Technische Universität Dresden
Herausgeber: Der Rektor

Using Bonds for Describing Method Dispatch in Role-Oriented Software Models

Henri Mühle

MATH–AL–04–2010 August 2010

Abstract. Role-oriented software modeling is an approach to object-oriented software engineering which provides a stricter encapsulation by separating the type behavior from the object into so-called *roles*. This role behavior can dynamically be accessed in certain situations and extends or alters the original type behavior. The process of extending or altering type behavior in object-oriented systems is realized by so-called method dispatch which controls message sending and routing. It is thus essential to guarantee the correct execution of the model.

In this report we present a context-based construction to describe the method dispatch via special formal contexts containing bonds. It turns out that the bond-induced morphisms serve well for determining the role method which is bound to a certain base method during runtime. This formal context can also be used to check the role model and determine whether base and role methods are bound correctly.

1 Introduction

Role orientation is an approach to object-oriented software modeling that relies on separating the behavior from the object. It was introduced in the 1990s by Trygve Reenskaug [11] and later investigated by Friedrich Steimann [12], who also gave a first formalisation of this approach, along with a proposal for an UML notation of these concepts. Role types encapsulate common behavior that is required in certain situations into separate modules. In contrast to subclassing or delegation – as standard techniques in object-orientation for encapsulating and altering behavior – role types allow for flexible and dynamic change of behavior without reinstantiating the object.

The scope of this report lies in formalizing the method dispatch in role-oriented software models. Method dispatch is a mechanism in object-oriented software models that determines and invokes the correct piece of code for a certain method call [8]. Subsidiary to method dispatch

along the inheritance hierarchy in standard object-oriented models, role-oriented modeling adds another dimension of method dispatch along the role-play relation. It is our goal to provide a sound, concept-based representation of this kind of method dispatch. However, our approach shall not be seen as a mechanism to implement method dispatch in role-oriented languages in order to allow for better performance. It shall serve as a design aid with whose help role modelers can check their models for correctness and may receive design advices to improve their models.

The rest of the paper is organized as follows: In Section 2 we present the basics of Formal Concept Analysis (Section 2.1) as well as a short introduction to the concept of roles (Section 2.2.1) and its application to software modeling (Section 2.2.2). We construct our model for representing role models in Section 3 in order to determine the method dispatch. It is necessary to construct a static model first (Section 3.1) which will then be extended towards a dynamic model (Section 3.2). Concluding this section we present a combined model that includes both – base and role – type hierarchies (Section 3.3). The next section (Section 4) is used to present our dispatch algorithm as well as an example for clarification. Concluding this paper, Section 5 summarizes our results and Section 6 gives an outlook towards future work.

2 Preliminaries

2.1 Formal Concept Analysis

Formal Concept Analysis (FCA) establishes a connection between binary relations and complete lattices [3]. Its basic elements are *formal contexts*, i. e. triplets (G, M, I) where G is a set of *objects*, M is a set of *attributes* and $I \subseteq G \times M$ describes whether an object *has* an attribute. Introducing two derivation operators for $A \subseteq G$ resp. $B \subseteq M$

$$A^I := \{m \in M \mid \forall\ g \in A : gIm\} \subseteq M$$
$$B^I := \{g \in G \mid \forall\ m \in B : gIm\} \subseteq G$$

one can create *formal concepts* of a formal context (G, M, I) as pairs (A, B) with $A \subseteq G, B \subseteq M, A^I = B, B^I = A$. A is then called *extent*, B is called *intent* of the concept (A, B). With introducing an order relation on the set $\mathfrak{B}(G, M, I)$ of concepts via

$$(A_1, B_1) \leq (A_2, B_2) :\Leftrightarrow A_1 \subseteq A_2 \quad (\Leftrightarrow B_1 \subseteq B_2)$$

the basic theorem of Formal Concept Analysis [3, p. 20] states that the *concept lattice*

$$\underline{\mathfrak{B}}(G, M, I) := (\mathfrak{B}(G, M, I), \leq)$$

is indeed a complete lattice. We will simply write g^I instead of the lengthty notation $\{g\}^I$ if we consider one-element sets.

An interesting way to combine two contexts is done via so-called bonds. Given two contexts $\mathbb{K}_s := (G_s, M_s, I_s), \mathbb{K}_t := (G_t, M_t, I_t)$, a relation $J_{st} \subseteq G_s \times M_t$ is called *bond*, iff $g^{J_{st}}$ is an intent of $\mathbb{K}_t$ for each object $g \in G_s$ and $m^{J_{st}}$ is an extent of $\mathbb{K}_s$ for each attribute $m \in M_t$.

As stated in [2, p. 15] each bond J_{st} induces two morphisms

$$\varphi_{st} : \underline{\mathfrak{B}}(G_s, M_s, I_s) \to \underline{\mathfrak{B}}(G_t, M_t, I_t), \quad \psi_{st} : \underline{\mathfrak{B}}(G_t, M_t, I_t) \to \underline{\mathfrak{B}}(G_s, M_s, I_s)$$

by

$$\varphi_{st}(A, A^{I_s}) := (A^{J_{st}I_t}, A^{J_{st}}), \quad \psi_{st}(B^{I_t}, B) := (B^{J_{st}}, B^{J_{st}I_s})$$

2.2 Role-Oriented Software Modeling

2.2.1 The Concept of a Role

The notion of roles was already introduced in the late 1970s by Bachman and Daya [1]. Though being contrary to the principle of database that each record should represent all aspects of one entity of the world, this notion helped solving a common problem of the usual network model [10]. Since the network model allows the members of a set to be records of various types, it is necessary to write the same redundant piece of

code for each of these types. By encapsulating the respective piece of code into a separate module, the *role*, redundancy of the code can be decreased enormously. [12]

Nevertheless, Bachman and Daya's work did not influence common modeling techniques that much. However, by looking at this concept from a software point of view, we notice that encapsulating pieces of code into separate modules and later accessing and using these if necessary appears as a special form of polymorphism [12]. Thus, the concept of roles should be thoroughly analyzed from an object-oriented point of view. A very extensive and general description of the role concept was given by Steimann [12]. He uniformly described the different approaches towards role modeling that evolved during the 1990s in a formal modeling language called Lodwick.

Today, the role-oriented modeling paradigm extends the object-oriented modeling paradigm by the role concept. Such roles are modules that encapsulate certain behaviour. According to Steimann, roles are placeholders in special relationships that can be filled by *actors* [12, p. 5]. A concept in conjunction with (software) models is a relevant entity of the real world that is represented by the according model[1]. It is necessary to distinguish whether a modeling concept describes a role type or not. Steimann states that Guarino identified two basic properties to separate modeling concepts that describe role types from such modeling concepts that describe natural or (as we will call them) base types: *semantic rigidity* and *foundation* [5].

Semantic rigidity means that a modeling concept determines the identity of its instances, i. e. an instance can not leave the extension of this modeling concept without losing its identity [12, p. 3]. E. g. a *person* is a semantically rigid modeling concept, since noone can stop being a person without giving up its identity. A *student*, however, is no semantically rigid modeling concept. One can stop being a student at any time without necessarily giving up the own identity.

Foundation means that a modeling concept needs to interact with some

[1] We will call such concepts *modeling concepts* to avoid confusion with formal concepts in FCA.

other modeling concept, i. e. no instance of this modeling concept can exist on its own [12, p. 3]. E. g. a student is a founded modeling concept, since it is necessary to be registered for a course of studies to be a student. A person, however, is not founded, since there is no interaction with other modeling concepts necessary to be a person.

Guarino concludes that *role types* are such modeling concepts that are not semantically rigid and founded, while *base types* are such modeling concepts that are semantically rigid and not founded [5]. This means, that a student is a role type, while a person is a base type.

2.2.2 Extending Object-Oriented Software Models by the Concept of a Role

With introducing roles as modeling concept, the modeler will obtain several advantages, such as stricter encapsulation or less redundancy. Roles encapsulate special behavior and provide it to type instances when necessary. This property reminds in a certain way of type polymorphism in object-oriented software modeling. Thus it is obvious to extend object-oriented software modeling towards role-oriented software modeling by introducing role types and related modeling concepts.

Steimann outlines some basic properties of role types [12, p. 4] that are also important for our modeling approach:

1. Role types always need a specific situation in which they are appropriate. E. g. a student is a role type in a real (open) world scenario, but if we assume the university as a closed world, a student would be a base type, being able to play role types such as participant in lectures, examinee, etc.

2. Roles come with their own behavior, i. e. role types provide attributes and methods that define the behavior encapsulated in this role.

3. Role types do not provide their own instances. Role types are filled by base types, i. e. base type instances are enriched by the

properties and the behavior specified in the role type.

4. Role types may be acquired and abandoned dynamically by base type instances, i. e. role types have a highly dynamic character.

As a practical example, the programming language ObjectTeams/Java [6] includes these modeling concepts and allows for role-oriented software modeling. Thus it can be used as an orientation to determine which technical difficulties appear in realizing Steimanns formal model. Particularly Property 2 is of crucial interest in realizing role models. This basically says, that role types encapsulate certain behavior. In combination with Property 1 and Property 4 it turns out that this behavior is situational and needs to be dynamically accessible. This is nothing more than following the idea of Bachman and Daya's role concept to avoid code duplication, which somehow reminds of type polymorphism. And indeed, ObjectTeams/Java introduces a so-called *translation polymorphism* [7] as a pendant for type polymorphism as well as *instance dispatch* as a pendant for method dispatch.

While *type polymorphism* in standard object-orientation describes that instances of a certain type can always be regarded as instances of a supertype (and in some cases as instances of a subtype), translation polymorphism in role-orientation describes that instances of base types can always be adressed via the role types that the according base type can play.

Method dispatch is a technique to realize message sending in object-oriented models. Due to late binding (which assesses the types of software objects not until runtime) and type polymorphism the proper target of a message call needs to be determined at runtime as well. Since type methods can be redefined in subtypes the target of a message call is directly dependent of the runtime type of the according type instance. Instance dispatch is the equivalent technique in role-orientation to determine of which role type a certain instance is.

Note that instance dispatch and translation polymorphism *extend* the object-oriented features and do not replace them. This means that e. g. classical method dispatch along the inheritance hierachy is extended by

another dimension: method dispatch along the role-play relation. This extension of method dispatch is closely related to instance dispatch, nevertheless has another quality. In this report we will present a representation of this kind of method dispatch with formal contexts in order to provide a means for model checking or design aid.

3 Model Construction

3.1 Static Models for the Hierarchies

We have already provided a concept-based representation for describing role models and the role-play relation in [9]. Thus it appears consequent to construct a concept-based formalization of the role-oriented method dispatch in order to create a sound and extensive formalization apparatus which could support software designers in their work.

In [9] we used the type names as formal objects, the type attributes as formal attributes as well as the type-attribute-incidence as context incidence to construct formal contexts that represented the role-play relation and allowed for checking consistency of the model as well as dynamically representing and visualizing the runtime state of the role model.

In this report, however, we use the type methods as formal attributes. This approach was proposed in [4] for standard object-oriented models with the goal to restructure and factorize the base type hierarchy. Since base type and role type hierarchy are according to [12] in a sense orthogonal we first describe the hierarchies each and later combine them into a single context. The first step in this process is to construct a very simple context that models only the base resp. role type hierarchies.

Definition 1. Let B be a set of base types, M_B an appropriate set of methods and let I_B describe the type-method-incidence. Thus, the context $\mathbb{B} := (B, M_B, I_B)$ is called **(method-based) base type context**. Let us assume that each method $m \in M_B$ is declared in at least one base type, i. e.

$$\forall\ m \in M_B : m^{I_B} \neq \varnothing \tag{1}$$

Analogously $\mathbb{R} := (R, M_R, I_R)$ is called **(method-based) role type context**.

For better understanding of the presented constructions, we will refer to a small running example of a lecture. The role model[2] of this lecture is shown in Figure 1, the base and role type contexts are shown in Figure 2.

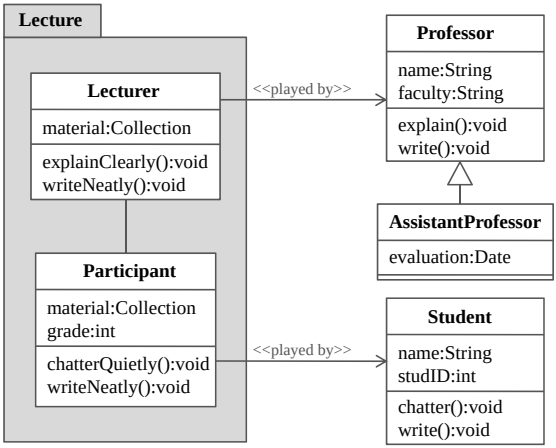

Figure 1: An example role model *Lecture*

I_B	write()	explain()	chatter()
Professor	×	×	
AssistantProfessor	×	×	
Student	×		×

I_R	writeNeatly()	explainClearly()	chatterQuietly()
Lecturer	×	×	
Participant	×		×

Figure 2: The method-based base type and role type contexts of the role model *Lecture* in Figure 1

[2]We defined a role model in [9] as a formal context (B, R, P), where B is a set of base types, R is a set of role types and $P \subseteq B \times R$ describes the role-play relation.

For describing the method dispatch, we need to adress the single methods of each base or role type unambiguously. Therefore we have to extend the objects of base and role type contexts by representatives of these methods. We call these representatives *virtual objects*. They basically encode the incidence relation in the object set of the base resp. role type context, because for each base resp. role type we only need to introduce such virtual objects representing a method that is possessed by this type.

Definition 2. Let (B, M_B, I_B) be a base type context. Introducing a set V_B of **virtual objects**, the **extended base type context** $\hat{\mathbb{B}} := (\hat{B}, M_B, \hat{I}_B)$ is defined as follows:

$$\begin{aligned} \hat{B} &:= B \cup V_B \\ \hat{I}_B &:= I_B \cup N_B \end{aligned}$$

with

$$\begin{aligned} V_B &:= \big\{(b, m) \mid \exists\ b \in B, m \in M_B : b I_B m\big\} \\ N_B &:= \big\{(v, m) \mid \exists\ v \in V_B : v = (b, m)\big\} \end{aligned}$$

The **extended role type context** $\hat{\mathbb{R}} := (\hat{R}, M_R, \hat{I}_R)$ is defined analogously.

For reasons of readability we will denote the virtual objects in V_B (V_R) with the first two letters of the respective base (role) type, followed by the column number of the base (role) type method this virtual object refers to. Figure 3 shows the extended base and role type contexts of Figure 1. Note that each of the method-based contexts is a subcontext of the respective extended type context.

Since the virtual objects in a sense represent the type methods, the following lemma holds for each extended type context.

Lemma 1. *Let $\hat{\mathbb{K}} = (\hat{G}, M, \hat{I})$ be a extended type context in the sense of Definition 2. Then, $\hat{\mathbb{K}}$ is co-atomistic, i. e.*

$$\forall\ m_1, m_2 \in M : m_1^{\hat{I}} \subseteq m_2^{\hat{I}} \Rightarrow m_1 = m_2$$

Proof. Let $m_1, m_2 \in M$. By definition of $\hat{I}$ and Assumption 1 there must

$\hat{I}_B$	write()	explain()	chatter()
Professor	×	×	
Pr1	×		
Pr2		×	
AssistantProfessor	×	×	
As1	×		
As2		×	
Student	×		×
St1	×		
St3			×

$\hat{I}_R$	writeNeatly()	explainClearly()	chatterQuietly()
Lecturer	×	×	
Le1	×		
Le2		×	
Participant	×		×
Pa1	×		
Pa3			×

Figure 3: The extended base type and role type contexts of Figure 1

exist virtual objects $v_1, v_2 \in V \subseteq \hat{G}$ with $v_1^{\hat{I}} = \{m_1\}$ and $v_2^{\hat{I}} = \{m_2\}$. Thus $\{m_1\}$ and $\{m_2\}$ are intents of $\hat{\mathbb{K}}$.

Now we can easily show the desired property:

$$\{m_1\}^{\hat{I}} \subseteq \{m_2\}^{\hat{I}} \Leftrightarrow \{m_1\}^{\hat{I}\hat{I}} \supseteq \{m_2\}^{\hat{I}\hat{I}} \Leftrightarrow \{m_1\} \supseteq \{m_2\} \Leftrightarrow m_1 = m_2$$

□

3.2 Dynamic Models for the Base Type Hierarchy

Since we want to describe the method dispatch during runtime it is necessary to describe the runtime instances as a formal context as well. According to Property 3 role types do not provide their own instances, but are played by base type instances.

For describing instances we will introduce the following notation: the runtime of the system will be denoted by $T \subseteq \mathbb{N}$. $\mathcal{I}^t$ is the set of all active instances at the point of (run)time $t \in T$. We further have $\mathcal{I}_b^t \subseteq \mathcal{I}^t$ as the set of all active instances of base type $b \in B$ and $\mathcal{I}_r^t \subseteq \mathcal{I}^t$ as the set of all active instances playing a role type $r \in R$. We will additionally introduce $\mathcal{I}_{b,r}^t := \mathcal{I}_b^t \cap \mathcal{I}_r^t$. Obviously holds:

$$\bigcup_{b \in B} \mathcal{I}_b^t = \mathcal{I}^t, \qquad \bigcup_{r \in R} \mathcal{I}_r^t \subseteq \mathcal{I}^t$$

The following context describes the instances which receive certain method calls.

Definition 3. Let $\hat{\mathbb{B}} = (\hat{B}, M_B, \hat{I}_B)$ be an extended base type context. We will introduce the **method call context** $\hat{\mathbb{B}}^t := (\hat{B}^t, M_B, \hat{I}_B^t)$ as follows:

$$\begin{aligned} \hat{B}^t &:= \hat{B} \cup \mathcal{I}^t \\ \hat{I}_B^t &:= \hat{I}_B \cup C^t \end{aligned}$$

where

$$C^t := \{(i, m) \mid \exists\, i \in \mathcal{I}_b^t, m \in M_B : i \text{ receives a call from } m\}$$

It has to be said that we assume a sequential execution of our role model which means that each instance can only receive calls from one single method at a time. This is sufficient since one can easily map parallel activities to a sequential execution plan.

To assign the co-atomicity of the extended base type context to the method call context, we need to prove that the concept lattices of either contexts are isomorphic.

Lemma 2. *Let $\hat{\mathbb{B}} = (\hat{B}, M_B, \hat{I}_B)$ be an extended base type context and $\hat{\mathbb{B}}^t = (\hat{B}^t, M_B, \hat{I}_B^t)$ the respective method call context. Then it is $\underline{\mathfrak{B}}(\hat{\mathbb{B}}) \cong \underline{\mathfrak{B}}(\hat{\mathbb{B}}^t)$.*

Proof. According to the basic theorem of FCA [3, p. 20] we need to find mappings $\hat{\gamma} : \hat{B}^t \to \underline{\mathfrak{B}}(\hat{\mathbb{B}}), \hat{\mu} : M_B \to \underline{\mathfrak{B}}(\hat{\mathbb{B}})$, s. t. $\hat{\gamma}(\hat{B}^t)$ is supremum-dense in $\underline{\mathfrak{B}}(\hat{\mathbb{B}})$, $\hat{\mu}(M_B)$ is infimum-dense in $\underline{\mathfrak{B}}(\hat{\mathbb{B}})$ and $b\hat{I}_B^t m \Leftrightarrow \hat{\gamma} b \leq \hat{\mu} m$.

Since the attribute sets of both contexts agree $\hat{\mu} : M_B \to \underline{\mathfrak{B}}(\hat{\mathbb{B}}), m \mapsto \mu_{\hat{\mathbb{B}}}(m)$ will serve as the attribute mapping. For the object mapping we introduce an equivalence relation $\theta^t := \{(b_1, b_2) \mid b_1^{\hat{I}_B^t} = b_2^{\hat{I}_B^t}\}$ on $\hat{B}^t$ that identifies all virtual objects that describe the same method with each other and with all instances that call this method. If we reduce the contexts $(\hat{B}^t/\theta^t, M_B, \hat{I}_B^t)$ and $(\hat{B}, M_B, \hat{I}_B)$ the respective sets of objects are obviously isomorphic. (The reduction is necessary, since θ^t removes

duplicate and empty rows from $\hat{\mathbb{B}}^t$.) Let $j^t : \hat{B}^t/\theta^t \to \hat{B}$ denote this isomorphism. Thus, $\hat{\gamma} : \hat{B}^t \to \underline{\mathfrak{B}}(\hat{\mathbb{B}}), \hat{b} \mapsto \gamma_{\hat{\mathbb{B}}}\big(j^t[\hat{b}]_{\theta^t}\big)$ has the desired properties.

The condition $b\hat{I}_B^t m \Leftrightarrow \hat{\gamma}b \leq \hat{\mu}m$ is immediately satisfied for $b \in \hat{B}$. Thus, we only have to check the condition for $b \in \mathcal{I}^t$. Since we assumed a sequential execution of the role model, we know that $|b^{\hat{I}_B^t}| = 1$. By construction of $\hat{B}$ there needs to exist $v \in V_B$ having $b^{\hat{I}_B^t} = v^{\hat{I}_B^t}$. Thus it is $b\theta^t v$. Reducing the context $(\hat{B}^t/\theta^t, M_B, \hat{I}_B^t)$ means, that there exists exactly one object having the same intent as b (and v). W. l. o. g. let $j^t([b]_{\theta^t}) = v$. Thus we have

$$\begin{aligned}
\hat{\gamma}b \leq \hat{\mu}m &\Leftrightarrow \gamma_{\hat{\mathbb{B}}}\big(j^t[b]_{\theta^t}\big) \leq \mu_{\hat{\mathbb{B}}}(m) \\
&\Leftrightarrow \gamma_{\hat{\mathbb{B}}}(v) \leq \mu_{\hat{\mathbb{B}}}(m) \\
&\Leftrightarrow v\hat{I}_B m \\
&\Leftrightarrow v\hat{I}_B^t m \\
&\Leftrightarrow b\hat{I}_B^t m
\end{aligned}$$

□

Thus the method call context is co-atomistic as well. This enables us to show the following lemma.

Lemma 3. *Let $\hat{\mathbb{B}}^t = (\hat{B}^t, M_B, \hat{I}_B^t)$ be a method call context. For each $(C, D) \in \mathfrak{B}(\hat{\mathbb{B}}^t)$ with $C \cap \mathcal{I}^t \neq \varnothing$ follows that $|D| \leq 1$.*

Proof. Obvious, since we assumed a sequential execution, i. e. each instance can only call at most one method at a time, and $\hat{\mathbb{B}}^t$ is co-atomistic. We have $|D| = 0$, iff the instances in $C \cap \mathcal{I}^t$ do not call any methods at all. □

3.3 Combining both Hierarchies

Since the kind of method dispatch that shall be described by our approach redirects calls from base type methods towards role type methods, it is

necessary to establish a connection between base and role type hierarchies. This means that we need to establish a connection between the attribute sets of either contexts. This connection is constructed in terms of a mapping between the concepts of either contexts. Such a mapping can be constructed in a natural way by a so-called bond between both contexts (cf. Section 2.1).

We will first explain our construction and then, in the next section, present an algorithm for describing the dispatch.

Definition 4. Let $\mathbb{C} = (B, R, P)$ be a role model and $\hat{\mathbb{B}} = (\hat{B}, M_B, \hat{I}_B)$ resp. $\hat{\mathbb{R}} = (\hat{R}, M_R, \hat{I}_R)$ be the extended base resp. role type contexts. The formal context (G, M, J) with

$$\begin{aligned} G &:= \hat{B} \cup \hat{R} \\ M &:= M_B \cup M_R \\ J &:= \hat{I}_B \cup \hat{I}_R \cup J_{BR} \end{aligned}$$

such that $J_{BR} \subseteq \hat{B} \times M_R$ is called **binding context** of $\mathbb{C}$.

If J_{BR} forms a bond between $(\hat{B}, M_B, \hat{I}_B)$ and $(\hat{R}, M_R, \hat{I}_R)$ and fulfills the following conditions

$$\forall\ b \in B \subseteq \hat{B}, m \in M_R : (b, m) \in J_{BR} \Leftrightarrow \exists\ r \in m^{\hat{I}_R} : bPr \tag{2}$$

$$\forall\ v = (b, m) \in V_B \subseteq \hat{B} : v^J \subseteq b^J \tag{3}$$

$$\forall\ v \in V_B : |v^{J_{BR}}| = 1 \tag{4}$$

$$\forall\ m \in M_R : m^{J_{BR}} \neq \varnothing \tag{5}$$

(G, M, J) is called **proper binding context**.

The first two conditions describe that the bond combines only such base types (and their respective virtual objects) with role methods if the according base type can play the according role type. The third condition says that each virtual object needs to be bound to exactly one role method. This is a comprehensible claim, since virtual objects in a sense represent base type methods and we assumed that each base type method is bound to exactly one role type method. And lastly, the fourth condition says that for each role type method there needs to exist at least one

virtual object (and thus at least one base type method) that is bound to this role type method.

4 Performing the Method Dispatch

It is essential for role modeling to determine the method dispatch between base types and the according role types they play at runtime. At modeling time (i. e. when setting up the role model) each method $m_b \in b^{I_B}$ of a base type $b \in B$ is assigned to a method $m_r \in r^{I_R}$ of a respective role type $r \in R$ that is played by b in order to alter the behavior of the base type when playing this role. At runtime it is necessary to correctly dispatch method calls from the base type method to the appropriate role type method to guarantee correct altering of the behavior.

For resolving the method dispatch, we will use the bond-induced morphisms φ_{BR} and ψ_{BR} of the proper binding context as recalled in Section 1.

Let $t \in T$ be a point of runtime and $i \in \mathcal{I}^t$ a certain instance calling a method $m \in M_B$. Algorithm 1 shows how the role method $\tilde{m} \in M_R$ that is bound to m can be determined.

Algorithm 1 An algorithm for method dispatch

Require: method call context $\mathbb{B}^t = (\hat{B}^t, M_B, \hat{I}_B^t)$,
proper binding context (G, M, J),
instance $i \in \mathcal{I}^t$,
method $m \in M_B$
Ensure: $m \in i^{\hat{I}_B^t}$
1: $c := (m^{\hat{I}_B}, m^{\hat{I}_B \hat{I}_B})$
2: $\tilde{c} := \varphi_{BR}(c)$
3: $\tilde{m} := \mathrm{int}(\tilde{c})$
4: **return** $\tilde{m}$

The algorithm requires the appropriate method call context as well as the proper binding context and gets an active instance $i \in \mathcal{I}^t$ as well as a

base type method $m \in M_B$ as inputs. It needs to be ensured that i indeed calls m. Applying the bond-induced morphism φ_{BR} from the base type context to the role type context to the attribute concept of m we receive a concept having only role methods in its intent (Line 2). We now have to show that the intent of this concept indeed consists of only one attribute to state that the method dispatch can be performed uniquely (Line 3).

Lemma 4. *Let (G, M, J) be a proper binding context, $(C, D) \in \mathfrak{B}(G, M, J)$ with $|D \cap M_B| = 1$. Then for the concept $(\tilde{C}, \tilde{D}) := \varphi_{BR}(C, D)$ holds $|\tilde{D}| = 1$.*

Proof. Let $(C, D) \in \mathfrak{B}(G, M, J)$ be as desired and $m \in M_B$. W.l.o.g. $m \in D$. By construction of (G, M, J) holds that $R \times M_B = \varnothing$. Thus, $C \subseteq \hat{B}$. Since $m \in M_B$ it is $m^J = m^{\hat{I}_B} \subseteq \hat{B}$. By construction of $(\hat{B}, M_B, \hat{I}_B)$ there exists $v \in V_B$ with $m \in v^{\hat{I}_B}$. It follows from Assumption 4 from Definition 4 that $\exists!\ \tilde{m} \in M_R$ with $\{\tilde{m}\} = v^{J_{BR}}$. Since $m = \{D \cap M_B\}$, we have $D = \{m, \tilde{m}\}$. Furthermore we have $C \subseteq m^{\hat{I}_B}$ due to $m \in D$.

Assume, that $\exists\ b \in B : b \in m^{\hat{I}_B} \wedge b \notin C$. This can then only be the case, if $m^{\hat{I}_B} \neq \tilde{m}^{J_{BR}}$. Since J_{BR} is a bond, $\tilde{m}^{J_{BR}}$ has to be an intent of $(\hat{B}, M_B, \hat{I}_B)$. $m^{\hat{I}_B}$ can not be this intent, thus there has to be an attribute $\overline{m} \in M_B$ with $\overline{m}^{\hat{I}_B} = \tilde{m}^{J_{BR}}$. Thus, $\overline{m} \in D$ with $\overline{m} \neq m$. This is in contradiction to the assumption $|M_B \cap D| = 1$.

Thus it follows that $C = m^{\hat{I}_B}$. By definition of φ_{BR} we know that for $(\tilde{C}, \tilde{D}) := \varphi_{BR}(C, D)$ holds $\tilde{D} = C^{J_{BR}} = D \backslash C^{\hat{I}_B} = \{m, \tilde{m}\} \backslash \{m\} = \{\tilde{m}\}$. □

Example 1. A proper binding context for our running example is shown in Figure 4. Let us assume a set $\mathcal{I} := \mathcal{I}_{\text{Professor}} \cup \mathcal{I}_{\text{Student}}$ of instances with $\mathcal{I}_{\text{Professor}} := \{\texttt{Aßmann}, \texttt{Ganter}\}$ and $\mathcal{I}_{\text{Student}} := \{\texttt{Mühle}, \texttt{Wende}\}$. If we further assume a point of runtime where Professor `Aßmann` holds a lecture and explains a situation, Student `Wende` writes down some notes and Student `Mühle` tries to chatter with his neighbor, we receive a method call context as depicted in Figure 5.

We will now explain the algorithm in more detail, by applying it step-

J	write()	explain()	chatter()	wrNeatly()	exClearly()	chQuietly()
Professor	×	×		×	×	
Pr1	×			×		
Pr2		×			×	
AssistantProfessor	×	×		×	×	
As1	×			×		
As2		×			×	
Student	×		×	×		×
St1	×			×		
St3			×			×
Lecturer				×	×	
Le1				×		
Le2					×	
Participant				×		×
Pa1				×		
Pa3						×

Figure 4: A proper binding context of the Lecture from Figure 1. The attribute concept of the attribute *write()* is marked lightgray, while the mapping of this concept under φ_{BR} is marked in a darker gray. The crosses marked both lightgray and darker gray represent the relevant part of the bond that helps mapping the respective attributes towards each other.

by-step to the example in Figure 5.

$\hat{I}_B^t$	write()	explain()	chatter()
Professor	×	×	
Pr1	×		
Pr2		×	
`Aßmann`		×	
`Ganter`			
AssistantProfessor	×	×	
As1	×		
As2		×	
Student	×		×
St1	×		
St3			×
`Mühle`			×
`Wende`	×		

Figure 5: The method call context of a specific situation

Input: $\texttt{Wende} \in \mathcal{I}_{\text{Student}}, \text{write}() \in M_B$

Ensure: $\text{write}() \in \texttt{Wende}^{\hat{I}_B}$

Line 1: $c := (C, D) = \left(\text{write}()^{\hat{I}_B}, \text{write}()^{\hat{I}_B \hat{I}_B}\right)$

$C = \{\text{Professor,Pr1,AssistantProfessor,As1,Student,St1}\}$

$D = \{\text{write}()\}$

Line 2: $\tilde{c} := (\tilde{C}, \tilde{D}) = \varphi_{BR}(c)$

$\tilde{C} = C^{J_{BR}\hat{I}_R} = \{\text{Lecturer,Le1,Participant,Pa1}\}$

$\tilde{D} = D^{J_{BR}} = \{\text{writeNeatly}()\}$

Line 3: $\tilde{m} := \text{int}(\tilde{c}) = \text{writeNeatly}()$

Return: writeNeatly()

For determining the dispatch targets of the other methods that are involved in the presented runtime state of Figure 5 we apply the algorithm analogously. Thus, we receive the following mapping results which are

exactly the intended method bindings.

$$\begin{aligned} \text{explain}() &\mapsto \text{explainClearly}() \\ \text{write}() &\mapsto \text{writeNeatly}() \\ \text{chatter}() &\mapsto \text{chatterQuietly}() \end{aligned}$$

5 Summary and Conclusion

Role-oriented software modeling is an approach towards object-oriented software engineering, gaining a higher encapsulation and modularisation of software models by separating the behavior from the object. The behavior is encapsulated in special modules, *roles*, and is woven into the software model during runtime.

The crucial point in role modeling is the so-called method dispatch, which redirects method calls to base methods towards appropriate role methods and thus enables the intended change of type behavior. Since the model designer determines at modeling time which base methods can be altered by which role methods when playing the respective role, it has to be guaranteed that performing the method dispatch during runtime strictly follows these assignments.

Our approach uses a context construction via so-called bonds to create special formal contexts which uniquely represent the assignment between base and role type methods. We then presented an algorithm using these contexts to determine the role type method that is assigned to a given base type method. Our construction can thus be used to assist the process of role-oriented software modeling.

6 Outlook

Together with the results from [9], where we introduced a basic context representation for base and role type hierarchy, as well as for the role-

play relation, this paper can be seen as a basic fundament for a formal, concept-based description language for role-oriented software modeling.

However, there are still a lot of open fields of research to completely cover role modeling with concept-based constructions. Among others, it is on the one hand necessary to describe composition and decomposition of large role models, e. g. to build kind of a design advisor for role models that helps to identify redundancy or inconsistencies of the model. On the other hand, it is necessary to provide an extensive framework to represent role-play constraints or special characteristics of role-oriented software design, like multiple role play. Since we have already applied FCA successfully towards the foundations of role modeling, a further research in this direction will be very promising.

Acknowledgements

The author would like to thank Christian Wende who supported this work with helpful comments and valuable suggestions.

References

[1] Charles W. Bachman and Manilal Daya. The Role Concept in Data Models. In *VLDB*. IEEE Computer Society, 1977.

[2] Bernhard Ganter. Relational Galois Connections. In *ICFCA 2007 Proceedings*, pages 1–17. Springer, 2007.

[3] Bernhard Ganter and Rudolf Wille. *Formale Begriffsanalyse: Mathematische Grundlagen*. Springer, Heidelberg, 1996.

[4] Robert Godin and Petko Valtchev. Formal Concept Analysis-based Class Hierarchy Design in Object-Oriented Software Development. In *Formal Concept Analysis: Foundations and Applications*, pages 304–323. Springer, 2005.

[5] Nicola Guarino. Concepts, Attributes and Arbitrary Relations. *Data and Knowledge Engineering*, 8:249–261, 1992.

[6] Stephan Herrmann. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology*, 2(2), 2007.

[7] Stephan Herrmann, Christine Hundt, and Katharina Mehner. Translation Polymorphism in Object Teams. Bericht 14369915, 2005.

[8] Wade Holst and Duane Szafron. A General Framework for Inheritance Management and Method Dispatch in Object-Oriented Languages. In *ECOOP 1997 Proceedings*, pages 276–301, 1997.

[9] Henri Mühle and Christian Wende. Describing Role Models in Terms of Formal Concept Analysis. In *ICFCA 2010 Proceedings*, pages 241–255. Springer, 2010.

[10] T. William Olle. *The Codasyl Approach to Data Base Management*. John Wiley & Sons, Inc., New York, NY, USA, 1978.

[11] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning Publications, Greenwich, CT, 1996.

[12] Friedrich Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data Knowledge Engineering*, 35:83–106, 2000.