWL.

Organization

The remainder of this work is organized as follows. Chapter 2 summarizes basic knowledge about ontologies, role modeling in object-oriented software engineering, and invasive software composition. In Chapter 3, we introduce roles into ontologies and explain the semantics of our extension. The subsequent chapter describes our prototype. Chapter 5 provides a first glance on how the ideas presented here can be further explored. Finally, Chapter 6 concludes this work.

2 Preliminaries

2.1 Ontologies, Description Logics, and the Semantic Web

This work is about ontologies, one of the key technologies of the semantic web. In the following, ontologies and one of its underlying formalisms, Description Logics (DLs), are introduced. Furthermore, the vision of the semantic web is shortly presented.

Ontologies

Definition

An ontology is a model representing the concepts of a domain, their relationships and constraints. The most widely cited definition is:

> *An ontology is an explicit specification of a conceptualization.*
>
> [Gru93]

Independent of the language used to express them, most ontologies share a common set of notions: individuals, classes, and properties. They will be shortly explained in the following. We use parts of the pizza ontology [DHS+07] as an example, presented using a self-explaining graphical notation.

Individuals

Individuals are the instance level elements of an ontology. They may represent concrete objects that exist in the physical world, as well as abstract objects like numbers or words.

Classes

Individuals are abstracted using the modeling concept of a *class*. A class describes a set of individuals with common properties and is built from class expressions. Such class expressions are conditions that have to be fulfilled by any individual belonging to that class. Class expressions are built using constructors that vary depending on the specific ontology language. We graphically represent classes by boxes with gray background. Specifications inside the box represent the definition of the class (in DL syntax). Subsumption relationships between classes, i.e. supertype-subtype relationships, are denoted by arrows in a UML-like fashion.

Properties

Relations between individuals are described using *properties*. They allow to assert facts about classes and individuals. A property is a binary relation (c, d), where c is an individual, and d is an individual or an element of a concrete domain. In the first case, the property describes a relation between two individuals and is called *object property*. Object properties manifest relations between classes. Our graphical notation presents an object property as an arrow labeled with the property name that connects two classes. In contrast, a property targeted towards an element of a concrete domain, e.g. the natural numbers or the set of strings, is called *data type property*. It describes an attribute that individuals of a class possess.

Class based ontologies

An ontology is a set of statements (axioms) using the modeling concepts

described above. Today's ontologies use classes as the only abstraction of sets of individuals. In this work, a finite set of statements using the modeling concepts individual, class, and property is called a *class-based ontology*.

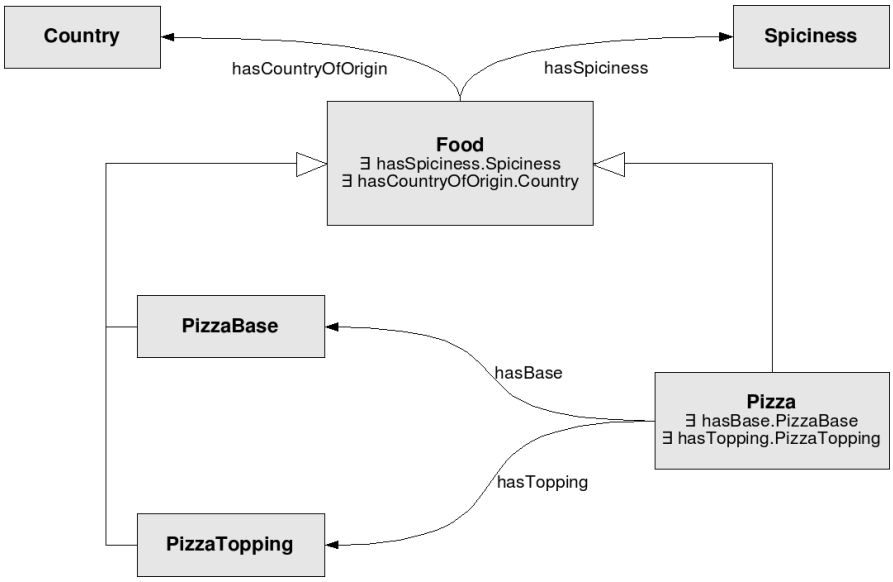

Figure 2.1: Parts of the pizza ontology [DHS+07] as class-based ontology.

Figure 2.1 shows parts of the pizza ontology as an example of a class-based ontology. Being a well-known ontology of an even better known domain, it will be used as a running example in this work.

Description Logics

DL is a family of logical knowledge representation languages that can be used to formalize ontologies. They have been developed as extensions of semantic networks [Qui67] and frames [Min74] which do not provide precise logic-based semantics [BCM+03].

Underlying formalism of ontologies

The syntax of DLs is built upon three disjoint sets: Class names, property names, and individual names. *Class expressions* (or briefly classes) are unary predicate symbols that are built using a collection of *constructors* which depend on the concrete language. *Property expressions* (or briefly properties) are binary relations, whereas individuals can be viewed as logical constants.

Syntax

The terminology used in DL partly differs from that of ontologies. Ontologies are normally called *knowledge bases* in DL jargon. Classes are referred to as *concepts*, and properties are called *roles*. However, this meaning of *role* is different from the modeling concept this work is about (see Section 2.4). In order to avoid confusion, we exclusively use the terms *ontology*, *class* and *property* to represent these notions.

Different terminologies

An ontology consists of a set of axioms. Axioms can be of the following

Axioms

types (where D, C are classes, p, q are properties, and a, b are individuals):

$$\begin{aligned} C \equiv D \quad & (class\ definition) \\ C \sqsubseteq D \quad & (class\ inclusion) \\ p \equiv q \quad & (property\ definition) \\ p \sqsubseteq q \quad & (property\ inclusion) \\ C(a) \quad & (class\ assertion) \\ p(a,b) \quad & (property\ assertion) \end{aligned}$$

TBox and ABox

The first four types of axioms describe the terminology of a domain, and are used to build the *TBox* (*terminology*). Class and property assertions form the *ABox* (*assertions*). A TBox and an ABox together constitute a knowledge base, or an ontology. The semantics of DLs are formally defined using model-based semantics.

Base language: $\mathcal{AL}$

Most DLs are built on top of the basic language $\mathcal{AL}$ (attributive language). More expressive languages, for instance the underlying DL of the Web Ontology Language OWL DL (see below), extend $\mathcal{AL}$ with more powerful constructs. In the following, we give the syntax and semantics of $\mathcal{ALC}$, a relatively simple but for our purposes sufficient extension of $\mathcal{AL}$.

$\mathcal{ALC}$ syntax

Syntax of $\mathcal{ALC}$ class terms Let N_C and N_P be disjoint sets of *class names* and *property names*. The set of $\mathcal{ALC}$ class terms is defined inductively as follows:

- Each class name $A \in N_C$ is an $\mathcal{ALC}$ class term.
- $\top$, $\bot$ are $\mathcal{ALC}$ class terms (called *top* and *bottom*).
- Let C, D be $\mathcal{ALC}$ class terms, and let $p \in N_P$. Then, the following are also $\mathcal{ALC}$ class terms:

$$\begin{aligned} \neg C \quad & (negation) \\ C \sqcap D \quad & (intersection) \\ C \sqcup D \quad & (union) \\ \forall p.C \quad & (value\ restriction) \\ \exists p.C \quad & (existential\ restriction) \end{aligned}$$

$\mathcal{ALC}$ semantics

The semantics of the class constructors are formally defined using interpretations.

Semantics of $\mathcal{ALC}$ class terms An *interpretation* $\mathcal{I}$ consists of a non-empty *interpretation domain* $\triangle^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$ which:

- assigns to each $A \in N_C$ a subset $A^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$
- assigns to each $p \in N_P$ a binary relation $p^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$.

The interpretation function is extended to (complex) $\mathcal{ALC}$ class terms by induction:

$$\begin{aligned}
\bot^{\mathcal{I}} &:= \emptyset \\
\top^{\mathcal{I}} &:= \triangle^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} &:= \triangle^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\forall p.C)^{\mathcal{I}} &:= \{d \in \triangle^{\mathcal{I}} \mid \forall e \in \triangle^{\mathcal{I}} : (d,e) \in p^{\mathcal{I}} \rightarrow e \in C^{\mathcal{I}}\} \\
(\exists p.C)^{\mathcal{I}} &:= \{d \in \triangle^{\mathcal{I}} \mid \exists e \in \triangle^{\mathcal{I}} : (d,e) \in p^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}
\end{aligned}$$

Reasoning in Ontologies

Advantage: Reasoning

The main advantage of DLs compared to other knowledge representation formalisms is that they allow for reasoning. This allows for the deduction of implicit knowledge from explicitly encoded knowledge. Furthermore, reasoning allows to detect contradictions in the way knowledge is modeled and to check whether it is possible to fulfill all the specified constraints.

TBox level

Typical reasoning tasks on the TBox level are:

- *Subsumption* Check if a class description C is subsumed by a class description D (or short $C \sqsubseteq D$). That is, if all individuals of C must also belong to D. Subsumption reasoning can be used to deduce the class hierarchy of a set of classes.
- *Equivalence* Check if two class descriptions C and D are equivalent (or short $C \equiv D$). Equivalent classes share the same semantics but may be syntactically different.
- *Satisfiability* Check if a class description is satisfiable with respect to a TBox. That is, if there may be individuals belonging to that class.

ABox level

Ontologies including individuals, i.e. a non-empty ABox, allow to solve the following reasoning problems:

- *Instance problem* Test if a given individual belongs to a class.
- *Realization problem* Find the most specific classes an individual belongs to.
- *Consistency* Check if there is an interpretation of the ontology that satisfies its TBox and ABox.

A more detailed description of reasoning in DLs can be found in [BCM+03].

OWL and the Semantic Web

Web Ontology Language OWL

The Web Ontology Language OWL [DS04] is an ontology language recommended by the World Wide Web Consortium[1] with an XML-based concrete syntax. Other concrete syntaxes, e.g. Manchester OWL syntax [HDG+06] are also available. OWL is constructed to support applications that process information on the web. It is a family of languages with three members differing in their expressiveness:

[1] http://www.w3.org/

- *OWL Lite* is the least expressive OWL sublanguage but has the lowest complexity for reasoning tasks. It is designed to construct classification hierarchies with simple constraints.
- *OWL DL* provides maximum expressiveness while possessing properties of computational completeness and decidability. That is, all conclusions will be computed and the computation will finish in finite time.
- *OWL Full* offers even more expressiveness than OWL DL but does not guarantee computational completeness and is undecidable.

Semantic web

One of the main application areas of OWL is the semantic web [BLHL01]. The main idea is to extend existing web content with semantic data such that it becomes processible by software agents. This vision is largely promoted by Tim Berners-Lee:

> *I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize.*
>
> [BLF01]

Multitude of ontologies

The content of the semantic web is annotated with metadata, e.g. by using terms defined in an ontology. A natural question is whether this can be achieved by one comprehensive consistent global ontology. While this might be wonderful in order to arrive at maximum interoperability between web content, it is not feasible and probably also not desirable. [MW04] analyzes three main problems encountered on the way to global consistency: the process of achieving consistency demanding very large efforts, the high costs of being consistent, and the serious costs of maintaining consistency over time. These problems have lead to a multitude of small ontologies covering different and partly overlapping domains.

2.2 Reuse in Ontologies

Dream: Component-based ontology engineering

As the use of ontologies increases and the ontologies themselves grow larger, the need to construct ontologies in a reuse-centered manner is becoming more and more important. Besides resulting in ontologies that are easier to manage and understand, the main goal is to construct ontologies (at least partly) from existing building blocks, similarly to component-based software [Szy02].

OWL's linking mechanism

The Web Ontology Language OWL mainly provides two mechanisms to make use of already defined ontologies. First, *linking* allows to loosely reference distributed web content and other ontologies using URIs. While the linking mechanism is convenient from a modeling perspective, it is semantically not well-defined. There is no guarantee that the referenced ontology or web content exists. Furthermore, the usually referenced component, an ontology class, is relatively small and often hard to detach from the surrounding ontology in a semantically well-defined way. Consequently, a full

ontology import is often required since it is unclear which other classes the referenced class depends on.

OWL's import mechanism

Second, OWL provides an *owl:imports* construct which syntactically includes the complete referenced ontology into the importing ontology. It can only handle complete ontologies and does not allow for partial reuse. The fact that *owl:imports* enforces a global interpretation of importing and imported ontologies can lead to inconsistencies in the resulting ontology due to conflicting modeling axioms. Overall, OWL seems to be inflexible in the kind of reuse provided, especially regarding the granularity of components.

Modular ontologies

The need to modularize ontologies is also considered on the level of the underlying DLs. One proposal is a logic-based notation that allows to distinguish between local and external signatures of a TBox [GHKS07]. Symbols in the external signature are assumed to be defined in another ontology. This distinction permits conservative extensions of TBoxes, that is, extensions that do not introduce new consequences in the original ontology. Another approach is semantic import (as opposed to syntactic import with *owl:imports*) [PSZ06]. Semantic import allows importing partial ontologies and additionally enforcing the existence of any referred external ontologies or ontology elements (classes, properties, individuals) by the notion of ontology spaces. Other approaches towards modular ontologies that focus less on reuse but rather on dealing with terminological clashes when combining ontology modules and selective knowledge sharing are summarized in [Bao06].

2.3 Ontology Aligning, Mapping, and Merging

Ontology reconciliation

Numerous applications require to relate and combine independently developed ontologies. For instance, a search engine for the semantic web needs to relate and combine the ontologies used for annotating the searched web content. During this process a number of problems may arise (see [VJBCS97, Kle01] for a categorization of typical problems into an ontology mismatch classification). Research has found a number of solutions to overcome these problems. This section presents ontology aligning, mapping, and merging and gives a short overview of existing tools to reconcile ontologies.

Alignment

Ontology alignment is the process of discovering correspondences between two source ontologies. [dBEF+06] proposes to distinguish the large number of algorithms for this task in two dimensions. First, the level of a knowledge base worked on allows to distinguish schema-based and instance-based approaches. While schema-based approaches concentrate on classes and relations of an ontology, instance-based ones consider its individuals. Second, algorithms can be distinguished regarding their locality. Element-level approaches compare one particular class or relation with another one. In contrast, structural-level ones compare for instance the inheritance hierarchies of two ontologies.

Mapping

The outcome of ontology alignment is a specification of the semantic overlap between two ontologies called an *ontology mapping*. Mapping approaches differ in the way of representing mappings. Since these are not the main concern of this work, the reader is referred to the survey of mapping approaches given in [dBEF+06].

Merging

Ontology mapping separates between the source ontologies and the mappings between them. In contrast, the creation of a new ontology containing

all information of two or more source ontologies in a coherent and consistent way is called *ontology merging*. In general, two ways of merging are possible. On the one hand, all elements from the source ontologies can be copied into the resulting one. This approach is for instance realized in the PROMPT tools [Noy04]. On the other hand, the resulting ontology can import the source ontologies, e.g. using the OWL *import* construct, and merge them with bridging axioms without changing them. OntoMerge [DMQ05] is an example for this approach.

iPROMPT

The PROMPT tools use a knowledge model based on the frame-based knowledge model underlying the ontology editor Protégé [KFNM04]. This model is general enough to be compatible with DL-based ontology languages like OWL DL (a version of PROMPT for Protégé-OWL is also available). One of these tools, iPROMPT, realizes an element-level approach helping to merge elements from two ontologies in a local context. At the beginning, an initial list of matchings is created based on class names. The user can choose to trigger one of the operations proposed by iPROMPT or specify the desired operation directly. Afterwards, the tool performs the operation, e.g. merging two classes into a new one or simply copying a class into the resulting ontology, and computes its consequences. If necessary, a set of conflicts is presented to the user, for instance because some referenced classes are not yet in the new ontology. Based on the results of the operation, the list of suggestions is updated and the process reiterates.

ANCHORPROMPT

While iPROMPT considers only the local context of ontology elements, ANCHORPROMPT uses a set of heuristics to analyze non-local context. It takes as input a set of pairs of similar classes (each pair consisting of one class from each ontology), called anchors, usually given by the user. ANCHORPROMPT compares two pairs by calculating similarity scores of the elements connecting the anchors. Based on the idea that classes between the given anchors are likely to be similar as well, a number of new matching suggestions is computed. A detailed description of the algorithm is given in [Noy04].

SAMBO

Another interesting approach is SAMBO [LT05]. It provides a framework for aligning ontologies that allows to combine different strategies. The tool includes strategies based on linguistic matching, structure-based strategies, constraint-based approaches, instance-based strategies, and the use of auxiliary information like dictionaries and thesauri. Unfortunately, the tool is not publically available such that no practical tests could be performed.

2.4 Role Modeling

Two ways of abstraction

Roles are a modeling concept for object-oriented software that focuses on objects and how they collaborate. The main idea is that objects can be abstracted in two ways [Ree96]. One abstraction regards their properties and classifies them accordingly, yielding classes, the major modeling concept used in most object-oriented programming languages. The other abstraction analyzes how objects collaborate in order to achieve a specific goal. This yields *roles* and *role models*.

Natural types vs. role types

On a slightly more philosophical level, one can distinguish between *natural types* and *role types*. Natural types can be defined as follows [Sow84]:

> A natural type is a *semantically rigid* and *non-founded* type insofar as an entity that has the type cannot stop being of the type

without loosing its identity and does not depend on any collaboration.

Obviously, classes are natural types. In contrast, role types are characterized by the following definition:

> A role type is a *founded* and *semantically non-rigid* type insofar as it characterizes an entity by some role it plays in relationship to another entity or other entities, and if left, does not give up identity of entities.

Classes are good abstractions to categorize objects by their properties. However, objects not only exist, they interact with each other in different contexts to achieve certain (common) goals, in other words, they play roles. For instance, a person may play the roles *baker* early in the morning, and *father* in the afternoon. Both are completely independent from each other and each depends on a certain context (for example the bakery and his son).

Definitions

Research on roles in object-oriented software has resulted in a multitude of (partly opposing) definitions [Ree96, Ste05]. The following definitions are based on [Rie00]:

> **Definition: Role**
> A role describes the behavior of an object in a certain context.

An object is said to play a role. It may play multiple roles at the same time and change its roles during its lifetime. The behavior of an object depends on a collaboration, and thus, a role never occurs alone but with at least one other role.

A role type provides an abstraction from similar roles:

> **Definition: Role type**
> A role type is a type that characterizes a set of roles with similar behavior.

Composing all roles of an object completely defines its behavior. Similarly to classes, role types are a modeling concept to abstract from a number of objects. However, a role type concentrates on the collaboration of objects with other objects. A specific collaboration is called an object collaboration task:

> **Definition: Object collaboration task**
> An object collaboration task is a collaboration of objects that play roles to reach a common purpose.

In other words, an object collaboration task defines the context in which objects playing roles interact. This leads to a new abstraction not available for classes:

> **Definition: Role model**
> A role model is a set of relating role types. It characterizes a set of similar object collaboration tasks.

Role models allow to abstract from a set of objects according to their collaborations. Thus, they provide a reusable abstraction that focuses on only

one context. In contrast, a class model presents multiple classes, each having several role types, and their relationships. Class models intermingle completely independent concerns, and thus, hamper reuse of a single concern. In constrast, role models allow for separation of concerns.

Role types can be connected by two kinds of relationships. On the one hand, there are the usual object relationships, i.e. association, aggregation and composition. On the other hand, one wants to model certain constraints between role types. For instance a person playing the father role cannot play the role of its son at the same time. Those constraints can be expressed by role constraints:

> **Definition: Role constraints**
> A role constraint is a value from the set *role-implied*, *role-equivalent*, *role-prohibited*, *role-don't care*. For every pair of role types, one such value is defined.

Role constraints are scoped by an object collaboration task. They only constrain the roles of objects inside this task. For example, every father also has a father, and therefor plays the role of a son. However, that is another context and hence, the role constraint prohibiting one object to play the roles of father and son at the same time does not apply.

The role constraints have the following meaning for a pair of role types (R, S):

- *Role-implied*. An object playing a role defined by role type R is always capable of playing a role defined by role type S.
- *Role-equivalent*. An object playing a role defined by role type R is always capable of playing a role defined by role type S, and vice versa. Thus, both roles imply each other.
- *Role-prohibited*. An object playing a role defined by role type R is not allowed to play a role defined by role type S in the same object collaboration task.
- *Role-don't care*. There is no constraint concerning objects playing roles defined by role types R and S.

Relation to class modeling

It is important to note that role modeling is no replacement for class modeling but an additional modeling concept. It is used to model object-oriented software with different abstractions than those of class modeling and it is a natural question how to relate both to each other. Since objects always belong to one class and may play several roles, it follows that a class may contain several role types. The composition of all its role types completely describes the behavior of instances of the class. Furthermore, a class model may consists of a set of role models. The role model composition leads to the class model.

Benefits

What is the benefit of role modeling? Its advantages in framework design and usage have been analyzed in [Ric00, RG98]. First of all, role models reduce complexity of classes and object collaborations by focusing on one specific aspect. Furthermore, role modeling offers a new dimension of separation of concern that is not provided by classes [OT00]. This leads to a more flexible design and encourages reuse. Moreover, applying role modeling at the framework border facilitates instantiating a framework since role

models help to describe what requirements a framework instantiator has to fulfill.

Roles in practice

Finally, the question may be asked how existing object-oriented software development techniques support roles. For modeling, a concept called *collaborations* which is part of the UML [OMG05] provides means to represent role types and role models in UML diagrams. For the implementation, there is unfortunately no object-oriented programming language directly supporting roles to the knowledge of the author. However, role types can be implemented with classes [BRSW97], interfaces [Ste01, Rie00] or using the language concept of *mixins* [SB02].

2.5 Invasive Software Composition

ISC

Invasive software composition (ISC) [Ass03] is a technique to construct software in a reuse-centered way. The idea of component software is to create applications from existing building blocks in order to maximize reuse. Components can be regarded as ready to use, unchangeable blackboxes [Szy02]. ISC goes beyond the ideas of component software by focusing on the composition and treating components as *grayboxes* that can be adapted to reuse contexts.

Graybox component model

A *composition system* is made up of a *component model*, a *composition technique* and a *composition language*. Invasive software composition employs a graybox component model. A component is regarded as a set of fragments forming a *fragment box*. A *fragment* is an arbitrary piece of source code ranging from complete packages down to individual statements. Fragments contain variation points, called *hooks*, that form the composition interface of the fragment boxes. *Implicit hooks* are positions or program elements contained in every component by definition of the programming language, for instance a method entry in Java. In contrast, *declared hooks* have to be explicitly specified by the component creator.

Invasive composers

The composition technique of invasive software composition is based on *invasive composers*. They transform one or more hooks of a component into other program elements in order to adapt the component to a new context. Composers are built from a set of basic *composition operators*, for instance *bind* which replaces a hook with a value, or *extend* that adds a value to a list hook.

Reflective invasive composition

The way components are connected and which adaptations are made must be specified in a composition language. Invasive software composition proposes to use the same language already used for the components and define transformations in a reflective way with metaprogramming. The Compost library [COM07] exemplifies invasive software composition for Java using static metaprogramming.

A unifying approach

Beyond adding the idea of graybox composition, ISC unifies a large number of existing software engineering concepts. For instance, architecture systems [GS94], hyperspace programming [OT00], and aspect oriented programming [KLM+97] can be explained as invasive software composition systems.

The Reuseware Composition Framework

Reuseware composition framework

While the Compost library implements ISC for Java, the Reuseware compo-

sition framework realizes its ideas for arbitrary languages [HAJZ07]. One problem area identified and targeted by Reuseware is that a large number of domain specific languages lack support for modularity and composition. Using the fact that composition has many aspects that are independent of the particular language model, the framework provides composition technology and techniques for every formal language lacking such built-in mechanisms.

Core language and reuse language

To extend a given language, one requires its metamodel, that is, a description of the language's constructs and their relations. Language metamodels can be expressed as EBNF grammars [Int96] or with meta object facilities (MOF). The idea of Reuseware is to extend a metamodel to support concepts of composition. The original language is referred to as the *core language* while the extended version is called the *reuse language*. More concretely, the context-free grammar of the core language is extended with new non-terminals that represent variation points. Variation points can be *hooks* and *slots*. While slots can be filled exactly one time, a hook may be extended as often as necessary. Fragments written in the reuse language can then be treated as components and may become subject of composition.

Composition operators

Reuseware supports the primitive composition operators *bind*, *extend*, *prepend* and *append*. Several of these operators can be grouped to a complex composition operator. It addresses several variation points belonging together by executing the primitive operators in sequence.

Tools

The Reuseware composition framework consists of two main components, where one of them is responsible for defining and extending languages. It uses metamodels based on the Eclipse Modeling Framework (EMF) to represent languages. EBNF-like grammars can be used as well by an internal mapping to a metamodel. The other component consists of a language to define fragment compositions and tooling for the composition execution. The framework can be used via an Eclipse plugin that allows to define core languages and reuse languages. From these definitions new plugins are generated that help in writing fragments, composition programs and actually execute compositions.

Applications

The framework has been successfully applied to a number of languages, including Xcerpt [BS03] and Java-, a subset of the Java programming language. In this work, Reuseware is used to add composition support to the Web Ontology Language OWL.

3 Role Modeling for Ontologies

No roles in ontologies

Ontologies describe the concepts of a domain and relations between them. Even if an ontology does not include individuals, they are always kept in mind when building it. The ontology creation process is in large parts driven by abstracting individuals and describing them using constructs of the ontology language. Today's ontologies are built from classes, that is, a set of individuals is abstracted by a class. However, classes turn out to be inappropriate to express a different way of abstraction—role types (see Section 2.4 for the distinction of natural types and role types).

Example

Consider for instance modeling a set of persons of whom some may play the role of a parent or a child in certain contexts. *Person*, *Parent*, and *Child* are obviously concepts of the domain under consideration and should consequently each be described as a class. The interesting question is how to relate them. Not relating them at all (Figure 3.1 (i)) is not considered a reasonable choice since there without a doubt exists a relationship between them. Defining *Person* as subclass of *Parent* (or *Child*) as shown in Figure 3.1 (ii) seems not meaningful since not all persons are parents (or children). A possible solution is using the subsumption relation the other way around by making *Parent* and *Child* specializations of *Person* (Figure 3.1 (iii)). Now, imagine the introduction of a class *Animal* in our ontology that is disjoint from *Person*. Since animals may be parents as well, we are tempted to model *Parent* as subclass of *Animal*. However, this results in *Parent* being unsatisfiable since individuals belonging to it would have to be persons and animals which are disjoint.

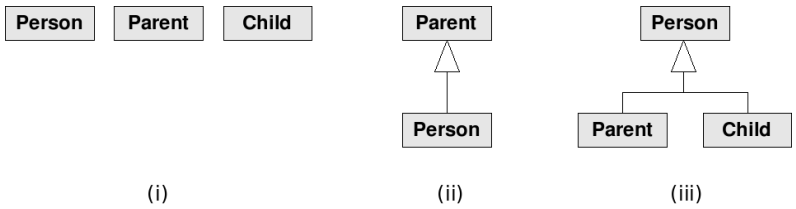

Figure 3.1: Different ways to relate the classes *Person*, *Parent*, and *Child*.

Roles as ontological primitive

In more general terms, the lack of roles as ontological primitive complicates an appropriate description of situations where individuals of a certain class may appear in different roles, which in turn may be played by individuals of other classes as well. This lack leads to poor modeling since ontology engineers are forced to (mis)use classes to express role types, and in the extreme case (as shown above) my have severe unintended implications. Furthermore, class-only modeling forces the modeler to intermingle the inherent properties of a class and those belonging to a specific role. This leads to complex classes that can be difficult to split up afterwards. In order to overcome these problems, we propose to introduce roles as a primitive of ontology languages in Section 3.1. Beyond allowing for more natural and

flexible modeling, using roles in ontologies permits to explicitly separate different concerns of a class by associating each with a role type.

Ontology components

As shown in Section 2.2 the reuse mechanisms provided by the Web Ontology Language OWL [DS04] do not satisfy all needs. In particular, it is impossible to reuse parts of an ontology in a semantically well-defined way. Approaches towards modular ontologies (also see Section 2.2) are promising, however, none of them achieves providing an intuitive notion of ontological components. Classes often are an inappropriate reuse unit since they combine different concerns and have various dependencies to other classes. We believe that ontological role models are an interesting reuse abstraction and we show how to employ them as ontology components in Section 3.2. This permits to build ontologies (at least partly) from reusable building blocks and leads to a more efficient ontology engineering process.

Translational semantics

The major advantage of ontologies compared to other knowledge representation mechanisms is its formal semantics allowing for reasoning (see Section 2.1). In Section 3.3, we give a formal semantics of the role modeling extensions proposed in this chapter by translating the extended OWL syntax into standard OWL. This translational semantics allows to benefit from modeling ontologies with roles and at the same time permits to use the large number of existing tools for ontology creation, management, and reasoning based on OWL.

Comparison with OO

Ontologies and the world of object-oriented software engineering have several differences (see [AZW06] for a discussion of the relation between them). Consequently, the transfer of role modeling into ontologies is not straightforward but involves a number of issues to consider. In Section 3.4, we examine these issues and compare the traditional view of role modeling with our approach.

Role modeling - a conceptual idea

Role modeling for ontologies is a conceptual idea, and thus, independent of any concrete ontology language. We aim to define roles such that our definitions are applicable to most ontology languages. To stay as much as possible on a conceptual level, we use the very general definition of an ontology from Section 2.1 in the first two sections of this chapter. Also, we only regard those elements that are related to role modeling, e.g. we leave out annotations.

3.1 Enhancing Ontology Modeling with Roles

Problems of class-based modeling

The usage of classes as exclusive modeling concept to abstract from similar individuals has a number of deficiencies. First, classes are relatively coarse-grained, and thus, introduce more complexity than needed. This not only leads to complex classes, but also to advanced complexity of their relations. Second, ontology classes often merge properties from different contexts, that is, multiple concerns are mixed into one conceptual unit. Finally, high complexity of class relations and missing separation of concerns hamper ontology reuse and composition. Applying the traditional class-based ontology modeling hinders reuse of parts of an ontology and composition of new ontologies from existing ones.

Individuals play different roles

Using the fact that one individual usually occurs in different contexts playing different roles, it is possible to overcome the problems stated above. We propose to explicitly model roles using the additional modeling concept of a *role*. This allows to distinguish different contexts in which individuals are related to each other, and abstractly define them as *role models*. However, role

modeling is no replacement for class-based modeling. Instead, it enhances the modeling concept of a class with the more fine-grained roles. Thus, role modeling for ontologies helps to remedy the deficiencies of class-based ontologies without loosing their advantages.

Definition: Role
A role describes the properties of an individual in a certain context.

An individual is said to play a role. Since a context consists of at least one other role, a role never occurs alone, but always in relation to other roles. An individual may play multiple roles, where each role may belong to a different context.

Role types allows for better modeling

Individuals can be abstracted in two ways: regarding what they are, i.e. using classes, and regarding how they are related to other individuals, i.e. using role types. Adding the modeling concept of role types adds flexibility since an ontology designer can explicitly distinguish natural types from role types.

Definition: Role type
A role type characterizes a set of roles with common properties in a certain context.

Role types are founded types, i.e. a role type depends on other role types defining its context, and thus, never occurs alone. A class is said to *contain* a role type, if all individuals of that class can play a role of the role type.

A role type abstracts from a number of roles with equal properties, similarly to a class abstracting from individuals that share properties. In contrast to a class which usually fully describes an individual, a role type characterizes only its properties in a certain context. This allows for separation of concerns of a class, defining each in a separate role type. The composition of several role types may result in a class.

Relations between role-playing individuals

Since roles focus on the collaboration of individuals, a concept to model these collaborations, i.e. the relations between individuals, is required. This is provided by role properties.

Definition: Role property
A role property is a binary relation (a, b), where a is an individual, and b is an individual or an element of a concrete domain.

When b is a role, the property represents the relation of two individuals. If b is an element of a concrete domain, it describes an attribute of an individual. Thus, the meaning of role properties are exactly the same as those of class properties, except that they describe the relations and attributes of role types instead of classes.

Related individuals form a collaboration

Individuals are related in numerous ways. The individual relationship graph of an ontology is usually very complex. However, it can be grouped into collaborations according to the context in which relationships occur.

Definition: Individual collaboration task
An individual collaboration task is a concrete collaboration between individuals related in a common context.

Role models abstract from similar collaborations

Often, different individuals relate to each other in very similar manners. Certain collaboration patterns are found again and again in ontologies. This allows to introduce an abstraction of similar individual collaboration tasks:

Definition: Role model
Multiple role types that are related to each other in a common context are a role model. It characterizes a set of concrete collaborations between individuals whose relations follow a certain pattern.

Each role type in a role model has to be directly or indirectly related to the others. Otherwise the role model would intermingle different concerns that could be split up into multiple role models. That is, the role type relationship graph must be non-partitioned. Composing several role models, such that role types are composed to classes, may lead to a class-based ontology.

Figure 3.2 shows an example of a role model. It described the relation between something tasting and a taste. White rectangles with rounded corners denote role types. The definition of a role type is inside its rectangle (in standard DL syntax). In addition, role types are tagged with the name of their role model, e.g. *(Taste)*. Labeled arrows present binary properties between role types.

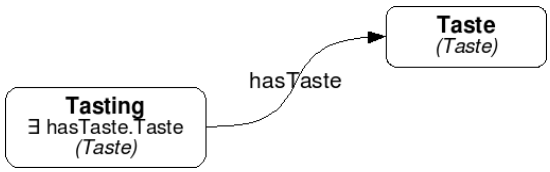

Figure 3.2: The *Taste* role model containing the two role types *Tasting* and *Taste*, as well as the role property *hasTaste*.

Role-based ontologies

Given the set of concepts defined above, we are able to enhance a class-based ontology with roles. Thus, ontology designers can abstract individuals not only as classes but also as role types, and hence, focus on their collaborations. The resulting ontologies are called *role-based*:

Definition: Role-based ontology
An ontology that is, in addition to the modeling concepts of a class-based ontology, (partly) described using role models is called a role-based ontology.

Example: Pizza ontology with roles

An example of a role-based ontology is shown in Figure 3.3. The pizza ontology from Figure 2.1 has been enhanced by two role models: *Origin* and *Taste*.[1] Originally, the *Food* class included properties describing the fact that it is tasting and that foods have a place of origin. However, these are completely independent concerns that should be modeled as such. Ontological role modeling allows to explicitly separate both concerns by splitting the *Food* class into two role types. The role models emphasize the collaborations of individuals of *Food* in different contexts and specify to which context a relation belongs.

Open role types envisage reuse

There may be role types that do not belong to any class but are included as part of a role model (image for instance Figure 3.3 without the *Origin* class). This indicates that the ontology may be extended with other classes taking the open roles. Thus, open roles allow to underspecify an ontology and explicitly point out which parts are missing. The set of open roles defines an interface of an ontology useful to combine it with other ontologies.

[1] All role models used in this work can be found in the appendix.

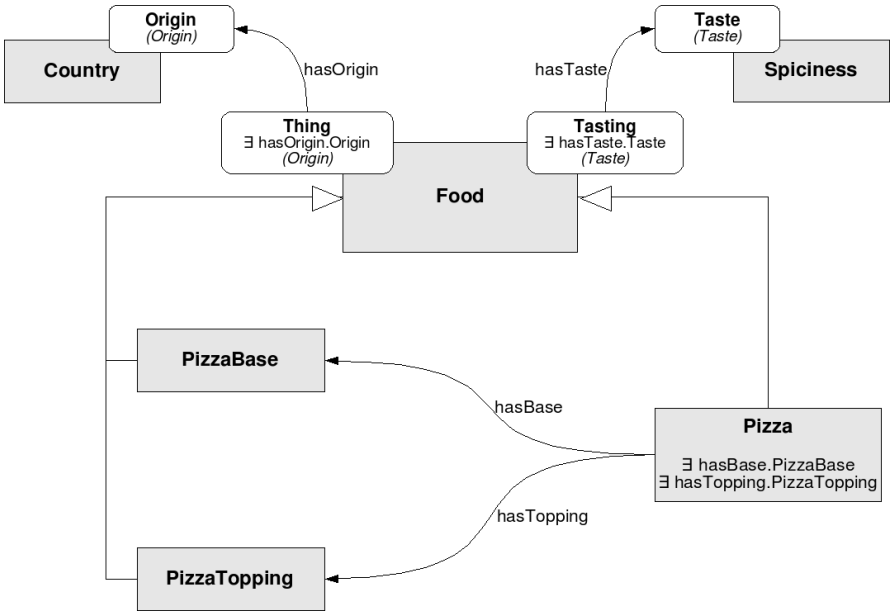

Figure 3.3: The pizza ontology enhanced by roles.

Definition: Bound role type
A role type that is contained in a class of an ontology is a bound role type.

Definition: Open role type
A role type that is not contained in any class of an ontology is an open role type.

Open role types should belong the a role model containing at least one bound role type. For a role model without any bound role type, there is no reason to include it since it has no relation to the ontology. The process of adding an open role type to a class is referred to as *binding* the role type to the class.

Summary

This accomplishes the introduction of roles as additional modeling concept for ontologies. The properties of individuals can be abstracted by role types. In contrast to classes, role types focus on collaborations of individuals and allow to split classes according to their concerns. Ontological role models abstract from recurring individual collaborations that have a similar context and allow to build role-based ontologies.

3.2 Composing Role Models to an Ontology

Reuse in ontology development

Due to the semantic web and other applications, there is a large number of ontologies covering different domains. However, few ontologies are built upon others or even built mainly from existing ontologies. A similar situation occurred in software development some decades ago and has lead to component-based software [Szy02, Ass03]. The idea to reuse existing work during ontology development is alluring. Ideally, it should be possible to construct entire ontologies from existing building blocks. However, we have

seen in Section 2.2 that the built-in mechanisms of OWL are not sufficient and an intuitive reuse abstraction for ontologies is still needed.

The open question

The problem can be summarized by asking: How can a component system for ontologies look like that enables reuse of ontology fragments when building new ontologies? This section tries to answer this question, by specifying what parts of an ontology can be reused and how to assemble them to obtain a valid ontology.

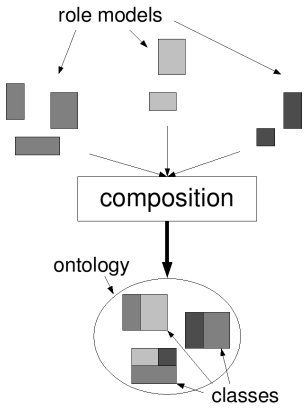

Figure 3.4: Composing role models yields an ontology.

Conceptual idea

Role models seem to be ideal candidates for ontology modules. They consist of several role types and their relationships representing one aspect of a domain. Given a role model for each aspect of a domain and composing them leads to the complete ontology (Figure 3.4 presents this idea schematically).

Methodology

We propose to compose an ontology from role models in two steps:

1. First, the main concepts of a domain have to be identified and described by classes. However, particular properties and their relationships to other classes belonging to a certain context should be left out for the moment. This first step results in a set of classes containing only inherent restrictions and properties, for example the name of date of birth of a person, or none at all. Each class represents a natural type occurring in the domain. These classes may be related via subsumption or equivalence relationships, as well as using properties that belong to individuals' identities.

2. Second, the ontology engineer considers the collaborations of individuals and searches for a role model describing their relations. In case that there is a reusable role model that describes the wanted collaboration, it is added to the ontology by binding its role types to classes. This step is repeated until the domain is described to the necessary degree.

Creating new role models

What happens if there is no role model available that describes a certain collaboration of individuals in the domain under consideration? One solution would be to add the necessary class expressions and relationships directly to the classes. However, once added, concerns are intermingled and can only be separated later with great difficulty. That is why a new role model should be created. Describing a missing concern as a role model allows to keep it separate from other concerns and furthermore, permits to reuse it later on.

Figure 3.5: First construction step of the wine ontology containing only unrelated classes without restrictions.

Example: Wine

As an example for role model composition, we build a simple ontology describing the wine domain. The result will be similar to parts of the wine ontology from the OWL guide[2], but to some extent adapted for our purposes. As a first step, the basic concepts of the domain are described by unrelated classes without any restrictions (Figure 3.5). To simplify matters, only four classes are considered here. Next, we try to find role models to represent collaborations of individuals in the wine domain. Wineries and wines are related since the former produces the latter. This relationship is very well described by the *Product* role model. It includes the role types *Product* and *Producer* as well as two others. The role model is introduced into the ontology by binding the role types to the classes *Wine* and *Winery* (Figure 3.6). The other two role types (*Consumer* and *Seller*) stay open, i.e. unbound, since they obviously do not occur in our ontology.

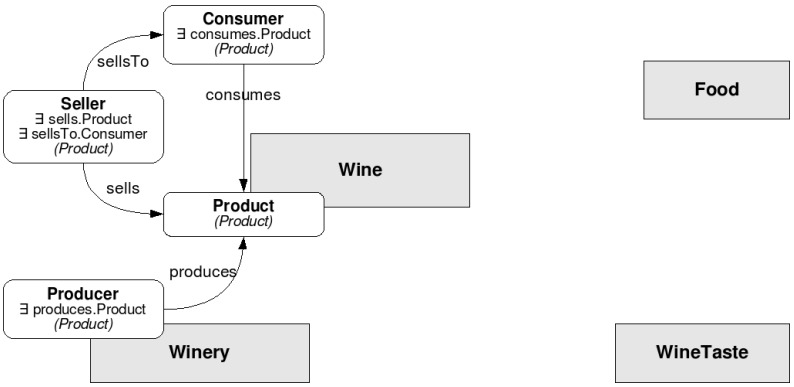

Figure 3.6: Wine ontology after adding the *Product* role model.

Now, let us consider the relationship between *Wine* and *WineTaste*. Clearly, we can reuse one of the role models already used in the pizza ontology, namely the *Taste* role model. The relation of wine and food can be described by the *Meal* role model, considering wine as a drink that goes with a meal (in that case simply food). Applying both to our ontology leads

[2] http://protege.cim3.net/file/pub/ontologies/wine/wine.owl

to Figure 3.7. This description of the wine domain shall be enough for our purposes.

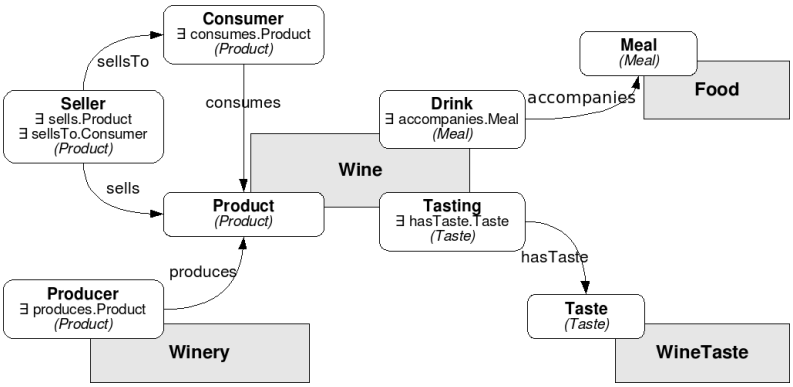

Figure 3.7: Wine ontology after adding the *Product*, *Origin*, and *Taste* role models.

Role models as ontology components

The result of our example illustrates the benefits of role model composition. Most parts of the ontology are built from existing role models that have already been used in other ontologies in this work. At the beginning of this chapter, we asked how it is possible to decompose ontologies into components. A component should be a reusable, self-contained entity with high inner cohesion and few relations to other components. Role models offer all these features. Reusability and high inner cohesion result from the fact that they represent exactly one concern of a domain. Since role models do not refer to other role models, they are self-contained. Thus, role models are a good candidate for ontology components.

Summary

This section showed how an ontology can be built from existing parts—ontological role models. A methodology for role model composition was briefly presented and illustrated with an example. The outcome is a role-based ontology built from reusable parts.

3.3 Semantics of Ontological Roles

Formalizing ontological roles

So far, role modeling for ontologies has been described on a conceptual level. To benefit from the advantages of formally defined syntax and semantics of DLs it is necessary to extend these formalism to roles. This section presents $\mathcal{ALC}^{\mathcal{R}}$, a syntactical extension of the $\mathcal{ALC}$ DL described in Section 2.1.

We believe that ontological role modeling requires no additional expressiveness of the ontology language. Instead, we extend the language with additional syntax to facilitate reuse-based modeling since $\mathcal{ALC}$, as well as OWL, lack built-in constructs supporting component-based ontology engineering. The fact that we do not add expressiveness is reflected by the use of translational semantics. That is, our syntax extensions can be translated into standard DL (or standard OWL). The semantics of roles in ontologies is thus automatically given by the usual DL semantics.

Syntax extensions

The following language elements are added to the syntax:

- Role type terms
- Role models

- Role type bindings
- Role type assertions

In the following, we will formally define the syntax of these extensions as $\mathcal{ALC}^{\mathcal{R}}$ and the translation into $\mathcal{ALC}$. Afterwards, the definitions and the translation are illustrated with examples using the more readable Manchester OWL syntax [HDG+06].

Adding Roles to the $\mathcal{ALC}$ Description Logic

Role type terms

The following definitions refer to the syntax and semantics of $\mathcal{ALC}$ from Section 2.1. At first, we define role type terms usable to define role types:

Definition: $\mathcal{ALC}^{\mathcal{R}}$ role type terms
Let N_R and N_{RP} be sets of *role type names* and *role property names*, such that N_P, N_C, N_R, and N_{RP} are pairwise disjoint. The set of role type terms is defined inductively as follows:

- Each role type name $R_A \in N_R$ is an $\mathcal{ALC}^{\mathcal{R}}$ role type term.
- $\top, \bot$ are $\mathcal{ALC}^{\mathcal{R}}$ role type terms (called *top* and *bottom*).
- Let R_C, R_D be $\mathcal{ALC}^{\mathcal{R}}$ role type terms, and let $p_R \in N_{RP}$. Then, the following are also $\mathcal{ALC}^{\mathcal{R}}$ role type terms:

$$\begin{aligned} \neg R_C \quad & (negation) \\ R_C \sqcap R_D \quad & (intersection) \\ R_C \sqcup R_D \quad & (union) \\ \forall p_R.R_C \quad & (value\ restriction) \\ \exists p_R.R_C \quad & (existential\ restriction) \end{aligned}$$

Role models

A set of role types and role properties relating them yields a role model. Formally, we define them as follows:

Definition: $\mathcal{ALC}^{\mathcal{R}}$ role models
Let N_M be a set of *role model names*, let $R_A \in N_R$ be a role type name, and let R_C be a role type term. An $\mathcal{ALC}^{\mathcal{R}}$ role model $\mathcal{M}$ with $\mathcal{M} \in N_M$ is a set of axioms of the form $R_A \equiv R_C$.

Role type terms occurring on the left hand side of such an axiom are called *defined role types*. All other role type names occurring in a role model are referred to as *primitive role types*.

Role binding

Of course, it is necessary to combine role types of $\mathcal{ALC}^{\mathcal{R}}$ with usual classes from $\mathcal{ALC}$ to describe that individuals of a class may play a certain role. This connection is realized with the following axiom:

Definition: Role binding axiom
Let C be an $\mathcal{ALC}$ class term and let R be an $\mathcal{ALC}^{\mathcal{R}}$ role type term. The following can be used as an axiom and is called *role binding*:

$$R \rhd C$$

Role assertions

Similarly to the class assertion axioms, we need an axiom to assert individuals to be in the extension of a role type:

Definition: Role assertion axiom
Let R be a role type and let a be an individual name. The following can be used as an axiom and is called *role assertion*:

$$R(a)$$

Using the above definitions, we can define role-based ontologies: Role-based ontology

Definition: Role-based ontology
An $\mathcal{ALC}^{\mathcal{R}}$ ontology is a pair $(\mathcal{T} \cup \mathcal{T}_R, \mathcal{A} \cup \mathcal{A}_R)$, where $\mathcal{T}$ is an $\mathcal{ALC}$ TBox and $\mathcal{A}$ is an $\mathcal{ALC}$ ABox, and:

- $\mathcal{T}_R = \mathcal{M}_1 \cup \cdots \cup \mathcal{M}_n \cup \mathcal{B}$, where $\mathcal{M}_1, \ldots, \mathcal{M}_n$ are role models and $\mathcal{B}$ is a set of role binding axioms
- $\mathcal{A}_R$ is a set of role assertion axioms

The sets of role type names (and role property names respectively) occurring in $\mathcal{M}_1, \ldots, \mathcal{M}_n$ are pairwise disjoint. Furthermore, they are disjoint with the set of class names (and property names respectively) occurring in $\mathcal{T} \cup \mathcal{A}$.

Translation to $\mathcal{ALC}$

The above syntax extensions allow to model ontologies with roles and hence, use the benefits described in the previous sections. In order to be compatible with existing standards and tools, we propose a translation of $\mathcal{ALC}^{\mathcal{R}}$ ontologies to $\mathcal{ALC}$ ontologies:

1. Make all role types (and their definition for defined role types) of the role models $\mathcal{M}_1 \cup \cdots \cup \mathcal{M}_n$ available as classes in the ontology.
2. Make all role properties used in the role models $\mathcal{M}_1 \cup \cdots \cup \mathcal{M}_n$ available as properties in the ontology.
3. For each role type R used in the ontology:
   a) Let $\{C_1, \ldots, C_n\}$ be the set of classes to which R is bound (i.e. $R \rhd C_i$). Then add axiom $R \sqsubseteq C_1 \sqcup \cdots \sqcup C_n \sqcup \bot$ to the ontology.
   b) For each role assertion $R(a)$, make the same assertion available in the resulting ontology, now referring to the class-representative for the role type R.
4. Remove $\mathcal{T}_R$ and $\mathcal{A}_R$ from the ontology.

The resulting ontology uses only $\mathcal{ALC}$ constructs since:

- All role type terms have been replaced by class terms.
- The role types and role properties of all role models have been transformed into classes and properties.
- All role type bindings have been removed.
- All role type assertions have been transformed into class assertions.

Why we need this syntactic sugar

One may argue that, since $\mathcal{ALC}$ and $\mathcal{ALC}^{\mathcal{R}}$ are equivalent from a semantic point of view, there is no need for a separate syntax for role modeling. However, we believe that ontology engineers should be able to describe the conceptual difference between role types and natural types with language primitives for role modeling. Doing this, they benefit from the advantages described in the preceding parts of this chapter.

Example using Manchester OWL Syntax

Example: Wine Ontology

In Section 3.2 we described the composition of several role models into an ontology by the example of a wine ontology using a graphical, rather informal notation. While this was useful to explain the idea and benefits of role model composition, we come back to the example now using a textual and formalized presentation.

Modeling the role-based ontology from Figure 3.7 in concrete syntax based on Manchester OWL would like this:

```
import http://ontology-rolemodels.org/product.rm
import http://ontology-rolemodels.org/meal.rm

Class: Wine
      Plays: Product
      Plays: Drink

Class: Winery
      Plays: Producer

Class: Food
      Plays: Meal
      Plays: Product
```

We have omitted the *Taste* role model as well as the *Taste* class since they are not interesting for illustrating the translation. To demonstrate the impact of binding one role type to multiple classes, we assume the *Product* role type also to be bound to *Food*. That is, foods can also be considered products in some contexts.

The role models that are imported could be defined as follows:

```
Rolemodel: http://ontology-rolemodels.org/product.rm
      Role: Product
      Role: Producer
            EquivalentTo: produces some Product
      Role: Seller
            EquivalentTo: sells some Product and sellsTo some Consumer
      Role: Consumer
            EquivalentTo: consumes some Product

      Role Property: produces
            Domain: Producer
            Range: Product
      Role Property: sellsTo
            Domain: Seller
            Range: Consumer
      Role Property: consumes
            Domain: Consumer
            Range: Product
```

```
Rolemodel: http://ontology-rolemodels.org/meal.rm
      Role: Drink
            EquivalentTo: accompanies some Meal
      Role: Meal

      Role Property: accompanies
            Domain: Drink
            Range: Meal
```

Our translation as defined above yields the following class-based ontology (omitting the role properties that are simply copied in as usual class properties):

```
Class: Wine
Class: Winery
Class: Food

Class: Product
      SubClassOf: Wine or Food
Class: Producer
```

```
      EquivalentTo: produces some Product
      SubClassOf: Winery
Class: Seller
      EquivalentTo: sells some Product and sellsTo some Consumer
      SubClassOf: owl:Nothing
Class: Consumer
      EquivalentTo: consumes some Product
      SubClassOf: owl:Nothing

Class: Drink
      EquivalentTo: accompanies some Meal
      SubClassOf: Wine
Class: Meal
      SubClassOf: Food
```

Role types are defined as subtypes of the disjunction of the classes they are bound to. Note that, open role types like *Consumer* appear as subtypes of *owl:Nothing*, that is, there cannot be any individuals belonging to those role types. This reflects the idea that each individual has an identity and consequently, belongs to at least one natural type, i.e. class.

Summary

This concludes the definition and illustration of the semantics of role modeling in ontologies. We have extended the syntax of the $\mathcal{ALC}$ DL and defined a translational semantics, transforming extended syntax into standard ontologies. An example helped the reader to understand this translation. We believe that the proposed syntax extensions and semantics are only one possible implementation of the conceptual idea of ontology roles. In Chapter 5 we will discuss others.

3.4 Comparison with Object-Oriented Software Engineering

OO modeling vs. ontologies

The preceding sections of this chapter were about applying the paradigm of role modeling to ontologies. While there is no similar work known to the authors, role modeling and its applications in object-orientation are well studied. An interesting question is how both worlds compare to each other with respect to roles. This section analyzes commonalities of roles in object-orientation and ontologies, and examines which differences we encountered and consider to be important for transferring roles into ontologies.

Natural types vs. role types

As a first commonality, both worlds abstract sets of elements on the instance level (objects and individuals respectively) with types on the conceptual level. Although our definition of ontological roles differ from the classical definition (see Section 2.4), the fundamental distinction of natural types and role types seems meaningful in both worlds. From a modeling point of view, this differentiation seems to be the most important aspect of roles.

Collaborations as reuse abstraction

Furthermore, ontology role modeling and roles in software engineering share most of the properties inherent to the role-based approach. In particular, roles allow to focus on relations between individuals or objects and to abstract them with role models that describe collaborations. These role models provide a self-contained reuse abstraction that allows to use parts of an ontology or pieces of source code again in cases where classes are not the appropriate abstraction. For instance, collaborations serve to describe design patterns in object-oriented designs [GHJV95, Rie96]. Chapter 3.2 showed how collaborations provide a reuse abstraction for ontologies.

Integration into mainstream languages

The last commonality to mention is a cheerless one. Although the ideas of role modeling are widely accepted and its benefits are undoubted, neither a mainstream language for object-oriented programming, nor an ontology

language supports roles as first-class concept. To deal with that situation, a number of work-arounds have been proposed in the software community [BRSW97, Ste01, SB02].

Subject of modeling

When transferring roles into ontologies, a number of differences of object-orientation and ontologies have to be considered. A substantial discrepancy exists regarding what is modeled. While object-orientation describes a dynamic world that changes during the runtime of an application, ontologies provide a static view of the world. This leads to different definitions of a role. On the one hand, a role describes the *behavior* of an object and on the other hand, the *properties* of an individual.

Open vs. closed world

While object-oriented modeling employs closed-world semantics (i.e. what is not explicitly modeled is assumed not to exist), ontologies assume an open world (i.e. non-existence does not imply falsity). Of course, this difference also affects roles. Using the semantics defined in Section 3.3, not binding a role type to a certain class does not imply that individuals belonging to the class cannot play that role (except for the case where the class is declared to be disjoint from the classes the role is bound to). We believe this to be a quite natural consequence of introducing roles into ontologies, whereas enforcing closed-world semantics for roles would contradict fundamental ideas of ontological modeling.

Role constraints

In Section 2.4, we described role constrains as a means to enforce certain conditions on which combinations of roles individuals are allowed to play. In ontologies, constraints of this kind can be expressed using the usual ontological axioms. Let (R, S) be a pair of role types. The role constraints from Section 2.4 roughly correspond to the following axioms:

- *Role-implied:* $R \sqsubseteq S$
- *Role-equivalent:* $R \equiv S$
- *Role-prohibited:* $R \sqcap S \sqsubseteq \bot$
- *Role-don't care:* (nothing)

However, these axioms do not reflect the fact that role constraints in object-orientation are scoped by an object collaboration task. How to express this notion of scoping in role-based ontologies is still an open question.

Reusability

An interesting question is whether ontological role models are more apt for reuse than their object-oriented counterparts. The contrasting nature of what is described influences how precise role types have to be defined. Using roles in object-orientation as partial implementation of a class (as for instance in [SB02]) requires a definite specification of the behavior the role represents in form of a concrete implementation. Ontologies only require to fix static modeling aspects and constraints imposed on role types. This may imply that role models in ontologies are less restrictive than implementations of role models in software. Consequently, their reusability would be larger while providing all modeling that is possible and necessary in ontologies. Clarifying this issue in more depth should be part of future research.

Summary

Although a couple of issues had to be considered, the differences of roles in object-orientation and ontologies do not hinder the transfer of many of its ideas. Most of the inherent properties of the role modeling approach can be adapted and lead to the conceptually very similar definitions in Section 3.1.

4 Implementation with Reuseware

Prototype of translation

This chapter presents a prototypical implementation of role-based ontology engineering using the Reuseware composition framework [HAJZ07]. Our implementation allows for defining role models and role-based ontologies, as well as for their translation into standard ontologies. We use a syntax based on the Manchester OWL syntax [HDG+06] and extend it with constructs needed for role modeling as described in Chapter 3.

Organization

At first, we give an overview of our implementation and explain how it is integrated in the Reuseware framework (Section 4.1). Section 4.2 presents the abstract syntax and concrete syntax of the role modeling enabled Manchester OWL. Finally, Section 4.3 shows how the translation to standard OWL is realized as invasive composition. The complete source code of our implementation can be found in Appendix B.

4.1 Overview

Core language and reuse language

Reuseware realizes the ideas of invasive software composition [Ass03] for arbitrary languages. For invasive composition, we require knowledge of the metamodel of the language we are working with. In Reuseware, two kinds of languages are distinguished: A *core language*, lacking sufficient support for modularity, and a *reuse language*, extending the core language with new syntax to overcome this lack.

Owlm and reuseowlm

As we have shown in Section 2.2, the built-in mechanisms of OWL for reuse and modularity are inflexible and do not allow for partial reuse. OWL, more precisely the Manchester OWL syntax, is hence our core language (called *owlm* in the sequel). In Chapter 3, we introduced roles into ontologies which lead to role models as a notion of ontological components. However, OWL does not support roles and has to be extended. This extension yields our reuse language (called *reuseowlm* in the sequel).

Overview

We define owlm and reuseowlm using EBNF-like grammars that allow to generate Eclipse plugins supporting the actual composition. A set of role models in reuseowlm are the components. Role-based ontologies describe how to combine role types and classes with role binding axioms and are hence the composition program. The details of the composition, i.e. the translation to standard OWL, is described by two composers and finally yields a class-based ontology in owlm. Figure 4.1 shows an overview of the implementation.

4.2 Core Language and Reuse Language

Abstract syntax of owlm

This section presents the specifications of owlm and reuseowlm. We concentrate on important parts of the grammars that should help the reader to understand our implementation without bothering about all details (see Appendix B for the complete grammars). At first, we define the abstract syntax of owlm:

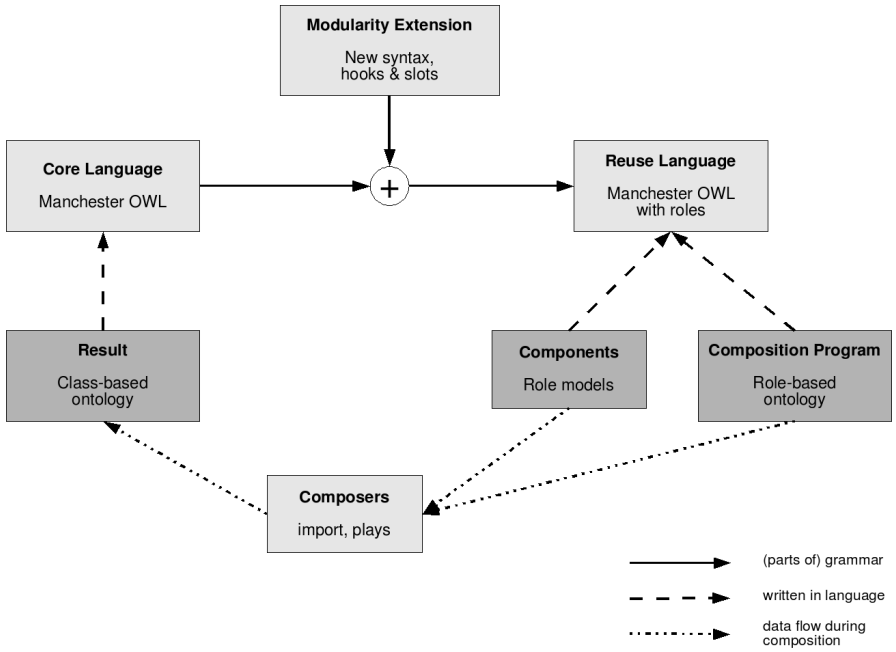

Figure 4.1: An extension of the core language yields the reuse language. The composition program describes how components are invasively combined using composers.

```
Ontology           = statements:OntologyStatement*;
OntologyStatement  = ClassDescription | PropertyDescription | Assertion;
ClassDescription   = classID:NamedType, description:Description*;
```

Concrete syntax of owlm

An ontology is a list of statements that can describe class descriptions, properties, and assertions. The abstract syntax is accompanied by a concrete syntax:

```
CONCRETESYNTAX owlm FOR owlm
Ontology           ::= "Ontology:" statements*;
ClassDescription   ::= "Class:" classID description*;
```

Abstract syntax of reuseowlm

The above grammars define the core language owlm and allow for generating a parser accepting class-based ontologies. To support role modeling, we define the reuse language reuseowlm. Its abstract syntax extends nonterminal symbols from owlm, for instance adding a new type of statement for importing role models:

```
owlm.OntologyStatement = ImportStmt;
ImportStmt             = filename:componentmodel.Location;
ImportStmt             ==> minimalcl.Composer;
```

The first line extends the derivation options for *owlm.OntologyStatement* with *ImportStmt*, i.e. an ontology now also may contain import statements. In the third line, the import statement is declared as a subclass of *Composer*. This allows to define a composer translating the import statements. Similarly, we also add a construct for combining classes with role types (*Plays*) to the grammar (omitted). Furthermore, the language extension adds support for role models:

```
RoleModel     = modelID:owlm.NamedType, stmts:RoleStatement*;
RoleStatement = RoleDescription | RolePropDescription;
```

A role model has an identifier (its name) and consists of a list of statements which can describe role types (*RoleDescription*) and role properties (*RolePropDescription*).

The translation from Section 3.3 requires to declare a class representing a role type as subclass of the disjunction of all classes it is bound to. For instance, if the role type *Product* is bound to *Pizza* and *Wine*, the class *Product* will be declared as a subclass of *Pizza OR Wine*. The disjunction of subclasses is built during the composition analyzing the *Plays* constructs used in the role-based ontology. Each occurrence of *Plays* will add a new subclass disjunct. Consequently, we introduce a hook, i.e. a variation point that may be extended multiple times:

```
AtomicExpressionHook;
owlm.AtomicExpression = AtomicExpressionHook;
AtomicExpressionHook  ==> componentmodel.Hook;
```

Hooks are declared by subclassing *componentmodel.Hook* and giving the type of fragments that may be bound to it (*owlm.AtomicExpression*).

The following is the concrete syntax of reuseowlm:

Concrete syntax of reuseowlm

```
CONCRETESYNTAX rowlm FOR reuseowlm EXTENDS owlm

owlm.Ontology        ::= "Ontology:" statements*;

ImportStmt           ::= "import" filename;

RoleModel            ::= "Rolemodel:" modelID stmts*;
RoleDescription      ::= "Role:" roleID descriptions*;
RolePropDescription  ::= "Role Property:" propID "Domain:" domain "Range:" range;

AtomicExpressionHook ::= "<<" name ">>";
```

The above grammars define the core language owlm and the reuse language reuseowlm. These grammars give us the metamodels of our languages and allow to continue with the actual composition.

4.3 Composers and Composition Program

Composition

In this section, the translation from role-based ontologies into class-based ontologies is explained as invasive composition of role models. During the composition, all constructs that have been introduced in reuseowlm must be translated into owlm. We have to consider the following:

- *Import* statements
- *Plays* constructs
- Role models consisting of role types and role properties
- Role assertions

Two composers

The main idea is to use a role-based ontology as a composition program referring to composers for *import* and *Plays*. The import composer makes all role types and role properties from the imported role model available as classes and properties (i.e. step 1 and 2 of the translation in Section 3.3). The plays composer builds the subclass-superclass relationships between classes representing role types and role-playing classes (i.e. step 3 from the translation). Role assertions have exactly the same syntax as class assertions and thus, do not require particular consideration but are simply left as they are.

Import composer

The following is an example import statement:

```
import /Taste.rowlm
```

Since we declared the import statement as a subclass of *Composer*, we can define a composer for it:

```
define composer reuseowlm.ImportStmt(filename) {

    fragmentlist reuseowlm.RoleModel rm = →filename;
    fragmentlist owlm.Ontology roleOntology = 'Ontology: Class: Top'.rowlm;
```

The filename (e.g. */Taste.rowlm*) is the only parameter and is used to read in the file's content as a role model (third line). Furthermore, we create an ontology that serves as a container for the newly added ontology statements.

Then, we iterate through the list of role model statements. A statement may be either of the type *RoleDescription* or of the type *RolePropDescription*:

```
    foreach (stmt : rm.stmts) {
        if (stmt instanceof reuseowlm.RoleDescription) {
            fragmentlist owlm.ClassDescription newCls = 'Class: ' + stmt.roleID
            + ''.rowlm;
            extend newCls.description with stmt.descriptions;

            fragmentlist owlm.SubClassOf extSCO = 'SubClassOf: ( Bottom OR <<'
            + stmt.roleID + 'SubClassHook>> )'.rowlm;
            extend newCls.description[first] with extSCO;

            extend roleOntology.statements[first] with newCls;
        }
```

For each role description (i.e. role type defined in the role model), a new class is created having the definition of the role type it represents. Then, we introduce a hook for adding new subclasses and add *Bottom* as first disjunct. If a role type is not bound to any class (i.e. left open), the hook will not be bound and the class is declared as a subclass of *Bottom*. Other subclasses are added to the disjunction in the plays composer that identifies the hook using a naming convention ('<<' + *roleID* + '*SubClassHook*>>'). Finally, the new class is added to the statements of the container ontology.

```
        if (stmt instanceof reuseowlm.RolePropDescription) {
            fragmentlist owlm.PropertyDescription newProp = 'Property: ' + stmt.propID
            + ' Domain:' + stmt.domain + ' Range:' + stmt.range + ''.owlm;
            extend roleOntology.statements[first] with newProp;
        }
    }
```

Role properties are simply converted into properties by copying their name, domain, and range, and adding them to the statements of the container ontology. Finally, the statements are returned by the import composer, i.e. instead of the import statement, the list of classes and properties is inserted into our resulting ontology:

```
    return roleOntology.statements;
}
```

The following is an example for the *Plays* construct triggering the plays composer: Plays composer

```
Class: Pizza
   Plays: Tasting
```

The plays composer has the *roleID* of the played role type as a parameter:

```
define composer reuseowlm.Plays(roleID) {

    fragmentlist owlm.Ontology ontology = CTX_ROOT;
    fragmentlist owlm.ClassDescription clsDesc = CTX_CONTAINER;
```

The composer must refer to the complete ontology in order to find the subclass hook that has been introduced in the import composer. It obtains a reference of the complete ontology from *CTX_ROOT* and a reference of the surrounding construct (e.g. the class *Pizza* in the above example) from *CTX_CONTAINER*.

```
    fragmentlist owlm.ClassExpression clsExpr = clsDesc.classID+''.owlm;
    fragmentlist componentmodel.VariationPointName hookName = roleID +
     'SubClassHook'.bc;
    extend →hookName on ontology with clsExpr;
}
```

Using our naming convention for hooks, we construct the correct hook name (*TastingSubClassHook*) and extend it with the surrounding class expression (*Pizza*). Applying the import composer for all import statements and the plays composer for all *Plays* statements transforms all reuseowlm constructs into owlm. Thus, the composition finally yields a class-based ontology.

Summary

In this chapter, we have presented an implementation of the translation from Section 3.3 using the Reuseware composition framework. It is a first prototype allowing to model ontologies with roles and afterwards translate it into a standard ontology language in order to use existing tools, e.g. for reasoning.

5 Outlook

Not the end but the beginning

The idea of introducing roles into ontologies has not been considered in any other work known to the author, and of course, not all aspects and implications of ontological roles can be part of this work. Quite the contrary, we believe that the proposed ideas are only the beginning and further investigations on a wide range of issues should be done. This chapter presents a couple of ideas that have not been examined in-depth but nevertheless seem to be worth mentioning to support future research.

Combining different ontologies

An ontology aims to provide a common model of domain concepts and their relationships. It attempts to offer a unified view in order to build interoperable applications. However, ontologies are modeled by many people and everyone possibly has a different view of the world. The diversity of people's views results in a diversity of ontologies describing different but often overlapping parts of the world. To achieve the actual goal of harmonizing the conceptual model of a domain, different ontologies have to be related and combined. Although promising approaches have been proposed (see Chapter 2.3), there still are a number of unsolved problems, in particular, mismatches that cannot be resolved automatically. We believe that role-based ontologies are in a certain sense easier to relate and combine than purely class-based ones since they offer an additional abstraction helpful when searching commonalities between different ontologies. This idea, composition of role-based ontologies, is presented in Section 5.1.

Degrees of coverage

Today's ontologies are modeled with classes. An important question to consider is how to introduce roles into existing ontologies in order to benefit from more natural modeling, separation of concerns of classes, and composition of role-based ontologies. In Section 5.2, we describe a step-by-step introduction of roles into class-based ontologies leading to different levels of role coverage. Role coverage denotes the degree of role usage in an ontology which may influence the likelihood of an ontology to be apt for role-based ontology composition.

Finally, Section 5.3 presents further ideas for future investigations and concludes this chapter.

5.1 Composition of Role-Based Ontologies

Composing role types instead of classes

This section tries to answer the question how to semi-automatically compose ontologies such that the result does not break the semantics. Promising approaches towards ontology aligning, mapping, and merging have been proposed and implemented (see Section 2.3). However, these approaches concentrate on only one abstraction of sets of individuals—classes. This leads to difficulties when dealing with classes that intermingle different concerns. We argue that the existing solutions should be complemented by role-based techniques that consider the collaborations of classes during the composi-

tion process.[1] We believe that composing role types is in some situations easier than composing classes since role types cover only one concern of an individual, whereas classes usually mix several. Furthermore, role models clarify the context to which class restrictions and relationships belong. This can hardly be determined by class-based approaches, since classes do not distinguish relationships from different contexts.

Example: Pizza and pasta

Before going into details of role-based ontology composition, an example is shown to give the intuition of the idea. We aim at composing two role-based ontologies covering the pizza and pasta domains (Figure 5.1). They are modeled such that all collaborations of individuals are abstracted with role models and no properties exists between classes.[2] The main idea is to initially relate the role models from both ontologies and afterwards compose the classes based on the role type relations bridging the ontologies. In Figure 5.1, we illustrate different role model relations. One role model (*Origin*) is shared by both ontologies which is the ideal case for the purpose of composition. The *Taste* and *Flavor* role models seem to describe the same issue but with different means of expression. Hence, they should be treated as being equivalent. The other role models (*Meal* and *Product*) do not relate to each other.

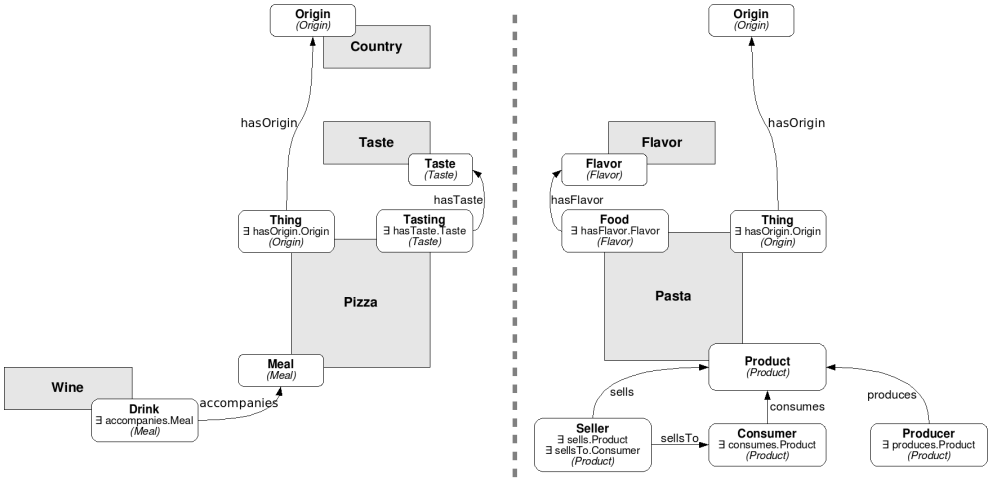

Figure 5.1: Pizza and pasta ontologies modeled with roles.

Aligning the role models is visualized in Figure 5.2, where the dashed line means equivalence of role types. This information can now be used to compose the classes of the ontologies. The open role type *Country* from the pasta ontology can obviously be bound to the class *Country* from the pizza ontology, since it only contains this identical role type. *Flavor* and *Taste* also contain only one role type each. Since these have been found to be equivalent during the role model alignment, their classes are also equivalent. In contrast, the classes *Pizza* and *Pasta* only partially overlap. Both pizza and pasta can have an origin (they share the *Thing* role type from the *Origin* role model), and in addition, the role types *Tasting* and *Food* are equivalent

[1] In this chapter, we refer to combining and relating ontologies as *ontology composition*, in contrast to *role model composition* as explained in Section 3.2.

[2] Class relationships and properties inherently belonging to a natural type are ignored here since they complicate role-based composition.

according to the role model alignment. However, pizzas are considered to be meals in some contexts, while pasta can be seen as a product. In other words, the residual role types *Meal* and *Product* do not match. To model this situation, the composition system should propose a common superclass for *Pizza* and *Pasta*. The result of the composition process is shown in Figure 5.3, where equivalent classes are merged into one and a new class is introduced as superclass of *Pizza* and *Pasta*. Giving it a meaningful name cannot be done automatically.

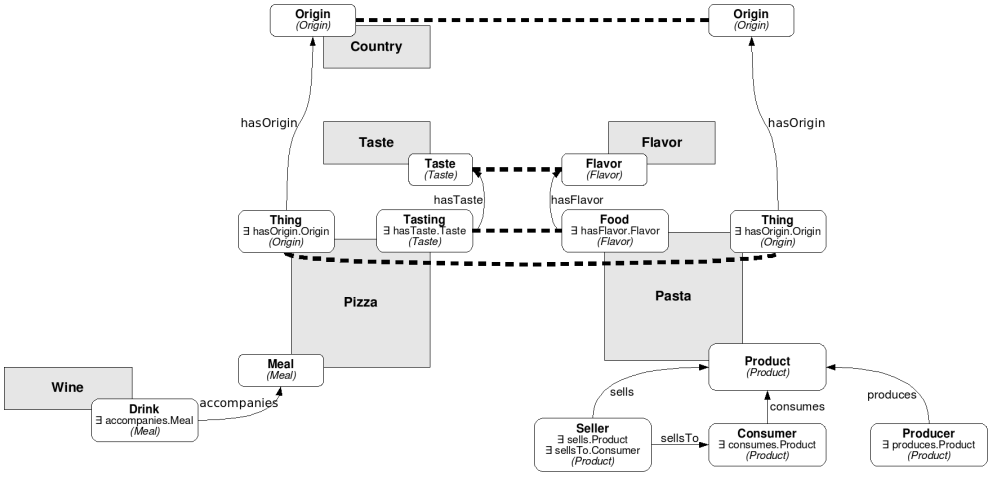

Figure 5.2: Alignment of role models of pizza and pasta ontology.

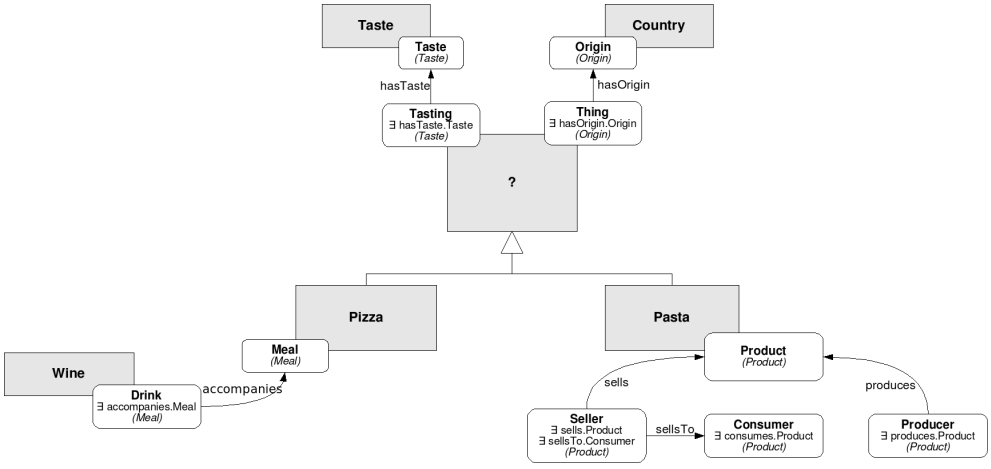

Figure 5.3: Composition of pizza and pasta ontologies using roles.

Composition technique

After illustrating it with an example, composition of role-based ontologies is described in the general case in the following. The composition process consists of two major steps.

Role model alignment

1. At first, the role models from the input ontologies have to be aligned. That is, relations between role types and role properties are analyzed and if possible, role models (and hence their role types) are declared to be equivalent. This step consists of pairwise comparison of all role

models, i.e. $m \times n$ comparisons if the ontologies contain m (and n respectively) role models. During this step, we consider two cases:

- Two ontologies may use the same role models, for instance taken from a common role model catalog. Consequently, they are equal and no further alignment is needed.
- Role models may want to express the same or a very similar collaboration but do not use exactly the same expressions. In other words, we encounter explication mismatches of role models. In this case, existing techniques to overcome those mismatches can be applied. For instance, techniques to align classes based on thesauri can be plugged-in to align role types.

2. The role model alignment described above yields two ontologies described by role models whereof some are equivalent. The second step of the composition process is now relating and combining classes. Four basic cases may occur: Class composition

   - All role types of two class are equivalent. Thus, the classes are equivalent as well.
   - Two classes have equivalent role types but also contain different ones. That is, their individuals share a number of common properties but differ in others. In the example, this is the case for the classes *Pizza* and *Pasta*. From a modeling perspective this situation quite clearly demands for a common superclass. A composition system should propose one to the ontology engineer who has to decide whether a superclass makes sense and if so, how to name it. Unfortunately, it seems that this step cannot be fully automated.
   - Classes do not have any equivalent role types, and hence, cannot be aligned at all.
   - Two role types are equivalent and one is open, i.e. not bound to any class. An open role types indicates an unspecified part of an ontology. If one ontology provides a class containing a role type that is left open in the other ontology, individuals of that class can play the required role. Thus, the open role type is bound to the class containing the bound role type. For instance, when composing the pizza and pasta ontologies, this case occurs for the *Country* role type.

   Relating and combining classes based on role model equivalences can lead to more complicated cases. For example, role type equivalence may propose to split a class into several classes. However, we limit our discussion to the basic cases described above.

Class alignment vs. role model alignment

So why not apply techniques to resolve explication mismatches directly to classes? The problem is that classes intermingle concerns. Consequently, they often overlap partially without being equivalent and it is hard to separate matching parts from those that differ in a semantically well-defined way. In contrast, role modeling splits classes into semantically independent parts and thereby allows to apply matching techniques on a more fine-grained level. For example, the classes *Pizza* and *Pasta* semantically overlap, but are not equivalent. A class-only approach would probably not be able to split both class definitions such that the roles of having an origin and a

taste are recognized as shared while the rest of the classes differ in some way.

Classification as ontology alignment approach

In Section 2.3, we mentioned a classification of ontology alignment approaches [dBEF+06]. Our proposal to compare role models can be classified as a schema-based approach (as opposed to instance-based) on a structural level (as opposed to element level).

For composing ontologies based on role models, we assumed the source ontologies to have a particular property—global role coverage. Intuitively, it means that all collaborations of individuals that an ontology formalizes are described by role types and none by classes (see the following section for more details). To understand why we need global coverage to decide whether classes are (partly) equivalent by only regarding their role types, imagine a class containing role types but also properties that are not part of any role type. If all its role types imply equivalence to another class, the non-role properties may still contradict this equivalence. Thus, to compose ontologies based on their role models, the concerned classes have to be covered by roles. In order to compose a complete ontology, it must be globally covered.

Summary

To sum up, this section proposed a role-based ontology composition technique. The main idea is that composing role types is in some situations easier than composing classes since role types separate different concerns and provide another abstraction than classes. The composition process consists of two steps. At first, the role models from both ontologies are aligned. Afterwards, their classes are composed based on the alignment of contained role types.

5.2 Levels of Role Coverage

How to add roles to a class-based ontology?

Due to the emerging semantic web and other applications of ontologies, e.g. in biomedical informatics, there exists a large number of ontologies. They are finite sets of axioms using the modeling concepts individual, class, and property. None of them is built from roles, role types, and role models. However, as shown in Chapter 3, role-based ontologies yields numerous benefits. The natural question to ask is how to introduce role modeling into existing ontologies.

Staged introduction

We propose to enhance an existing ontology with roles using a staged approach that includes four levels. Each level increases the degree of role modeling used in the ontology, and at the same time the advantages that can be taken from it. Level 0 are *class-based ontologies*, i.e. not containing roles at all. Almost all existing ontologies are on that level. Introducing a couple of role models leads to *partially covered ontologies* (level 1). It allows to reuse role models and improves modeling since natural types and role types can be distinguished. Level 2, *locally covered role-based ontologies*, demands a property called *coverage* to be fulfilled in parts of the ontology. In addition to the benefits from level 1, local composition becomes possible. That is, parts of the ontology can be composed with others as described in Section 5.1. When the coverage property holds for the complete ontology, level 3 is reached: *Globally covered role-based ontologies*. In that case, the complete ontology can be composed based on roles. Figure 5.4 provides an overview of the levels and their benefits.

The notion of coverage

The coverage of a role-based ontology describes to what extent the properties and collaborations of individuals that are formalized are described by roles. A role-based ontology is *partially covered* if parts of its collaborations

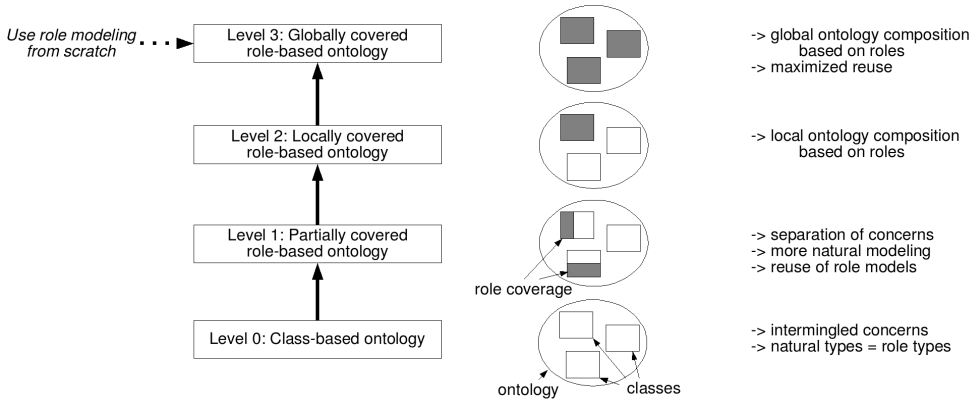

Figure 5.4: Levels of role modeling of ontologies and their benefits.

are described using role types. In that case, an individual may belong to all role types of a class but nevertheless not belong to the class itself, since some restrictions of the class may not be specified by any role type. If all parts of a class description are included in one of its role types, this class is *covered*. Thus, an individual belonging to all role types of a class, necessarily belongs to the class, too. An ontology including some covered classes is called *locally covered*. If all classes of the ontology are covered, it is *globally covered*. Figure 5.4 schematically presents the degrees of coverage.

In the following, a step-by-step introduction of roles into an existing ontology is described. We explain the properties of each level, what has to be done to reach it, and what benefits arise.

Level 0: Class-Based Ontology

Existing ontologies

Almost all existing ontologies belong to level 0. These are class-based ontologies without any role models. Although well-established, they possess a number of substantial problems: No distinction of natural types and role types leading to poor modeling, intermingling of concerns hampering reuse, and hard-to-resolve ontology mismatches complicating ontology composition.

Example

For instance, the pizza ontology from Figure 5.5 is a class-based ontology. It intermingles two independent concerns of food in the *Food* class. One of them, the relation between *Food* and *Spiciness* describes the taste of food and seems to be interesting to reuse. However, it is not specified which parts of *Food* concern the taste.

Level 1: Partially Covered Role-Based Ontology

Introducing role models

A class-based ontology may be enhanced by role models in two ways. Either some mismodeled classes are partially replaced by role types, or new aspects of the domain under consideration are described using roles. Both leads to a partially covered role-based ontology.

In our example, we detach the concern of food to have a taste into a role model as shown in Figure 5.6. Parts of the definition of the *Food* class are transferred into the role type *Tasting* of the *Taste* role model.

Benefits

Enhancing the pizza ontology with the *Taste* role model brings two main

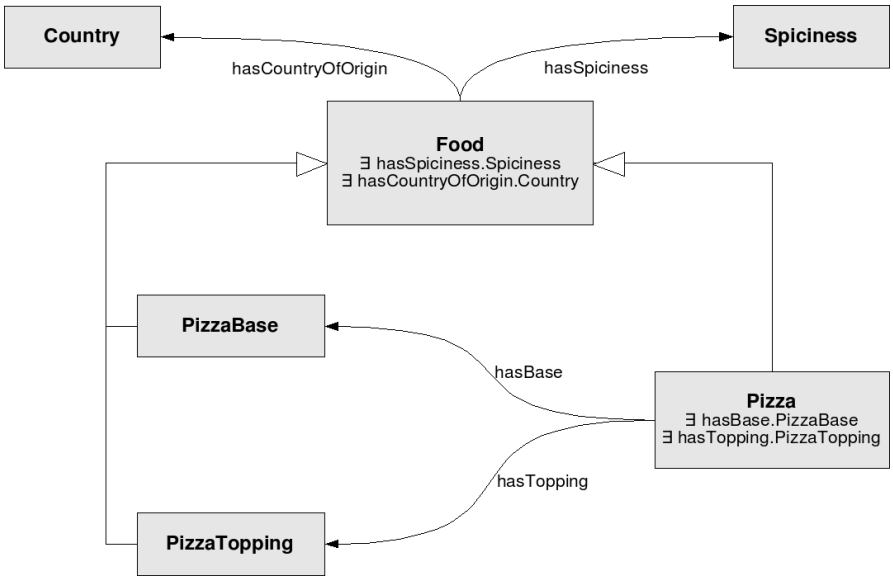

Figure 5.5: A level 0 ontology without any role models (same figure as in Chapter 2).

advantages. First, the modeling becomes clearer since it is explicitly defined which parts of *Food* belong to the taste concern. The role model provides a self-contained fragment of the modeled whole that can be added and removed without affecting the rest of the model. Second, the role model is reusable and may be used again in another ontology describing individuals possessing a taste.

Level 2: Locally Covered Role-Based Ontology

Covering particular classes

In a level 1 ontology, no class is covered, i.e. there are still collaborations described in the class and not by a role type. If, however, a proper subset of its classes is completely covered, we have a locally covered role-based ontology.

Example

In Figure 5.6, parts of the *Food* class and its relationships have been transferred to the *Taste* role type, whereas the concern of having an origin is still part of the class. Recalling the definition of natural types and role types, that implies having an origin to be an inherent property of food. However, we believe that this should be modeled as role type which yields the ontology in Figure 5.7. Now, the *Food* class is completely covered by role types, and thus, the ontology is locally covered.

Benefits

The main advantage of complete coverage is that completely covered classes can be composed by only regarding their role types and role relationships. The reason is that no other restrictions may contradict what has been found when looking at the role types. For instance, another ontology requiring some of the role types of a completely covered class and indicating this lack by open role types, can be composed immediately. Furthermore, relating and combining classes is facilitated since different concerns can be treated independently (see Chapter 5.1).

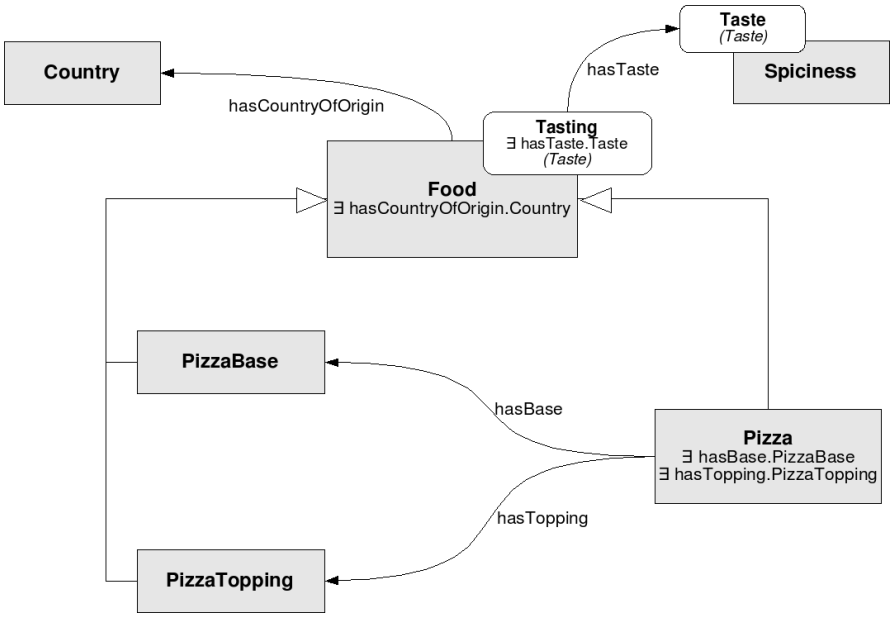

Figure 5.6: The pizza ontology as partially covered role-based ontology. The *Food* class is only partially covered since the concern of having an origin is not described as role type.

Level 3: Globally Covered Role-Based Ontology

Covering all classes

By expanding the coverage property to all classes and properties of an ontology, we arrive at globally covered role-based ontologies. That means, all described collaborations of an ontology are represented by role models.

Do we still need classes?

An interesting question is whether covered classes should still be allowed to contain restrictions or whether the complete class description can be deduced by merging its role types. Referring to the definition of natural types and role types, we believe that inherent properties of individuals should still be modeled with classes. Another answer could be identity roles [Bac80], a somewhat denaturalized usage of role types for representing properties actually belonging to the identity of individuals (and thus, normally to be represented as natural types). A better answer to this question should be subject of further investigations.

Example

Figure 5.8 presents our example as a globally covered role-based ontology. The *Layers* role model may be subject of discussion since having a base and a topping can also be seen as an inherent property of pizzas. Though, it is used here for the sake of our example. Another (maybe better) example for for a globally covered role-based ontology is the wine ontology from Figure 3.7.

Benefits

Global coverage provides all benefits of local coverage for the complete ontology. That is, the entire ontology can be composed based on roles as described in Section 5.1. Furthermore, the degree of reuse is maximized, since the complete knowledge description is based on potentially reusable role models.

Summary

This completes the step-wise introduction of role modeling into an existing ontology. Starting from an ontology modeled exclusively based on classes, adding reusable role models leads to better modeling. Transferring all de-

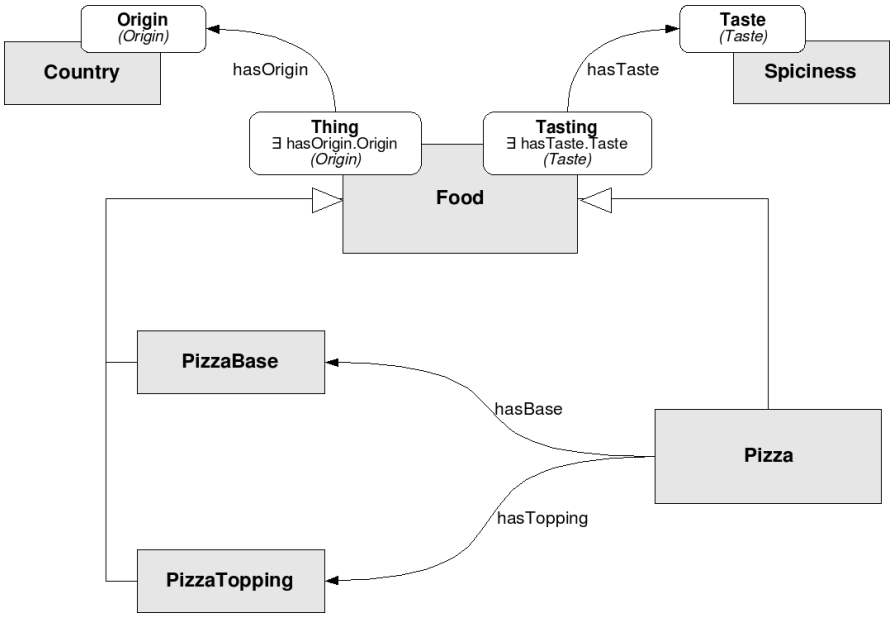

Figure 5.7: The upper part of the ontology (*Country*, *Food*, and *Spiciness*) are completely covered, while the lower part is not—a locally covered role-based ontology.

scriptions of a class' collaborations into role types yields role coverage, and thus enables local composition based on roles. By expanding this property onto the complete ontology, global composition becomes possible and reuse is maximized.

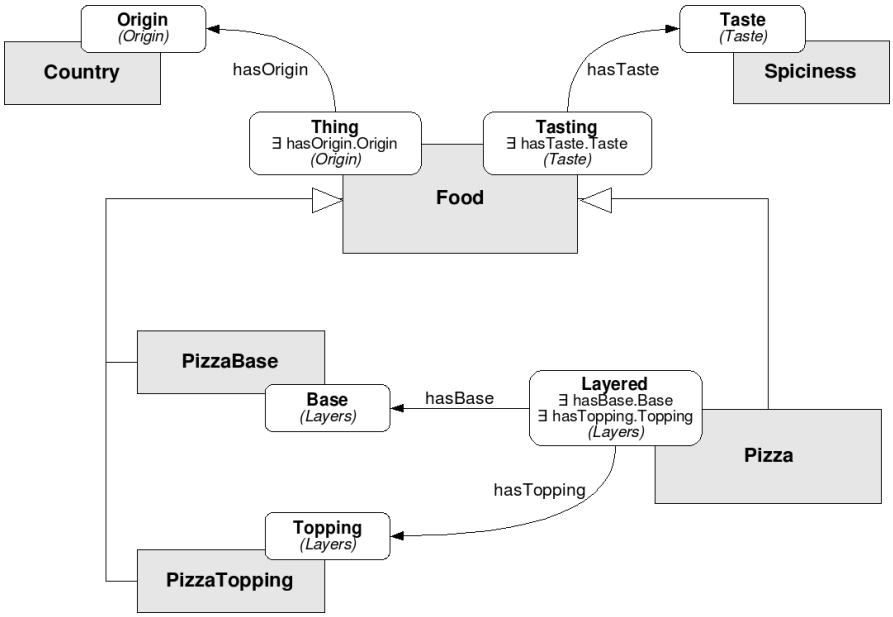

Figure 5.8: All classes are completely covered—a globally covered role-based ontology.

5.3 Further Ideas

Tool integration

Ontological role modeling should be integrated into common ontology tools like the Protégé ontology editor [KFNM04]. Since role modeling can be translated into standard OWL (see Section 3.3), an adaptation of ontology reasoners is not required. However, ontology engineers should be supported in using roles on a syntactical level.

Role modeling in Protégé

As a very first teaser for further developments, we have implemented some mockups demonstrating how Protégé support for role modeling could look like. Figure 5.9 is a screenshot of a tab for the creation of role models. The upper-right panel (*Role Model Browser*) shows the list of role models currently used in the ontology and allows to create new ones or delete them. For a selected role model, its role type can be edited in the lower-right panel (*Roles*). The right panel (*Role Editor*) serves for editing role types by adding and removing role expressions and disjoint role types.

More extensions are necessary to fully implement the ideas of this work. The class tab should be enhanced such that binding roles to classes is possible. Furthermore, the individuals tab should provide support for role assertions. An interesting question is how to store role models, how to relate them with the ontology and when to invoke the translation into the standard ontology language. One proposal would be on-the-fly translation and marking translated elements with annotations that are interpreted by the Protégé extension. Of course, completely different realizations of ontological role modeling in Protégé and other tools is possible as well. Our example merely intends to be a starting point for discussion.

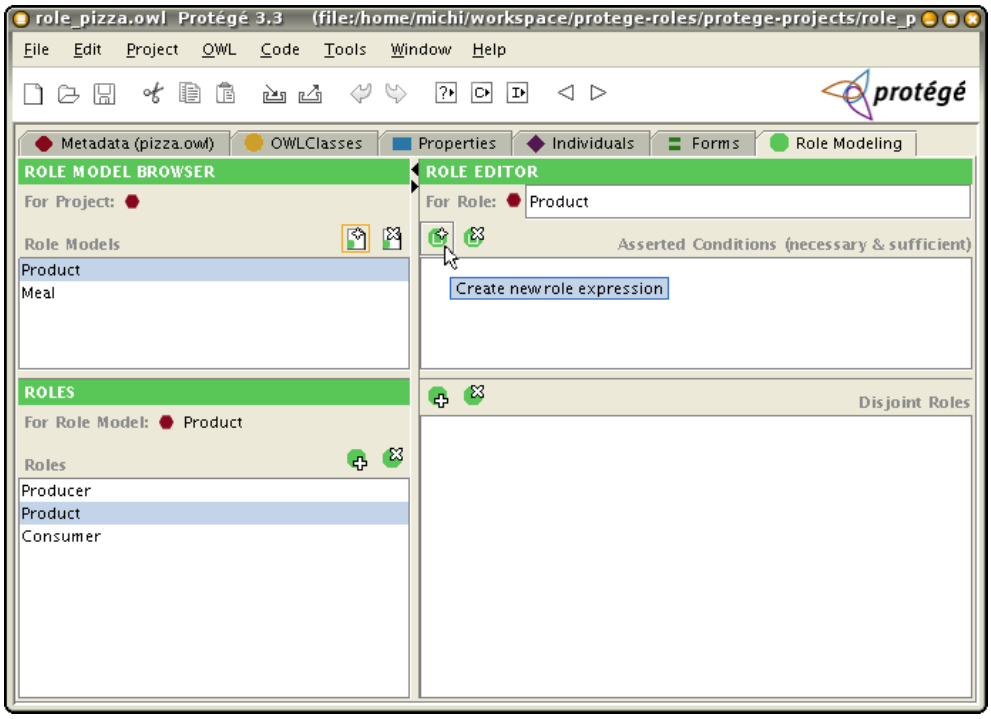

Figure 5.9: A simple role modeling tab for the Protégé ontology editor.

Other issues also remain to be further clarified. The semantics of roles may be subject of discussion. Apart from focusing on *can-play* semantics, *must-play* may in some cases be desirable for role bindings. It would be interesting to define different translations of role-based ontologies into classical ontologies providing different semantics. Another issue to clarify is the implication of applying one role model several times in an ontology. One could argue for multiple imports where each import is associated with a unique name space. However, this would disallow to refer to all individuals of a certain role type, for instance to all products in an ontology. Finally, further investigations into the implications of the open-world semantics of ontologies relating to role bindings and role assertions should be performed.

A final warning

The ideas of this chapter are highly experimental and could not be investigated in the desired extent during this work. However, we believe them to be worth mentioning as outlook and thought-provoking impulse for future research.

6 Conclusion

In this work, we have merged the role modeling paradigm with ontologies. We have taken the first steps to transfer the idea of roles from object-oriented software development to ontologies and propose ontological role models as a novel reuse abstraction. Role models provide a notion of ontological components and allow to enhance the insufficient reuse mechanisms of existing ontology languages. Furthermore, roles enable more natural modeling since natural types and role types can be distinguished explicitly. Due to the translational semantics, our approach is compatible with existing formalisms and tools.

The main contributions of this work are the following:

- We present the conceptual idea of ontological roles and illustrate their usefulness with examples.
- We define ontological roles and role models on a conceptual level as well as formally.
- We propose a methodology for role-based ontology engineering.
- We provide a translational semantics and an implementation of it.

During this work, a number of new questions emerged that should be subject of further research. The idea of ontological role modeling should be further developed on a conceptual level. At first, the semantics of roles may be subject of discussion. Furthermore, other uses of ontology roles should be analyzed, for instance, composition of role-based ontologies. Beyond addressing conceptual questions, the ideas of this work should be developed on a practical level. As a next step, integrating roles into existing ontology modeling tools would be important. Moreover, one should study ontologies of a larger size and come up with a role model catalog.

Overall, we argue that roles provide an interesting reuse abstraction for ontologies and thus, should be supported as an ontological primitive.

A Role Models

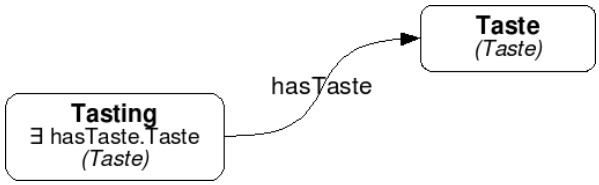

Figure A.1: *Taste* role model.

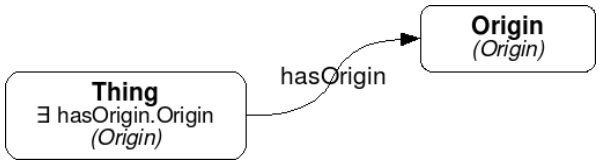

Figure A.2: *Origin* role model.

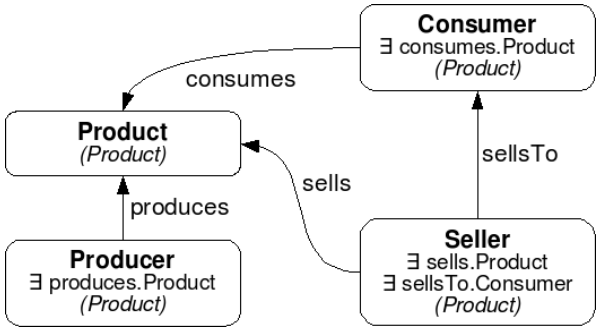

Figure A.3: *Product* role model.

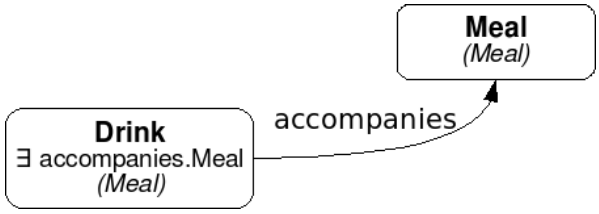

Figure A.4: *Meal* role model.

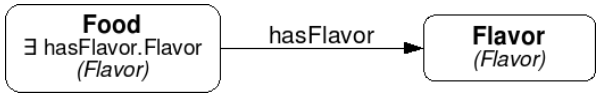

Figure A.5: *Flavor* role model.

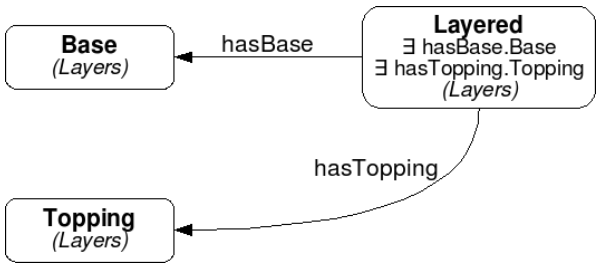

Figure A.6: *Layers* role model.

B Source Code

Languages

```
Ontology            = statements:OntologyStatement*;

OntologyStatement   = ClassDescription | PropertyDescription | Assertion;

ClassDescription    = classID:NamedType, description:Description*;

Description         = SubClassOf | EquivalentTo | DisjointWith;

SubClassOf          = c:ClassExpression;
EquivalentTo        = c:ClassExpression;
DisjointWith        = c:ClassExpression;

ClassExpression     = AtomicExpression | Conjunction | Disjunction | Existential |
                      Universal;

Conjunction         = conjunction:ClassExpression+;
Disjunction         = disjunction:ClassExpression+;
Existential         = prop1:NamedProperty, class1:ClassExpression;
Universal           = prop2:NamedProperty, class2:ClassExpression;

AtomicExpression    = NamedTypeEx;

PropertyDescription = propID:NamedProperty, domain:NamedType, range:NamedType;

Assertion           = ClassAssertion | PropertyAssertion;
ClassAssertion      = type:NamedType, indiv:NamedIndiv;
PropertyAssertion   = prop:NamedProperty, indiv1:NamedIndiv, indiv2:NamedIndiv;

NamedTypeEx         = atom:NamedType;

NamedType           = value:S;
NamedProperty       = value:S;
NamedIndiv          = value:S;
```

Listing B.1: languages/owlm.as

```
CONCRETESYNTAX owlm FOR owlm

Ontology              ::= "Ontology:" statements*;

ClassDescription      ::= "Class:" classID description*;

SubClassOf            ::= "SubClassOf:" c;
EquivalentTo          ::= "EquivalentTo:" c;
DisjointWith          ::= "DisjointWith:" c;

NamedTypeEx           ::= atom;

Conjunction           ::= "(" conjunction ("AND" conjunction)* ")" ;
Disjunction           ::= "(" disjunction ("OR" disjunction)* ")";
Existential           ::= "(" prop1 "SOME" class1 ")";
Universal             ::= "(" prop2 "ONLY" class2 ")";

PropertyDescription   ::= "Property:" propID "Domain:" domain "Range:" range;

ClassAssertion        ::= type "(" indiv ")";
PropertyAssertion     ::= prop "(" indiv1 "," indiv2 ")";

NamedType             ::= value[('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' |
                          '0'..'9' | '_' | '-')*];
NamedProperty         ::= value[('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' |
                          '0'..'9' | '_' | '-')*];
NamedIndiv            ::= value[('A'..'Z' | 'a'..'z') ('A'..'Z' | 'a'..'z' |
                          '0'..'9' | '_' | '-')*];
```

Listing B.2: languages/owlm.cs

```
owlm.OntologyStatement    = ImportStmt;

ImportStmt                = filename:componentmodel.Location;
ImportStmt                ==> minimalcl.Composer;

Plays                     = roleID:owlm.NamedType;
Plays                     ==> minimalcl.Composer;
owlm.Description          = Plays;

RoleModel                 = modelID:owlm.NamedType, stmts:RoleStatement*;
RoleStatement             = RoleDescription | RolePropDescription;
RoleDescription           = roleID:owlm.NamedType, descriptions:owlm.EquivalentTo*;
RolePropDescription       = propID:owlm.NamedProperty, domain:owlm.NamedType,
     range:owlm.NamedType;

AtomicExpressionHook;
owlm.AtomicExpression     = AtomicExpressionHook;
AtomicExpressionHook      ==> componentmodel.Hook;
```

Listing B.3: languages/reuseowlm.as

```
CONCRETESYNTAX rowlm FOR reuseowlm EXTENDS owlm

owlm.Ontology                       ::= "Ontology:" statements*;

ImportStmt                          ::= "import" filename;

Plays                               ::= "Plays:" roleID;

RoleModel                           ::= "Rolemodel:" modelID stmts*;

RoleDescription                     ::= "Role:" roleID descriptions*;

RolePropDescription                 ::= "Role Property:" propID "Domain:" domain
                                        "Range:" range;

AtomicExpressionHook                ::= "<<" name ">>";

componentmodel.FragmentName         ::= name[('A'..'Z' | 'a'..'z') ('A'..'Z' |
                                        'a'..'z' | '0'..'9' | '_' | '-')*];
componentmodel.Location             ::= path[('http' ':' '/')? ('/' ('A'..'Z'|
                                        'a'..'z' | '0'..'9' | '_' | '.' | '-')+)+];

componentmodel.VariationPointName   ::= name[('A'..'Z' | 'a'..'z') ('A'..'Z' |
                                        'a'..'z' | '0'..'9' | '_' | '-')*];
```

Listing B.4: languages/rowlm.cs

Fragments

```
Rolemodel: Taste
   Role: Tasting
   EquivalentTo: (hasTaste SOME Taste)
   Role: Taste

   Role: Open

   Role Property: hasTaste
      Domain: Taste
      Range: Tasting
```

Listing B.5: fragments/components/Taste.rowlm

```
define composer reuseowlm.ImportStmt(filename) {
// realizes 'import' statement by copying role models into the ontology

    // simply ontology to extend with role-classes and to be
    // included in the importing ontology
    fragmentlist owlm.Ontology roleOntology =
        'Ontology: Class: Top'.rowlm;

    fragmentlist reuseowlm.RoleModel rm = →filename;

    foreach (stmt : rm.stmts) {
        if (stmt instanceof reuseowlm.RoleDescription) {
            // transform all roles into classes and copy their definition
            fragmentlist owlm.ClassDescription newCls =
                'Class: ' + stmt.roleID + ''.rowlm;
            extend newCls.description with stmt.descriptions;

            // add hook for new subclasses (to be extended by plays-composer)
            fragmentlist owlm.SubClassOf extSCO =
                'SubClassOf: ( Bottom OR <<' +
                stmt.roleID + 'SubClassHook>> )'.rowlm;
            extend newCls.description[first] with extSCO;

            // add role to base-ontology
            extend roleOntology.statements[first] with newCls;
        }

        if (stmt instanceof reuseowlm.RolePropDescription) {
            // transform role properties into properties and copy their definition
            fragmentlist owlm.PropertyDescription newProp =
                'Property: ' + stmt.propID + ' Domain:' + stmt.domain +
                ' Range:' + stmt.range + ''.owlm;
            extend roleOntology.statements[first] with newProp;
        }

    }
    return roleOntology.statements;
}
```

Listing B.6: fragments/composers/importComposer.bc

```
define composer reuseowlm.Plays(roleID) {
// transforms 'plays' statement to subclass relationships

    // get surrounding ontology and class
    fragmentlist owlm.Ontology ontology = CTX_ROOT;
    fragmentlist owlm.ClassDescription clsDesc = CTX_CONTAINER;

    // role-playing class - we add it as possible subclass of the role
    fragmentlist owlm.ClassExpression clsExpr = clsDesc.classID+''.owlm;

    // a hook with this name was added by the import-composer
    fragmentlist componentmodel.VariationPointName hookName =
        roleID + 'SubClassHook'.bc;

    extend →hookName on ontology with clsExpr;
}
```

Listing B.7: fragments/composers/playsComposer.bc

```
Ontology:

import /Taste.rowlm

Class: Spicy
   Plays: Taste

Class: Pizza
   Plays: Tasting

Class: Pasta
   Plays: Taste
   EquivalentTo: ((accompaniedBy SOME Sauce) OR (Noodles AND Italian))

Property: hasOrigin
   Domain: Pasta
   Range: Country

Pasta(myPasta)
hasTaste(myPasta, hot)
```

Listing B.8: fragments/compositionsPrograms/ontology.rowlm

Bibliography

[ADN+03] Heidrun Allert, Peter Dolog, Wolfgang Nejdl, Wolf Siberski, and Friedrich Steimann.
Role-oriented models for hypermedia construction - conceptual modeling for the semantic web.
Technical report, 2003.

[Ass03] U. Assmann.
Invasive Software Composition.
Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[Ass05] Uwe Assmann.
Reuse in semantic applications.
In *Proceedings of Summer School Reasoning Web 2005, Msida, Malta (25th–29th July 2005)*, volume 3564 of *LNCS*, pages 290–304. REWERSE, 2005.

[AZW06] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner.
Ontologies, Meta-Models, and the Model-Driven Paradigm, pages 249–273.
Springer, 2006.

[Bac80] Charles W. Bachman.
The role data model approach to data structures.
In *International Conference on Databases (ICOD)*, pages 1–18, 1980.

[Bao06] Honavar Bao.
Divide and conquer semantic web with modualr ontologies - a brief review of modular ontology language proposals.
In *Proceedings of the First International Workshop on Modular Ontologies (WoMo2006)*, 2006.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors.
The description logic handbook: theory, implementation, and applications.
Cambridge University Press, New York, NY, USA, 2003.

[BLF01] Tim Berners-Lee and Mark Fischetti.
Weaving the web: The original design and ultimate destiny of the world wide web by its inventor.
DIANE Publishing Company, 2001.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila.
The semantic web.
Scientific American, 2001.

[BRSW97] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf.
The role object pattern.
In *Proceedings of the Conference on Pattern Languages of Programs (PLoP '97)*, 1997.

[BS03] François Bry and Sebastian Schaffert.
The XML query language Xcerpt: Design principles, examples, and semantics.
In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 295–310, London, UK, 2003. Springer-Verlag.

[COM07] The COMPOST Consortium.
The COMPOST system portal.
http://www.the-compost-system.org/, 2007.

[dBEF+06] Jos de Bruijn, Marc Ehrig, Cristina Feier, Francisco J. Martin-Recuerda, Francois Scharffe, and Moritz Weiten.
Ontology mediation, merging and aligning.
John Wiley & Sons, 2006.

[DHS+07] Nick Drummond, Matthew Horridge, Robert Stevens, Chris Wroe, and Sandra Sampaio.
Pizza ontology v1.5.
http://www.co-ode.org/ontologies/pizza/2007/02/12/, February 2007.

[DMQ05] Dejing Dou, Drew V. McDermott, and Peishen Qi.
Ontology translation on the semantic web.
Journal of Data Semantics, 2:35–57, 2005.

[DS04] Mike Dean and Guus Schreiber.
OWL Web Ontology Language reference.
W3C recommendation, W3C, February 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design patterns: Elements of reusable object-oriented software.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHKS07] Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler.
A logical framework for modularity of ontologies.
In Manuela M. Veloso, editor, *Proceedings of IJCAI'07: the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6–12, 2007*, pages 298–303, 2007.

[Gru93] Thomas R. Gruber.
A translation approach to portable ontology specifications.
Knowledge Acquisition, 5(2):199–220, 1993.

[GS94] David Garlan and Mary Shaw.
An introduction to software architecture.
Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[HAJZ07] Jakob Henriksson, Uwe Assmann, Jendrik Johannes, and Steffen Zschaler.
Reuseware - Adding modularity to your language of choice.
In *Proceedings of Technology of Object-Oriented Languages and Systems Europe 2007, Zurich, Switzerland (24th–27th June 2007)*, 2007.

[HDG+06] Matthew Horridge, Nick Drummond, John Goodwin, Alan Rector, Robert Stevens, and Hai Wang.
The Manchester OWL syntax.
OWL: Experiences and Directions (OWLED), November 2006.

[Int96] International Organization for Standardization.
ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF.
1996.

[KFNM04] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen.
The Protege OWL plugin: An open development environment for semantic web applications.
Third International Semantic Web Conference (ISWC), November 2004.

[Kle01] M. Klein.
Combining and relating ontologies: an analysis of problems and solutions.
In A. Gomez-Perez, M. Gruninger, H. Stuckenschmidt, and M. Uschold, editors, *Proceedings of the Workshop on Ontologies and Information Sharing, IJCAI'01*, Seattle, USA, 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin.
Aspect-oriented programming.
In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1997)*, pages 220–242, 1997.

[LT05] Patrick Lambrix and He Tan.
A framework for aligning ontologies.
In *Principles and Practice of Semantic Web Reasoning 2005: Third International Workshop, PPSWR*, pages 17–31, 2005.

[Min74] Marvin Minsky.
A framework for representing knowledge.
Technical report, Cambridge, MA, USA, 1974.

[MW04] Prasenjit Mitra and Gio Wiederhold.
An ontology-composition algebra.
In *Handbook on Ontologies*, Springer Series on Handbooks in Information Systems, pages 93–116. Springer, 2004.

[Noy04] Natalya Fridman Noy.
Tools for mapping and merging ontologies.
In *Handbook on Ontologies*, Springer Series on Handbooks in Information Systems, pages 365–384. Springer, 2004.

[OMG05] Object Management Group OMG.
Unified modeling language: Superstructure, version 2.0, formal/05-07-04, August 2005.

[OT00] H. Ossher and P. Tarr.
Multi-dimensional separation of concerns and the hyperspace approach.
In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[PSZ06] Jeff Z. Pan, Luciano Serafini, and Yuting Zhao.
Semantic import: An approach for partial ontology reuse.
In *Proc. of the ISWC2006 Workshop on Modular Ontologies (WoMO)*, 2006.

[Qui67] M. R. Quillian.
Word concepts: a theory and simulation of some basic semantic capabilities.
Behavioral Science, 12(5):410–430, September 1967.

[Ree96] Wold P.-Lehne O. A. Reenskaug, T.
Working with Objects, The OOram Software Engineering Method.
Manning Publications Co, 1996.

[RG98] Dirk Riehle and Thomas Gross.
Role model based framework design and integration.
In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 117–133, New York, NY, USA, 1998. ACM Press.

[Rie96] D. Riehle.
Describing and composing patterns using role diagrams.
In *Proceedings of the 1st International Conference on Object-Oriented Technology in Russia (WOON '96)*, 1996.

[Rie00] Dirk Riehle.
Framework Design: A Role Modeling Approach.
PhD thesis, 2000.

[SB02] Yannis Smaragdakis and Don Batory.
Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs.
ACM Transactions on Software Engineering and Methodology (TOSEM), 11(2):215–255, 2002.

[Sow84] J. F. Sowa.
Conceptual structures: information processing in mind and machine.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Ste00] Friedrich Steimann.
On the representation of roles in object-oriented and conceptual modelling.
Data Knowledge Engineering, 35(1):83–106, 2000.

[Ste01] Friedrich Steimann.
Role = interface: a merger of concepts.
Journal of Object-Oriented Programming, 14(4):23–32, 2001.

[Ste05] Friedrich Steimann.
The role data model revisited.
Roles, an interdisciplinary perspective, AAAI Fall Symposium, 2005.

[Szy02] Clemens Szyperski.
Component Software: Beyond Object-Oriented Programming.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[VJBCS97] Pepjijn R. S. Visser, Dean M. Jones, T. J. M. Bench-Capon, and M. J. R. Shave.
An analysis of ontological mismatches: Heterogeneity versus interoperability.
In *AAAI 1997 Spring Symposium on Ontological Engineering*, Stanford, USA, 1997.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, September 20, 2007