TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

A FAMILY OF ROLE-BASED LANGUAGES

Thomas Kühn
Born on: 11.09.1985 in Karl-Marx-Stadt (now Chemnitz)

DISSERTATION

to achieve the academic degree

DOKTOR-INGENIEUR (DR.-ING.)

Referee
Prof. Dr. Colin Atkinson
Supervising professors
Prof. Dr. Uwe Aßmann
Prof. Dr.-Ing. Wolfgang Lehner

Submitted on: 3.3.2017
Defended on: 24.3.2017

For my loving children

Statement of authorship

I hereby certify that I have authored this Dissertation entitled *A Family of Role-Based Languages* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 3.3.2017

Thomas Kühn

ABSTRACT

Role-based modeling has been proposed in 1977 by Charles W. Bachman [Bachman et al., 1977], as a means to model complex and dynamic domains, because roles are able to capture both context-dependent and collaborative behavior of objects. Consequently, they were introduced in various fields of research ranging from data modeling via conceptual modeling through to programming languages [Steimann, 2000a]. More importantly, because current software systems are characterized by increased complexity and context-dependence [Murer et al., 2008], there is a strong demand for new concepts beyond object-oriented design. Although mainstream modeling languages, i.e., Entity-Relationship Model, Unified Modeling Language, are good at capturing a system's structure, they lack ways to model the system's behavior, as it dynamically emerges through collaborating objects [Reenskaug and Coplien, 2009]. In turn, roles are a natural concept capturing the behavior of participants in a collaboration. Moreover, roles permit the specification of interactions independent from the interacting objects. Similarly, more recent approaches use roles to capture context-dependent properties of objects. The notion of roles can help to tame the increased complexity and context-dependence. Despite all that, these years of research had almost no influence on current software development practice. To make things worse, until now there is no common understanding of roles in the research community and no approach fully incorporates both the context-dependent and the relational nature of roles [Kühn et al., 2014]. In this thesis, I will devise a formal model for a *family of role-based modeling languages* to capture the various notions of roles [Kühn et al., 2015a]. Together with a software product line of *Role Modeling Editors*, this, in turn, enables the generation of a *role-based language family* for Role-based Software Infrastructures (RoSI).

ACKNOWLEDGEMENTS

As any PhD students, I too stood on the shoulders of giants. Henceforth, I highlight some of them. First of all, I thank my family, especially, my wife Josephine as well as my children Mira, Vincent and Milan, who supported me and endured my repeated absence. Additionally, I thank my parents Andrea and Peter, which allowed me to become the man I always wanted. Secondly, I thank Sebastian Richly, who I consider my mentor, and Sebastian Götz. Both introduced me to the intricacies of roles and role-based software development. My discussions with them significantly shaped my understanding of roles in software systems. On the same level, I want to thank both my supervisors Uwe Aßmann and Wolfgang Lehner, who guided my throughout my thesis, gave me focused goals, and where always available when I had questions, even at 4 am in the morning. Similarly, I thank Walter Cazzola, who hosted me during my research stay in Italy and opened up my perspective towards the development of Language Product Lines. Besides them, I am also grateful for Colin Atkinson's support, as he quickly and throughly reviewed my written thesis and provided interesting suggestions for future research. In a broader sense, I consider both Friedrich Steimann and Trygve Reenskaug paragons of thoroughness and curiosity in the research field on roles. Third, I want to thank all the other RoSI students with whom I had great discussions on the notion and usefulness of roles. Among others, I want to highlight Tobias Jäkel, Stephan Böhme, Steffen Huber, Max Leuthäuser, and İsmail İlkan Ceylan. Moreover, I am grateful for all the students I mentored. Because, despite the general assumption, I learned a lot from them. In particular, I want to thank Kay Bierzynski, Duc Dung Dam, Christian Deussen, David Gollasch, Marc Kandler, Kevin Ivo Kassin, and Paul Peschel, who contributed to the development of FRaMED. Fourth, I owe the German Research Foundation (DFG) the funding of this thesis as well as a great experience within the research training group (RTG: 1907) on Role-Based Software Infrastructures for continuous-context-sensitive Systems (RoSI). Last but not least, I need to thank Eduard Graf and Andre Lange from the zickzack GmbH as well as all members of the Café ASCII, because the former produced and the latter distributed the beverage that fueled my writing, i.e., I found a strong correlation between my beverage consumption and text output.

Thomas Kühn
3.3.2017

CONTENTS

LIST OF PUBLICATIONS

Primary publications featured in this thesis:

1. **Thomas Kühn**, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A metamodel family for role-based modeling and programming languages. In *Software Language Engineering, volume 8706 of Lecture Notes in Computer Science*, pages 141–160. Springer, 2014.
2. **Thomas Kühn**, Böhme Stephan, Sebastian Götz, and Uwe Aßmann. A Combined Formal Model for Relational Context-Dependent Roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–124. ACM, 2015b.
3. **Thomas Kühn**, Böhme Stephan, Sebastian Götz, and Uwe Aßmann.A Combined Formal Model for Relational Context-Dependent Roles (Extended). Technical Report TUD-FI15-04-Sept-2015, Technische Universität Dresden, 2015.
4. **Thomas Kühn**, Walter Cazzola, and Diego Mathias Olivares. Choosy and picky: Configuration of language product lines. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15)*, 2015.
5. **Thomas Kühn** and Walter Cazzola. Apples and oranges: Comparing Top-Down and Bottom-Up Language Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC'16)*, pages 50–59, ACM, 2016.
6. **Thomas Kühn**, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. Framed: Full-fledge role modeling editor (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016*, pages 132–136, ACM, 2016.

Coauthored publications in related fields of research:

7. Tobias Jäkel, **Thomas Kühn**, Hannes Voigt, and Wolfgang Lehner. Rsql - A Query Language for Dynamic Data Types. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 185–194. ACM, 2014.
8. Tobias Jäkel, **Thomas Kühn**, Stefan Hinkel, Hannes Voigt, and Wolfgang Lehner. Relationships for Dynamic Data Types in RSQL. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2015.
9. Tobias Jäkel, **Thomas Kühn**, Hannes Voigt, and Wolfgang Lehner. Towards a Contextual Database. In *20th East-European Conference on Advances in Databases and Information Systems*, 2016.

Other coauthored publications:

10. Christian Piechnick, Georg Püschel, Sebastian Götz, **Thomas Kühn**, Ronny Kaiser, and Uwe Aßmann. Towards context modeling in space and time. In *MORSE 2014, The 1st International Workshop on Model-Driven Robot Software Engineering*, pages 39–50. CEUR Workshop Proceedings, 2014a.
11. Christian Piechnick, Sebastian Richly, **Thomas Kühn**, Sebastian Götz, Georg Püschel, and Uwe Aßmann. Contextpoint: An Architecture for Extrinsic Meta-Adaptation in Smart Environments. In *ADAPTIVE 2014, The Sixth Interna- tional Conference on Adaptive and Self-Adaptive Systems and Applications*, pages 121–128, 2014b.
12. Sebastian Götz, **Thomas Kühn**, Christian Piechnick, Georg Püschel, and Uwe Aßmann. A models@run.time Approach for Multi-Objective Self-Optimizing Software. In *Adaptive and Intelligent Systems*, pages 100–109. Springer, 2014.

PART I

REVIEW OF CONTEMPORARY ROLE-BASED LANGUAGES

"Modeling is one of the most fundamental processes of the human mind."
— Rothenberg et al. [1989]

1 INTRODUCTION

In other words, Jeff Rothenberg reminds us that, modeling is the basic ability of abstracting aspects of reality to better comprehend and reason about certain aspects of reality avoiding its *complexity*, *danger* and *irreversibility* [Rothenberg et al., 1989]. In detail, he characterized *modeling* as activity *"to represent a particular referent cost-effectively for a particular cognitive purpose"*. This is particularly true for *conceptual modeling*, which is *"the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication."* [Mylopoulos, 1992]. By extension, a conceptual model is a formal description of parts of a subject domain by means of concepts and their interrelations. In contrast to other models, such as street maps or floor plans, conceptual models are not only required to be understood by humans, but also by computers. Thus, this enables their use not only for communication and problem-solving, but also for formal validation and artifact generation. Although most would assume that classical conceptual modeling languages, such as the Entity-Relationship Model (ER) [Chen, 1976] or the Unified Modeling Language (UML) [Rumbaugh et al., 1999], are appropriate conceptual modeling languages, several researchers, e.g., [Steimann, 2000c, Atkinson and Kühne, 2002, Guizzardi et al., 2004, Liu and Hu, 2009a], have pointed out their deficiencies when used to model more complex, context-dependent, and dynamic domains. This, in turn, makes these conceptual modeling languages inappropriate for such domains, and – in the words of Jeff Rothenberg – their use "can do considerable harm" [Rothenberg et al., 1989].

This work contributes to the field of conceptual modeling by investigating role-based modeling languages (RMLs) as promising approach to more appropriately model nowadays complex, context-dependent and dynamic domains.

1.1 BACKGROUND

The history of software systems is characterized by an increasing complexity, level of uncertainty, and rate of change. From the early mainframe applications with limited inputs and outputs, over desktop applications for personal computers up to distributed applications enabled by the ARPANET and the later Internet. At each stage, application developers had to face increased complexity, uncertainty, and rate of change of the application domain and execution environment. This trend continues to today's mobile, pervasive, and very large software systems that should be aware of their execution context and be able to adapt themselves to cope with (un)anticipated situations. Moreover, such systems require a continuous development process able to keep up with hourly changes to the application domains and execution environments to incorporate new business op-

portunities or patch implementation errors. In the near future, more and more applications will be such *context-sensitive* distributed software systems incorporating mobile and pervasive devices that are *continuously* developed. This further challenges application developers and requires them to adopt continuous development methodologies; more flexible, adaptive runtime environments; and finally domain modeling languages able to capture context-dependent and collaborative behavior of objects. Accordingly, to lay the foundation for the development of these future systems the *Research Training School* on *Role-based Software Infrastructures for continuous-context-sensitive Systems (RoSI)* investigates whether the concept of roles can be feasibly employed to approach the aforementioned challenges. In particular, this thesis studies roles as a promising addition to object-oriented modeling languages able to model context-dependent, collaborative aspects of an application domain. In sum, this thesis establishes the foundations of RML and provides the necessary tools to feasibly use roles for conceptual modeling.

1.2 MOTIVATION

Both role-based modeling languages (RMLs) and role-based programming languages (RPLs) have been investigated for several decades.[1] The first account for the application of roles to modeling dates back to 1977, when Bachman and Daya proposed the *Role Data Model* [Bachman et al., 1977]. They facilitate roles as *"a defined behavior pattern which may be assumed by entities of different kind"* [Bachman et al., 1977, p.45] to handle different entity types playing the same role type coherently [Bachman et al., 1977]. Till the year 2000, the term *"role"* has occurred in multiple areas within computer science, e.g., access control,[2] knowledge representation, conceptual modeling, data modeling, as well as object-oriented design and implementation [Steimann, 2000a]. However, after Steimann surveyed the contemporary literature on roles [Steimann, 2000b], he correctly observes that by 2000 *"the influence of the role data model on modelling has at best been modest"* [Steimann, 2000b, p.85]. While Steimann is right when he claims that roles as behavioral pattern were not adopted in modeling, most modeling languages feature roles as named places at the end of relationships, e.g., ER [Chen, 1976] and UML [Rumbaugh et al., 1999]. Thus, while roles are well-established in modeling languages their semantics differ and their full potential is rarely utilized. Since Steimann's survey, more elaborate applications of the role concept have been proposed ranging from knowledge representation [Loebe, 2005, Guarino and Welty, 2009], via data modeling [Halpin, 2005, Liu and Hu, 2009a, Jäkel et al., 2015], and conceptual modeling [Guizzardi and Wagner, 2012, Hennicker and Klarl, 2014] through to object-oriented design and implementation [Baldoni et al., 2006c, Herrmann, 2005, Balzer et al., 2007]. Although these approaches employ roles to model, reason, and implement context-dependent behavior of objects to cope with the requirements of mobile and pervasive applications [Herrmann, 2005, Liu and Hu, 2009a], their proposals had almost no impact on mainstream modeling and programming languages. Reenskaug and Coplien perfectly put this into perspective, when they emphasize that *"Object-oriented programming languages traditionally afford no way to capture collaborations between objects"* [Reenskaug and Coplien, 2009] and that *"roles [could] capture collections of behaviors that are about what objects do"* [Reenskaug and Coplien, 2009]. In other words, the past years of research on roles had next to no influence on current software development practice, in spite of clear evidence that roles can tame the increased complexity and context-dependence of current context-adaptive, distributed software systems.

[1] By the 8th of October 2017, it will be exactly 40 years.

[2] Throughout this thesis we consider Role-Based Access Control (RBAC) [Ferraiolo et al., 1995] as a special application for roles with a rather narrow scope.

1.3 PROBLEM DEFINITION

From the previous discussion, one could argue that the introduction of roles failed due to the insufficiency of the role concept. While it is true that no big case study has shown the sufficiency of role-based modeling and programming, it does not necessarily follow that the lack of adoption in practice is a result of its insufficiency. In my opinion, there are four basic reasons why roles have not been adopted by more researchers and practitioners. First, until now there is no common understanding of roles in the research community [Steimann, 2000b, Kühn et al., 2014]. Instead, each approach focuses on certain features attributed to roles. Second, the research field itself suffers from *discontinuity* and *fragmentation* of the various role definitions [Kühn et al., 2014]. In particular, most approaches *did not* take previous results into account and *did not* continuously improve role-based modeling. Third, there are only few approaches, e.g., [Genovese, 2007, Zhu and Zhou, 2006, Hennicker and Klarl, 2014], that provide a formal foundation for their RML incorporating most of the features of roles [Steimann, 2000b, Kühn et al., 2014]. Last but not least, most role-based modeling and programming languages are not readily applicable, because of their complexity, level of abstraction, and/or missing tool support. These issues not only hinder researchers improving previous approaches, but also software practitioners exploring new modeling and programming languages. The first two issues can be tackled by developing a family of role-based modeling languages by means of the features of roles. However, the third and fourth issue must be addressed by providing a comprehensive formal foundation for roles, a role-based modeling language incorporating most features of roles, and, finally, readily applicable tools that support its use for the design of future role-based software systems.

To achieve these goals, this thesis makes the following contributions to the research areas on role-based modeling and programming languages:

1. It extends the initial list of features of roles by adding 12 new features retrieved from contemporary approaches, discussed in Chapter 2.
2. Based on these features a Systematic Literature Review (SLR) [Kitchenham, 2004] was conducted to survey the contemporary literature on RMLs (Chapter 4) and RPLs (Chapter 5).
3. Next, it establishes the foundations of the RMLs (Chapter 7) by introducing the *Compartment Role Object Model (CROM)*, a framework for conceptual modeling that incorporates most of the features of roles into a coherent, graphical modeling language and provides a set-based formalization of roles.
4. Additionally, a corresponding graphical modeling editor, called *Full-fledged Role Modeling Editor (FRaMED)*, is presented that allows the creation, manipulation and provisioning of *CROM* models.
5. To approach the *discontinuity* and *fragmentation* of the research area on roles, this thesis proposes a metamodeling approach for the creation of a family of RMLs (Chapter 8).

In summary, these contributions propose *CROM* as new formal RML together with *FRaMED* as readily usable graphical modeling editor. This, in turn, permits conceptual modeling practitioners to apply role-based modeling to deal with nowadays complex, context-dependent, and dynamic domains. In addition, this thesis facilitate a family of metamodels and corresponding family of modeling languages that enables researchers to develop language variants for their individual approach. The latter, more importantly, harmonizes the various notions of roles by structuring them with respect to the features of roles, to permit researchers to develop individual yet composable role-based languages.

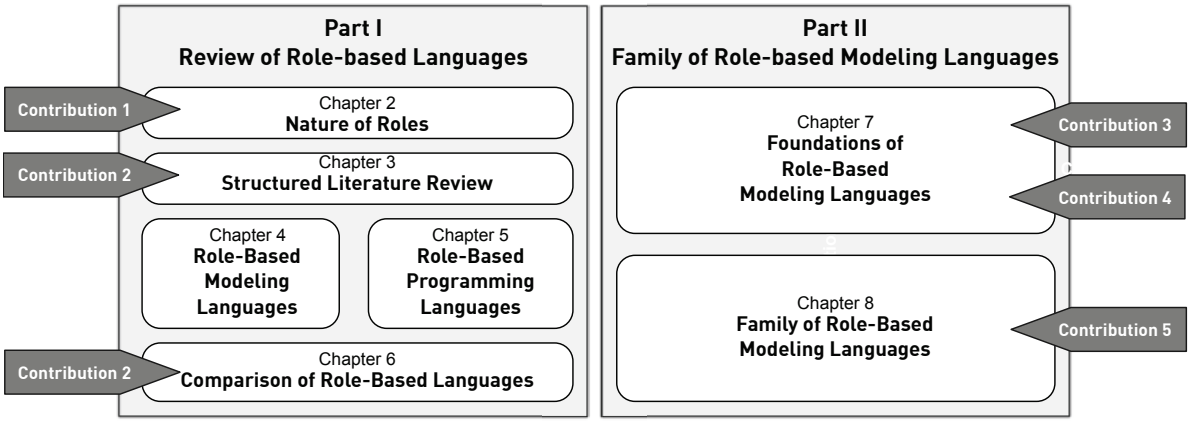

Figure 1.1: Overview of this Dissertation.

1.4 OUTLINE

This thesis can be seen as a response to the excellent work of Friedrich Steimann, who studied the notion of roles in 2000 and proposed a unifying model for roles [Steimann, 2000b,a]. Accordingly, this thesis is divided into two parts, as shown in Figure 2.1.

The **first part** establishes the notion of roles by surveying the state of the art and comparing the various contemporary RMLs and RPLs. Hence, Chapter 2 discusses the different notions found in the literature and explains the features typically attributed to roles. These features of roles form the prerequisite for the conducted SLR, outlined in Chapter 3 highlighting the methodology and selection process. Afterwards, Chapter 4 and Chapter 5 describe and evaluate the identified contemporary role-based modeling and programming languages, respectively. The first part is concluded by Chapter 6 comparing the role-based languages and discussing the result of the comparison.

The **second part**, in turn, introduces an approach to reconcile the different role-based languages, namely the development of language families. First, Chapter 7 provides the foundations for RMLs by providing an ontologically founded formal definition of the graphical modeling language CROM. Build on this foundation, Chapter 8 describes how the family of RMLs was created highlighting the established family of metamodels and the developed family of modeling editors. Finally, Chapter 9 concludes this thesis by summarizing its contributions, highlighting related approaches to family engineering, and discussing prospects for future research. In conclusion, the first part highlights the shortcomings in contemporary role-based research and the second part proposes a family of role-based languages as solution to these issues.

"All the world's a stage,
And all the men and women merely players;
They have their exits and their entrances,
And one man in his time plays many parts,
His acts being seven ages."

— Shakespeare [1763]

2 NATURE OF ROLES

The notion of roles is very old. For linguists, the first definition of the notion dates way back to Lodwick in 1647, who describes *appelative nouns*, as *"a name by which a thing is named and distinguished, but not continually, only for the present, in **relation** to some action done or suffered"* [Lodwick, 1647, p. 7–8]. Steimann [2000b] correctly infers that *appelative nouns* denote *roles*, which are in **relation** by means of an action, e.g., the murderer and murder victim related by the murder [Lodwick, 1647, p. 7–8]. Later in 1763, Shakespeare's famous quote, shown above, also captures another core aspect of roles. It emphasizes that persons play multiple *roles* on various **stages**, simultaneously. On a closer look, he also implies that persons **change their behavior**, when they assume and abandon *roles* by entering or exiting a **stage**.[1] While there are many more definitions for roles throughout history,[2] these two suffice to convey the three fundamental natures of roles. In short, the **behavioral nature** emphasizes the role's ability to change the player's behavior, the **relational nature** highlights that roles are usually related to other roles, and finally the **context-dependent nature** captures that roles are defined within a certain action, stage, or more generally context. These three natures provide a simple yet sufficient classification scheme, as each nature focuses on an orthogonal aspect. Here many readers would probably object that this classification is to narrow. This is certainly true, if it were the only classification scheme employed. However, this classification is accompanied by a list of 27 features that have been identified in role-based modeling and programming languages. Thus, while the former is used to group similar approaches throughout the thesis, the latter is employed for the detailed classification of each language.

This chapter is structured, accordingly. First, Section 2.1 introduces a small real world example that is used as running example throughout this thesis. Afterwards, Section 2.2, 2.3, and 2.4 individually elucidate the behavioral, relational, and context-dependent nature of roles, respectively. After introducing the natures of roles, Section 2.5 reviews the various constraints found in role-based languages. Finally, Section 2.6 extends the list of features of roles introduced in [Steimann, 2000b] to incorporate features found in the contemporary literature. Notably though, the natures of roles have been published in [Kühn et al., 2015a] and the extended classification scheme in [Kühn et al., 2014].

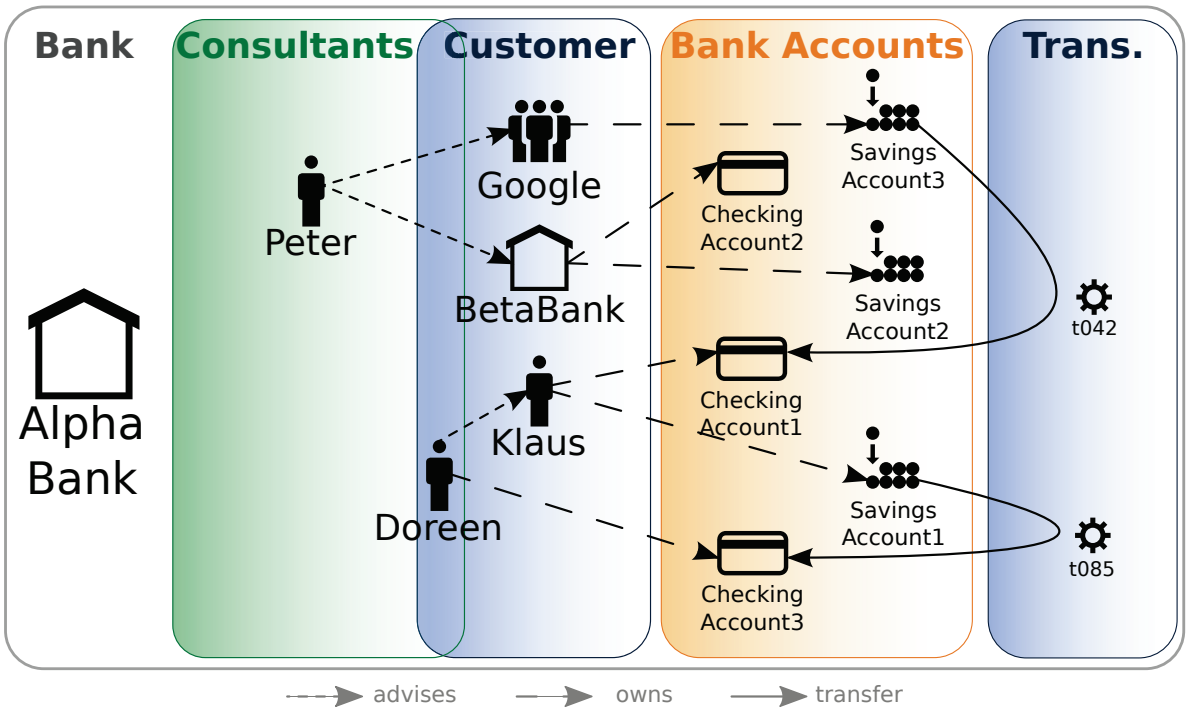

Figure 2.1: Scenario of an exemplary financial institution.

2.1 RUNNING EXAMPLE

Before further exploring the natures of roles, it is best to consider a concrete yet small scenario to provide vivid examples. Accordingly, lets assume you are tasked to model and implement a small banking application able to capture the scenario depicted in Figure 2.1. In detail, the scenario encompasses the *AlphaBank* containing both *Peter* and *Doreen* as consultants, serving *Google*, *BetaBank*, *Klaus*, and *Doreen* as customers, and managing their various checking and savings accounts. Following this scenario, a bank is described as a financial institution that employs consultants, serves customers, manages their accounts, and performs money transferals. Furthermore, the scenario indicates that consultants are persons who advise at least one customer. Customers, in turn, can be either persons, companies or banks, can own several checking and savings accounts, and can perform transactions. Accounts have a unique id and can be either savings or checking accounts. Transactions are managed by the bank and represent the process of transferring money from one source to one target account. In addition to these domain concepts, the domain model must also include the following four domain constraints, as they are required by financial regulations. First, consultants are prohibited to advise themselves as a customer. Second, checking accounts must have exactly one owner whereas savings accounts can have multiple owners. Third, it is forbidden to transfer money from one account back to itself within one transaction. Last but not least, each account referenced in a transaction must be a valid account in a bank. In sum, this scenario is small enough to serve as a comprehensive running example, however, it still imposes several design challenges for application designers. These challenges will be addressed in the following sections, to motivate the different natures of roles.

[1] Obviously, not only persons but every thing can play roles.
[2] See [Steimann, 2000a, Sect. 1.2], for a thorough investigation on the origins of the role concept.

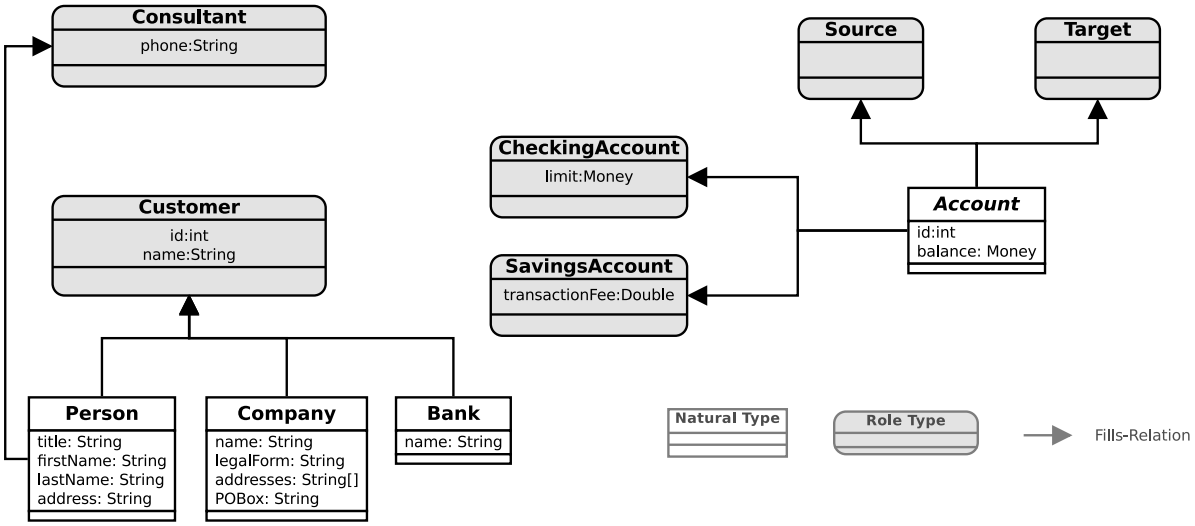

Figure 2.2: Example model highlighting the behavioral nature of roles.

2.2 BEHAVIORAL NATURE

The behavioral nature emphasizes that roles are able to change the behavior of their players. Following Shakespeare's line of thought, a person's or similarly an object's behavior is affected by the roles it currently plays. This entails that a role is able to adapt the behavior of its player [Steimann, 2000b]. In addition to this aspect, the behavioral nature of roles is further characterized by the following key features. First, while a role (instance) has exactly one player, an object can play multiple roles simultaneously [Bachman et al., 1977, Steimann, 2000b]. For instance, the person *Doreen* (Figure 2.1) can be both a customer and a consultant at the same time. Second, unrelated objects can play the same role (type) [Bachman et al., 1977, Steimann, 2000b]. Again, the customer role can be played by persons, companies, and banks alike without requiring them to have a common super type. Thus, the bank can handle all customers equally regardless of their actual player type. Third, objects can assume and abandon roles dynamically and thus change their state and behavior dynamically [Steimann, 2000b]. Considering our bank application, a person can dynamically assume the customer role in the bank application. Later on, this person can also become a consultant, as well as a customer in another bank. Last but not least, an object might play the same role (type) multiple times [Steimann, 2000b]. This becomes clear when considering the many times an account can be the source or target of a money transaction during its lifetime. In its core, the behavioral nature highlights that multiple roles can be played by unrelated objects and that all assumed roles of an object affect its state and behavior.

In modeling and programming languages, the behavioral nature is usually represented by *role types* as addition to *natural types* [Sowa, 1984, Steimann, 2000b], as well as the *fills* relation between *natural types* and *role types* on the type level. Granted that there are various other names for this relation, e.g., *role-filler* [Steimann, 2000b], *role-of* [Loebe, 2005] or *played-by* [Baldoni et al., 2006c, Herrmann, 2005], yet most authors agree that it denotes a many-to-many relation specifying that instances of a certain natural type can play roles of a certain role type. Consider the bank scenario, in which the entities `Person`, `Company`, and `Bank` could be modeled as natural type individually filling the `Customer` role type. Additionally, the `Person` natural type could be defined to also fill the `Consultant` role type. Thus, all persons (i.e., instances of the natural type `Person`) can play instances of the `Customer` role type as well as instances of the `Consultant` role type. Similarly, the

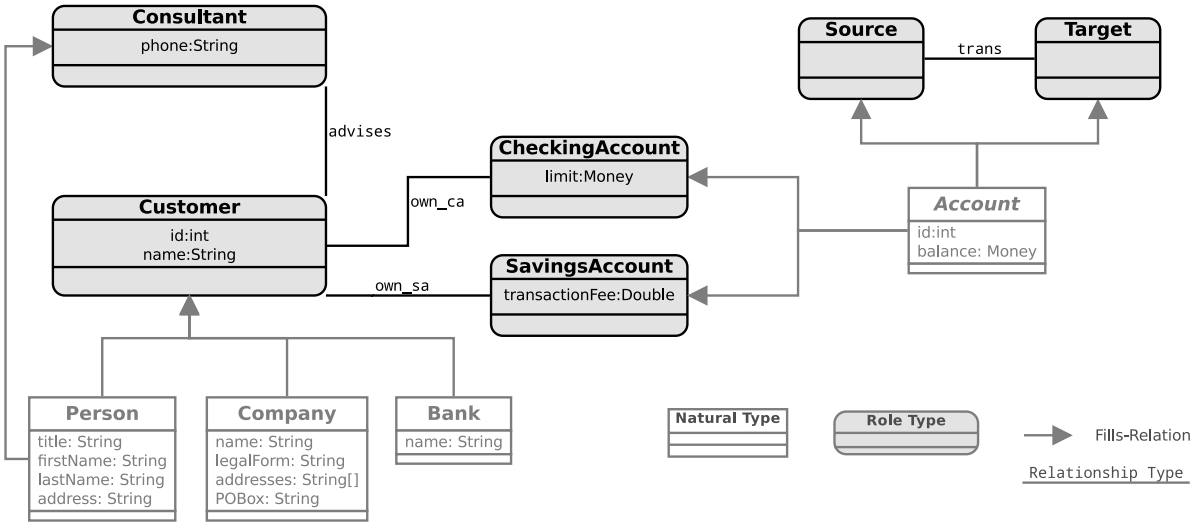

Figure 2.3: Example model highlighting the relational nature of roles.

entity `Account` can be modeled as natural type, whereas `CheckingAccounts`, `SavingsAccounts`, `Source` and `Target` are role types the `Account` can fill. In sum, Figure 2.2 depicts an informal model of the banking application incorporating the behavioral nature. For brevity, method definitions will be omitted henceforth. Notably, this model utilizes roles to express the dynamics of customers, consultants, and bank accounts of the banking domain. While it is true that this domain could also be modeled with classical object-oriented modeling and programming languages, the resulting domain model would require exponentially many types with redundant implementations, to model all combinations of natural types playing roles types by means of specialization and generalization [Steimann, 2000b]. Moreover, such a design would entail that objects must be reinstantiated whenever they assume or abandon a role. In sum, such a model would fail to capture the dynamic aspect of the banking domain.

2.3 RELATIONAL NATURE

In contrast to the behavioral nature, the relational nature can be found in most current modeling languages. Consider the modeling languages ER [Chen, 1976] and UML [Rumbaugh et al., 1999] where roles denote the named ends of relationships and associations, respectively. Chen [1976] himself writes, "[the] role of an entity in a relationship is the function that it performs in the relationship" [Chen, 1976]. In other words, Chen believes that roles are functionally dependent on the relationship they are participating. However, as Steimann argues, roles as named places "[fail] to account for the fact that roles come with their own properties and behaviour" [Steimann, 2000b, p. 88]. As a result, the relational nature requires that both roles and relationships are first-class citizens, that roles can depend on relationships [Steimann, 2000b], and that roles have their own properties [Steimann, 2000b]. Admittedly, this definition excludes the aforementioned classical modeling languages, as well as some programming languages with first-class relationships, e.g. *RelJ* [Bierman and Wren, 2005], *Relationship Aspects* [Pearce and Noble, 2006]. However, these languages do not treat roles as types, and thus cannot provide insight into the relation between roles and relationships. Nevertheless, multiple role-based modeling and programming languages embrace the relational nature of roles, e.g. [Bachman et al., 1977, Steimann, 2000b, Halpin, 2005, Balzer et al., 2007, Nelson et al., 2008, Jäkel et al., 2015]. In particular, those role-based languages typically de-

fine role types and their filling natural types within the definition of relationship types. On the instance level, however, links connect the related role instances. In addition, [Balzer et al., 2007, Nelson et al., 2008] collect all links of the same relationship type into a singleton instance, to permit the representation of relationships as holistic entity on the instance level. Consequently, these holistic relationship instances can play roles in other relationships, as well [Balzer and Gross, 2011]. As an illustration of the relational nature of roles, Figure 2.3 extends the behavioral model (Figure 2.2) with relationships between role types. In detail, the banking domain model is enriched by four relationships showcased in the banking scenario (Figure 2.1). First, the `advises` relationship type specifies the relation between *consultants* and *customers* they advise, basically, establishing that some customers are advised by a consultant. Next, the fact that *customers* own *savings* and *checking accounts* is modeled by two different relationship types, namely `own_sa` and `own_ca`, respectively. While this separation appears to be superfluous, it actually accounts for the different cardinalities and different semantics of the two relationships. As a reminder, *saving accounts* can be owned by multiple customers whereas *checking accounts* by only one customer. Last but not least, the process of transferring money between two accounts is also modeled as the relationship type `trans` between the *source* and *target* role of account. Notably, all those relationships can be further constraint by providing cardinalities. These and the various other modeling constraints will be discussed in Chapter 2.5. In conclusion, the relational nature of roles underline the strong connection between roles and relationships.

As can be seen from the domain model in Figure 2.3, languages solely focusing on the relational nature of roles assume that all roles and relationships are equally relevant to an object's properties. In other words, there is no notion of context or scope on which both roles and relationships depend on. This becomes apparent, if you reconsider that a money transaction can only be represented as the `trans` relationship. Consequently, the transaction cannot be tied to the owning bank, and, moreover, it is impossible to reuse the modeled money transaction, as it is tied to the modeled scenario. To resolve this dilemma, it must be understood that roles can be context-dependent, as well as relationships themselves.

2.4 CONTEXT-DEPENDENT NATURE

In the past ten years, computer scientists shifted their focus towards context-aware applications. According to Piechnick et al. [2012], *"[due] to the wide acceptance and distribution of mobile devices, it has become increasingly important that an application is able to adapt to a changing environment"* [Piechnick et al., 2012, p. 93]. In other words, Piechnick argues that distributed mobile applications must be able to adapt themselves to an ever-changing context. As a result, more recent role-based modeling and programming languages, e.g. [Genovese, 2007, Herrmann, 2005, Baldoni et al., 2006c, Liu and Hu, 2009a, Hennicker and Klarl, 2014], have employed context-dependent roles to simplify the design and implementation of context-aware applications. In general, all these approaches have introduced new concepts to encapsulate those roles relevant to a certain context, situation, interaction, or collaboration. However, different researchers have used different terms to denote the semantic context of roles, e.g., *stage* [Shakespeare, 1763], *organization* [Da Silva et al., 2003], *institution* [Baldoni et al., 2006c, Genovese, 2007], *team* [Herrmann, 2005], *relations* [Nelson et al., 2008, Harkes and Visser, 2014], *ensemble* [Hennicker and Klarl, 2014], and *context* [Kamina and Tamai, 2010]. Although one might argue that *context* is the most appropriate term to use when denoting the entities roles should depend on, I would disagree. Not only is the term itself massively overloaded in computer science, but there exists a dichotomy between *context* in computer science and *context* in role-based modeling and programming. On the one hand, consider one of

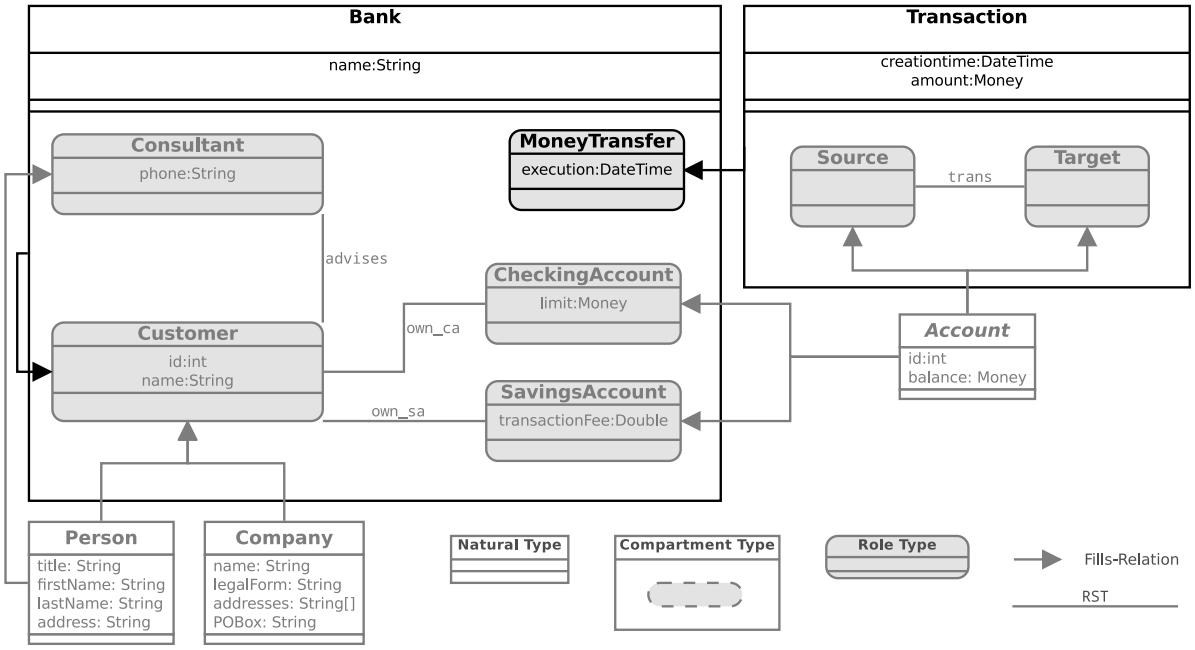

Figure 2.4: Example model highlighting the context-dependent nature of roles.

the most cited definitions of *context* by Dey stating that *"Context is any information that can be used to characterise the situation of an entity"* [Dey, 2001, p.5]. In essence, Dey classifies *context* as any observable information (i.e. inferable from sensor data) relevant to classify a situation. Consequently, *context* should have no identity, no existential parts, and an indefinite lifespan. On the other hand, Kamina and Tamai, for instance, assume that *"Each context consists of a set of roles that represents collaborations performed in that context."* [Kamina and Tamai, 2010, p.15]. Furthermore, they describe *context* to have an identity, require roles as their parts, and have a defined lifespan denoted by its activation scope [Kamina and Tamai, 2010]. Apparently, each quote referred to a fundamentally different conceptual entities. To resolve this dichotomy, I propose to use the term *compartment* to refer to the latter.[3] *Compartments* are defined as an *"objectified collaboration with a limited number of participating roles and a fixed scope"* [Kühn et al., 2014, p.146]. In particular, *compartments* have their own identity, properties, behaviors [Genovese, 2007, Herrmann, 2005, Baldoni et al., 2006c, Hennicker and Klarl, 2014], and might play roles like objects [Herrmann, 2005, Baldoni et al., 2006c]. Similar to classes and objects, researchers distinguish between compartment types and compartments as their instances [Genovese, 2007, Herrmann, 2005, Baldoni et al., 2006c, Hennicker and Klarl, 2014]. Additionally, compartments represent the definitional boundary and scope for the participating roles, i.e., they limit the object's visibility and accessibility within a compartment instance to those who play a role in this compartment [Herrmann, 2013]. Furthermore, *compartments* are a generalization of the other notions of context found in role-based languages, e.g., *process*, *situation*, *institution*, *organization*, *group*, *ensemble*, *relation*, and *collaboration*. Hence, compartments can not only enclose and specify context-dependent behavior, but more generally any collaboration-dependent behavior. In the running example, both *transactions* and the *banks* can be represented as compartments. The `Transaction` is modeled as a compartment type with the two participating role types: `Source` and `Target`. This compartment represents

[3] Other researchers suggested to use *semantical context* or *extrinsic context* to denote compartments. However, their semantics is misleading, as it either focuses on semantics or identity.

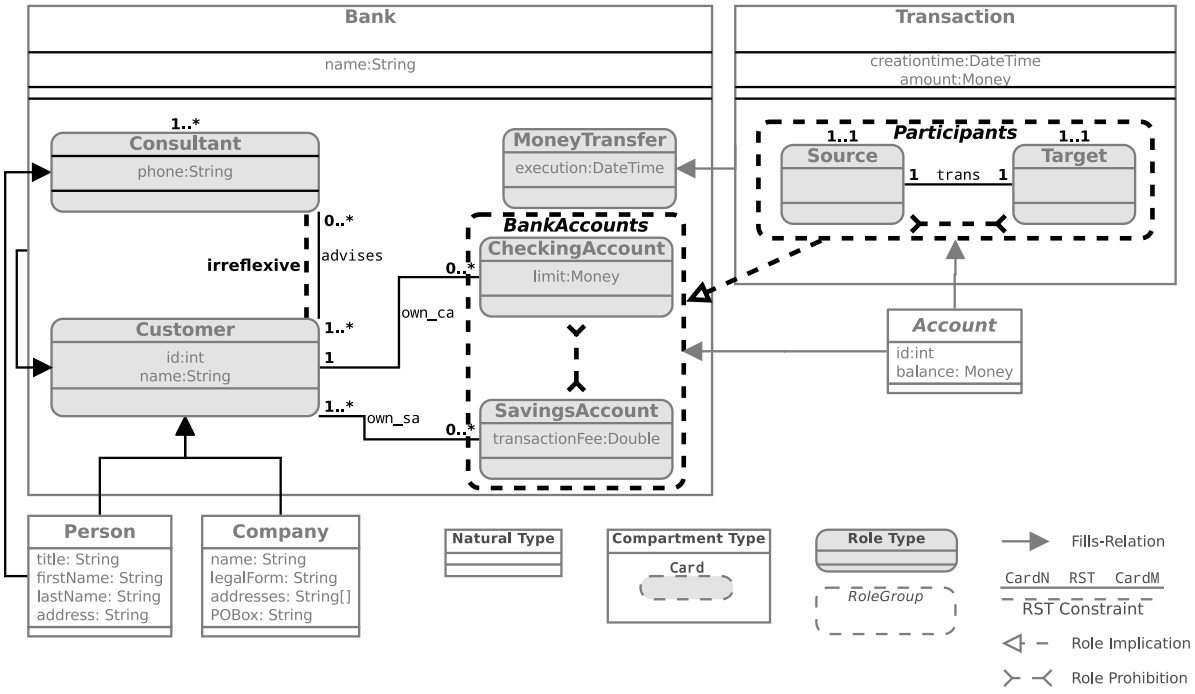

Figure 2.5: Example model highlighting various modeling constraints.

the process of transferring money from one account to another. Thus, each transaction is represented in an individual compartment instance, with individual state. Consequently, an account can play the `Source` role in multiple compartment instance simultaneously. Similarly, the `Bank` is modeled as a compartment type enclosing all role types relevant to the financial institution. As a result, each instance of a `Bank` represents an individual financial institution with its own customers, consultants, bank accounts and transactions it manages. Moreover, the `Bank` compartment is also defined to fill the `Customer` role permitting banks to become customers of other banks. Notably, to model that *banks* manage their transactions, the `MoneyTransfer` role type is introduced. Accordingly, this role is assumed by each transaction that is issued by a customer of a particular bank. Conversely, a `Bank` collects and executes its money transferals by means of the `MoneyTransfer` role that is played by individual `Transaction` compartment instances.[4] The resulting model of the banking application is depicted in Figure 2.4. In conclusion, the context-dependent nature of roles enables the representation of the individual collaboration, process or compartment a set of roles depends on. In short, it captures the existential dependency between roles and compartments.

2.5 CONSTRAINTS IN ROLE-BASED LANGUAGES

Although one would assume that the appropriateness of a modeling language is determined by the number of concepts that can be captured by the model. Anyone familiar with conceptual modeling should see that an appropriate domain model must not only include the domain concepts and relationships, but also the domain constraints. Consider, the banking domain modeled so far. While it correctly reflects the dynamics of the domain, it does not capture the imposed financial regulations. Hence, this section recollects the various kinds of constraints found in contemporary RMLs.

[4] This design has the additional benefit to allow for adding new kinds of money transactions easily, for instance transactions to *BitCoin*.

Similar to classical modeling languages, like ER [Chen, 1976] and UML [Rumbaugh et al., 1999], most RMLs and few RPLs include relationship constraints. These can be further divided into three classes of constraints. First, *cardinality constraints* [Chen, 1976, Rumbaugh et al., 1999, Balzer et al., 2007, Guizzardi and Wagner, 2012, Jäkel et al., 2015] limit the number of entities that can be related by a relationship. In the banking example all relationships can be further restricted by adding *cardinalities* to their ends. The `own_ca` relationship type between *customers* and *checking accounts*, for instance, should be constrained with the cardinality *one-to-many*, in order to ensure that each *checking account* is owned by exactly one *customer*. Similarly, the other relationships in the banking domain can be augmented with cardinalities, as depicted in Figure 2.5. In contrast to cardinalities, *intra-relationship constraints* [Halpin, 2005, Balzer et al., 2007, Guizzardi and Wagner, 2012] impose a certain structure over all links of a relationship. In particular, most of the constraints, in [Halpin, 2005, Balzer et al., 2007, Guizzardi, 2005], correspond to mathematical properties of binary relations, such as *reflexivity*, *acyclicity*, and *total order*. These can, for instance, be used to declare that the `advises` relationship is an *irreflexive* relation, depicted in Figure 2.5 denoting that consultants cannot advise themselves as customers. In detail, it prohibits that a link in the `advises` relationship relates the same player objects. Notably, the *parthood constraints* introduced in [Guizzardi, 2005] can be seen as special kinds of *intra-relationship constraints*. While the previous constraints are imposed on one relationship, *inter-relationship constraints* are established between two relationships, such as *relationship implication* [Halpin, 2005, Bierman and Wren, 2005] and *relationship exclusion* [Halpin, 2005]. In particular, the former can be used to enforce that one relationship is a subset of another relationship. As an example, let us assume we want to specialize the `advises` relationship to *serve* premium customers. Consequently, the `serve` relationship would be specified to imply the `advises` relationship guaranteeing that all served premium customers of a consultant are also captured as advised customers. So far, these constraints refer to relationships rather than roles. Nonetheless, three kinds of constraints for roles have been proposed in the literature. First, *role constraints* restrict the roles that one object can simultaneously play. Introduced by Riehle and Gross [1998], they include binary constraints between roles to either prohibit or require that both roles are played at the same time [Riehle and Gross, 1998]. Specifically, the *role prohibition* between the `CheckingAccount` and `SavingsAccount`, showcased in Figure 2.5, prohibits that an account can play both roles at the same time. Second, [Zhu and Zhou, 2006, Kühn et al., 2014] suggested employing notions to group roles and constrain them together. Accordingly, the banking model (Figure 2.5) specifies the two *role groups* `Participants` and `BankAccounts` to collect the role types filled by accounts in the `Bank` compartment type and `Transaction` compartment type, respectively. As a result, to ensure that all transaction only involve valid bank accounts, the model designer could simply specify a *role implication* from the `Participants` to the `BankAccounts` *role groups*. In sum, both *role groups* and *role constraints* can only restrict the roles an object is permitted to play simultaneously. However, to constrain the number of roles that can exist within a compartment, Zhu and Zhou [2006], as well as Hennicker and Klarl [2014] have introduced *occurrence constraints*. In detail, they impose a lower and upper bound (or short *cardinality*) on the number of instances of a given role type that can exist concurrently within one compartment instance. In the banking application, both the `Source` and `Target` role type have an *occurrence constraint* of `1..1`, Figure 2.5. This indicate that each transaction requires exactly one source and one target account throughout its lifetime. Accordingly, while *role constraints* focus on one individual object, *occurrence constraints* restrict roles contained in individual compartments. While the presented examples indicate the importance of these modeling constraints to appropriately model the banking domain, none of the contemporary RMLs and RPLs have combined all of them into one coherent modeling or programming language.

Table 2.1: Steimann's 15 classifying features, extracted from [Steimann, 2000b].

1.	Roles have properties and behaviors	*(M1, M0)*
2.	Roles depend on relationships	*(M1, M0)*
3.	Objects may play different roles simultaneously	*(M1, M0)*
4.	Objects may play the same role (type) several times	*(M0)*
5.	Objects may acquire and abandon roles dynamically	*(M0)*
6.	The sequence of role acquisition and removal may be restricted	*(M1, M0)*
7.	Unrelated objects can play the same role	*(M1)*
8.	Roles can play roles	*(M1, M0)*
9.	Roles can be transferred between objects	*(M0)*
10.	The state of an object can be role-specific	*(M0)*
11.	Features of an object can be role-specific	*(M1)*
12.	Roles restrict access	*(M0)*
13.	Different roles may share structure and behavior	*(M1)*
14.	An object and its roles share identity	*(M0)*
15.	An object and its roles have different identities	*(M0)*

2.6 CLASSIFICATION OF ROLES

Fifteen years after Steimann's attempt to unify the different definitions of roles, his observation that *"the divergence of definitions contradicts the evident generality and ubiquity of the role concept"* [Steimann, 2000b, p.84] still holds true. In particular, the focus on the context-dependent nature has led to various new definitions for roles. Accordingly, the list of classifying features of role, introduced by Steimann [2000b], is not sufficient anymore to account for the additional characteristics associated to roles. As a result, this section reevaluates Steimann's features and extends this list by new features to incorporate the characteristics found in the contemporary literature. In sum, the classification scheme, presented henceforth, was initially published in [Steimann, 2000b, Sec.2] and extended in [Kühn et al., 2014, Sec.3].

The classification proposed by Steimann [2000b] encompasses a list of 15 features attributed to roles. This list, enumerated in Table 2.1, mostly captures the behavioral and relational nature of roles. In fact, only Feature 2 denotes that *"roles depend on relationships"* [Steimann, 2000b, p.86]. Hence, all the other features contribute to the behavioral nature of roles. Features 1 and Feature 13, for instance, entail the existence of role types that can define fields and methods for the corresponding role instances, as well as inheritance among role types [Steimann, 2000b]. Similarly, Features 3 and Feature 7, classifies the *fills* relation between natural types and role types as *many-to-many* relation [Steimann, 2000b]. Features 4 and 5, in turn, describe the *plays* relation on the instance level as a dynamic relation between objects and role instances such that an object may play different instances of the same role type multiple times [Steimann, 2000b]. Moreover, according to Feature 6, the *plays* relation might be constrained to restrict the sequence of role acquisition and relinquishment [Steimann, 2000b]. This also entails, that a role instance can be transferred from one player to another player, captured in Feature 9. The actual effects of *playing a role* are captured in Feature 10, 11, and 12 stating that both the state and features (i.e., fields and methods) of an object can depend on roles. In other words, a role can adapt both the state and the features of its player. Conversely, a role can hide some of the object's features, and thus facilitates access re-

strictions to the object. In the discussion of Steimann, one controversial issue has been, whether roles themselves can play roles (Feature 8) and roles have an independent or shared identity (Feature 14 and 15, respectively). On the one hand, Steimann argues that *"an object in a role is the object itself"* [Steimann, 2000b, p.99] and consequently cannot play roles themselves. On the other hand, Boella and Van Der Torre [2007] contents that *"[an] agent and its roles have different identities"* [Boella and Van Der Torre, 2007, p.219] followed by the observation that *"roles are defined as agents and agents can play roles"* [Boella and Van Der Torre, 2007, p.219]. Others even maintain that the latter inadvertently results in *object schizophrenia* [Sekharaiah and Ram, 2002]. However, by focusing on the role's identity, both overlook that the question of identity is a question of perspective. To put it bluntly, the answer should depend on whether you look at the object playing a role from the outside or the inside. From the outside, an object and its roles should be indistinguishable, as they form a conceptual unit. However, from within a role the program must be able to discern between its own identity and the identities of both its player and the other played roles. As a solution, Herrmann [2007] postulates that to avoid *object schizophrenia "we only need to define two separate operators: == will continue to distinguish a role from its base whereas a second operator [...] considers all roles as identical to their base"* [Herrmann, 2007, p.196]. While this resolves the problem of *object schizophrenia*, it cannot avoid the ordering problem that occurs when *roles play roles*. In detail, the sequence of roles and subroles influences not only the object's behavior, but also its overall type. Consider a person being an employee and a consultant of the same bank. Is the employee playing the role of a consultant or is the consultant playing the employee role? Conceptually, there should be no difference as the person and its roles form a unit. However, as the subrole individually adapt the role's behavior both sequences might result in different behavior. As a result, if *roles play roles* the order of the roles that play other roles influences the behavior and thus the type of the compound object. In sum, Steimann's classification has proven to be useful and has been employed to classify several contemporary approaches, e.g., [Herrmann, 2007, Boella and Van Der Torre, 2007]. Nevertheless, his classification scheme has two major shortcomings. First, several features concern either the type and/or the instance level. For instance, Features 4, 5, 9, 10, 12, 14, and 15 only apply to role instances [Kühn et al., 2014]. Especially, Feature 5 and 12 are usually not applicable to modeling languages, as they do not provide an operational semantics. To indicate the affected level, Table 2.1 appends *M1* and *M0* to each feature denoting whether the type or the instance level is affected. Finally, it cannot fully distinguish contemporary RMLs and RPLs, as it does not capture the *context-dependent nature of roles* and the various *modeling constraints*. In conclusion, the following paragraph introduces the additional characteristics of roles found by investigating the contemporary literature.

In accordance to Steimann's pragmatic approach, the investigation of contemporary RMLs and RPLs, published in [Kühn et al., 2014, Sec.3.2] has uncovered the following features of roles. As this thesis studies the representation of roles and role models in role-based languages, we focused on the type level representation of roles rather than their implementation.

"16. Relationships between roles can be constrained" [Kühn et al., 2014, p.145]. If roles depend on relationships, these relationships might be further constrained by *intra-relationship constraints*, i.e., a relationship can be additionally classified as, for instance, reflexive, acyclic, total or exclusive parthood relation [Steimann, 2000b, Kim et al., 2003, Halpin, 2005, Genovese, 2007, Guizzardi and Wagner, 2012, Balzer et al., 2007].

"17. There may be constraints between relationships" [Kühn et al., 2014, p.145]. In contrast to the previous feature, this property indicates the existence of *inter-relationship constraints* in the language, i.e., that constraints between relationship types can be specified, such as subset and exclusion constraints [Halpin, 2005, Genovese, 2007, Guizzardi and Wagner, 2012].

Table 2.2: Additional classifying features, partially published in [Kühn et al., 2014].

16. Relationships between roles can be constrained	*(M1)*
17. There may be constraints between relationships	*(M1)*
18. Roles can be grouped and constrained together	*(M1)*
19. Roles depend on compartments	*(M1, M0)*
20. Compartments have properties and behaviors	*(M1, M0)*
21. A role can be part of several compartments	*(M1, M0)*
22. Compartments may play roles like objects	*(M1, M0)*
23. Compartments may play roles which are part of themselves	*(M1, M0)*
24. Compartments can contain other compartments	*(M1, M0)*
25. Different compartments may share structure and behavior	*(M1)*
26. Compartments have their own identity	*(M0)*
27. The number of roles occurring in a compartment can be constrained	*(M1)*

"18. Roles can be grouped and constrained together" [Kühn et al., 2014, p.145]. Although several approaches restrict roles [Dahchour et al., 2002, Ferber et al., 2004, Herrmann, 2005, Baldoni et al., 2006c], they usually omit grouping related roles and constraining the whole group of roles, as suggested in [Ferber et al., 2004, Herrmann, 2005, Zhu and Zhou, 2006].

In sum, these properties reflect the different constraints for roles and relationships proposed in contemporary RMLs. Nevertheless, to classify the context-dependent nature of roles, the following features highlight the characteristics of compartments.

"19. Roles depend on compartments" [Kühn et al., 2014, p.146]. Most of the recent approaches agree that roles are dependent on some sort of compartment, e.g., *organization* [Da Silva et al., 2003], *environment* [Ubayashi and Tamai, 2001, Zhu and Zhou, 2006], *ensemble* [Hennicker and Klarl, 2014], *collaboration* [Pradel and Odersky, 2009], *institution* [Baldoni et al., 2006c], or *context* [Genovese, 2007, Hu and Liu, 2009, Reenskaug and Coplien, 2009, Kamina and Tamai, 2010, Hennicker and Klarl, 2014]. By extension, a university represents the prototypical example of a compartment that defines *student* and *teacher* roles collaborating in *Courses* [Herrmann, 2007, Balzer et al., 2008, Liu and Hu, 2009b].

"20. Compartments have properties and behaviors" [Kühn et al., 2014, p.146]. Like objects and roles, compartments are sometimes considered as types with specified state and behavior [Serrano and Ossowski, 2004, Herrmann, 2005, Baldoni et al., 2006c, Genovese, 2007, Liu and Hu, 2009a, Pradel and Odersky, 2009, Kamina and Tamai, 2009, Hennicker and Klarl, 2014].

"21. A Role can be part of several compartments" [Kühn et al., 2014, p.146]. In detail, this property describes that a *role type* can be part of multiple compartment types [Ferber et al., 2004, Zhu and Zhou, 2006, Baldoni et al., 2006c, Genovese, 2007, Kamina and Tamai, 2009]. Consider again the role type *customer*. It can be used in different compartments, e.g. a *Bank* or a *Shop*, where it might be implemented and constrained differently.

"22. Compartments may play roles like objects" [Kühn et al., 2014, p.146]. Although, compartments are usually employed to group context-dependent roles, several approaches treat compartments like objects and permit them to play roles [Herrmann, 2005, Balzer et al., 2007, Genovese, 2007, Liu and Hu, 2009a, Pradel and Odersky, 2009, Harkes and Visser, 2014].

"23. Compartments may play roles which are part of themselves" [Kühn et al., 2014, p.146]. If compartments might play roles, it might be permitted that a compartment plays a role belonging to itself [Genovese, 2007, Herrmann, 2005].[5] Consider the banking example, here the feature would allow a *bank* compartment instance to become a *customer* of itself.

"24. Compartments can contain other compartments" [Kühn et al., 2014, p.146]. Independent of the previous features, some approaches support the specification of compartments within compartments [Ubayashi and Tamai, 2001, Da Silva et al., 2003, Herrmann, 2005, Liu and Hu, 2009a, Pradel and Odersky, 2009, Kamina and Tamai, 2009]. This, so called *nesting*, is introduced to permit the subdivision of compartments into sub-compartments [Da Silva et al., 2003, Herrmann, 2005, Kamina and Tamai, 2009]. The *bank* compartment type, for instance, could contain both a *retail banking* and an *investment banking* sub-compartment types, to reflect the two different financial services provided by a *bank*.

"25. Different compartments may share structure and behavior" [Kühn et al., 2014, p.146]. Similar to classical types, compartment types might inherit properties, features, roles, and constraints from each other [Herrmann, 2005, Genovese, 2007, Nelson et al., 2008, Liu and Hu, 2009a, Pradel and Odersky, 2009]. As such, compartment inheritance is closely related to *family polymorphism* [Ernst, 2001, Igarashi et al., 2005] as recognized by Herrmann et al. [2004].

"26. Compartments have their own identity" [Kühn et al., 2014, p.146]. This property is acknowledged by all approaches featuring compartments as first-class citizens [Ubayashi and Tamai, 2001, Da Silva et al., 2003, Serrano and Ossowski, 2004, Ferber et al., 2004, Herrmann, 2005, Zhu and Zhou, 2006, Liu and Hu, 2009a, Reenskaug and Coplien, 2009, Pradel and Odersky, 2009, Kamina and Tamai, 2009, Hennicker and Klarl, 2014]. Moreover, compartments can only exist on the instance level, if they have their own identity.[6]

27. The number of roles occurring in a compartment can be constrained. In particular, *occurrence constraints* enforce and restrict the number of instances of a specific type that must exist within one compartment instance throughout its lifetime [Kim et al., 2003, Zhu and Zhou, 2006, Hennicker and Klarl, 2014, Harkes and Visser, 2014].

In sum, these additional features, summarized in Table 2.2, not only incorporate the *context-dependent nature of roles*, but also the various constraints found in contemporary RMLs and RPLs. Accordingly, Steimann's initial list encompasses both the behavioral and relational nature of roles, the additional features extend the relational nature and add the context-dependent nature to roles. Consequently, while Steimann's 15 features are still applicable, they need to be accompanied by 12 additional features to sufficiently assess the diversity of the contemporary role-based languages. As a result, this list of 27 characteristic features allows for a fine-grained distinction of the various definitions of roles presented in the contemporary literature. Henceforth, this classification scheme is employed to review, compare and classify the various contemporary RMLs and RPLs.

[5] This feature is described in §2.1.2 (b) of `Object Teams/Java`'s language definition [Herrmann and Hundt, 2013]}.
[6] The question whether compartments have a unique or composite identity will be discussed in Chapter 7.1.

"A systematic literature review is a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest."

— Kitchenham [2004]

3 SYSTEMATIC LITERATURE REVIEW

After establishing a suitable classification scheme, this chapter describes the Systematic Literature Review (SLR) [Kitchenham, 2004] conducted to survey the contemporary literature on role-based modeling and programming languages. Following Kitchenham's definition above, an SLR is a *structured* process to gather, classify, and interpret published results related to a specific topic or area of research. However, the main advantage of an SLR is that it not only identifies and evaluates related work, but also provides information on how and why related work has been selected and how it has been evaluated [Kitchenham, 2004]. In general, a systematically performed literature review has the following features. First, they define a precise review protocol that allows for evaluating individual publications [Kitchenham, 2004]. Second, systematic reviews specify a search strategy to collect and select as much of the relevant literature as possible [Kitchenham, 2004]. Consequently, SLRs additionally provide explicit inclusion and exclusion criteria to elucidate the selection process [Kitchenham, 2004]. Last but not least, they stipulate the information that should be obtained from each relevant approach [Kitchenham, 2004]. In conclusion, an SLR gives considerable insight into the selection and evaluation process of the literature review. This, in turn, increases the comprehensibility, replicability, and quality of the performed literature review.

Even though SLRs are generally performed by a large group of researchers [Kitchenham, 2004], the general process is also applicable by PhD students performing a literature review. Ultimately, the review process has three distinct stages [Kitchenham, 2004]. The *planning stage* identifies the need for a review and the underlying research question, as well as develops the review protocol [Kitchenham, 2004]. The second stage is tasked with *conducting the review* [Kitchenham, 2004]. At this stage, relevant publications are identified, appropriate approaches are picked, and relevant information on each approach is extracted. Finally, the quality of the selected approaches is assessed and the evaluation results are synthesized. The third stage, *reporting the review*, outlines the research protocol, results, and limitations [Kitchenham, 2004]. The resulting process, however, is not a strict sequence of actions. Kitchenham herself emphasizes that *"many activities are initiated during the protocol development stage, and refined when the review proper takes place."* [Kitchenham, 2004, p.3]. In other words, she argues that this process should not be viewed as sequential but as iterative, such that both the review protocol, the selection criteria, and the evaluation method is refined with each iteration to improve the quality of the conducted SLR. Accordingly, the discussion henceforth will not only describe the process and methods used to conduct the SLR, but also highlight the various iterations and adjustments employed throughout the review process.

In conclusion, this chapter outlines the literature review performed throughout the writing of this thesis. Following Kitchenham's classification [Kitchenham, 2004], this SLR of contemporary RMLs and RPLs is conducted *"[to] provide a framework/background in order to appropriately position new research activities"* [Kitchenham, 2004, p.2]. More precisely, it presents a systematic review of all role-based modeling and programming languages published since the year 2000 evaluating each approach with respect to the 27 classifying features of roles, introduced in Chapter 2. However, while Chapter 4 and Chapter 5 discusses the identified RMLs and RPLs, respectively; this chapter describes the details of the conducted SLR by following the general structure of systematic reviews [Kitchenham, 2004, Tab.9]. Therefore, Section 3.1 gives a detailed description of the employed review process focusing on the collection and selection of relevant literature. Section 3.2, in turn, summarizes the results of the collection process culminating in the selection of 25 contemporary role-based modeling and programming languages. Last but not least, Section 3.3 assesses the quality of the performed SLR, discusses limitations in the collection process, and possible biases in the paper selection process.

3.1 METHOD

Before starting a literature review, it is crucial to identify the need for an SLR and corresponding research question first. In case of role-based modeling and programming languages, this need arises from the fact that the latest major literature reviews date back to 2000 for conceptual modeling [Steimann, 2000b] and 2008 for information systems [Zhu and Zhou, 2008b]. Hence, they cannot account for more recent developments in the field. More importantly, while both survey's cover a huge body of literature, they do not provide any insight into their individual literature collection, selection and evaluation procedures. As a result, neither of them can be easily reproduced and/or extended to include more recent related works. Admittedly, this only explains the *reason* to conduct an SLR on contemporary RMLs and RPLs but not the underlying *research question* the literature review intends to answer. Basically, this SLR aims at evaluating the various definitions of roles found in the literature since the year 2000 by applying the list of classifying features of roles. Conversely, this evaluation should answer the following three research questions:

1. *Is there a common subset of features all contemporary approaches satisfy?*
2. *How did Steimann's seminal work influenced the research field?*
3. *Have advances in RMLs been adopted by later RPLs and vice versa?*

To answer these questions, this survey aims at providing a thorough investigation and evaluation of contemporary role-based modeling and programming languages. As already pointed out, there exist two major surveys identifying various related publications [Steimann, 2000b, Zhu and Zhou, 2006], and providing a possible classification scheme. Steimann's features of roles, for instance, were derived from role-based modeling and programming languages. Zhu's classification, in contrast, was developed with a much broader scope including both *roles in access control*, *roles in agent systems* and *roles in social psychology and management*. As a result, Steimann's 15 features of roles represented a more suitable initial classification scheme for this SLR. However, this classification scheme was continuously augmented with 12 additional features identified during the review (cf. Chapter 2.6).

The review process went through several iterations to identify a suitable *search strategy* and distinct *inclusion/exclusion criteria* for relevant publications. As a preliminary search, I looked through those publications directly referencing and applying Steimann's classifications [Steimann, 2000b].

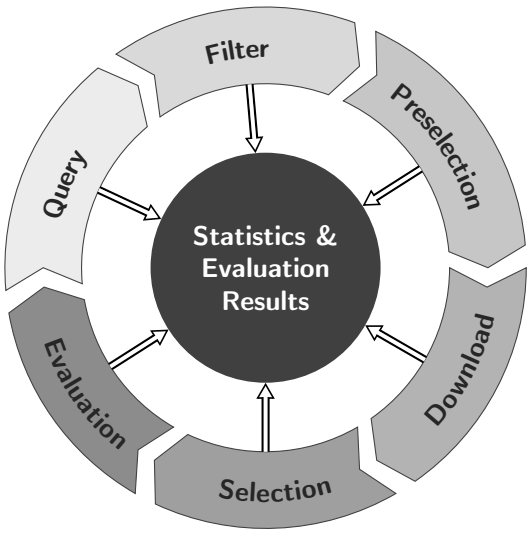

Figure 3.1: Visualization of the review process

Unfortunately, from the 442[1] publications referencing Steimann's seminal paper, only few published modeling and programming languages have applied his classification, e.g. [Herrmann, 2005, Boella and Van Der Torre, 2007, Pradel and Odersky, 2009]. It follows, then that just focusing on the publications referring to [Steimann, 2000b] is insufficient.[2]

Therefore, a more sophisticated *review process* was devised to identify as much of the related work as possible. Figure 3.1 depicts the resulting process and its six phases. The full review process was performed twice. The first iteration was conducted in the beginning of this thesis in April 2014 and the second iteration at its end in September 2016. In the *query* phase, a suitable electronic database is queried for all publications with respect to a given query string. Henceforth, *Google Scholar*[3] was employed as one of the biggest bibliographic databases supporting full-text search of publications. Although I concede that there are many other electronic databases available, none of them encompasses as many publications while still providing full-text search. To put it bluntly, *Google Scholar* was estimated to encompass 87 percent of all English publications available on the web in 2014 [Khabsa and Giles, 2014]. Therefore, *Google Scholar* was queried for all papers published since 2000 containing the exact phrase "role based"[4], the words "software", "programming", "language", and either "modeling" or "modelling" to account for the British spelling. However, as this also included publications from unrelated research topics, e.g. psychology, sociology, and biology, the query was augmented to exclude publications containing either of the following terms "rbac", "policy", "sociology", "bio", and "psycho". As a result, *Google Scholar* returned 3433 publications in April 2014 and 4183 in September 2016. For each of these publications a `BibTex` entry was retrieved. This entry includes information on the authors, the title, its publisher, the citation count, and a link to the publication's website. After querying, the *filter* phase excludes all items that have not been peer reviewed for publication, such as technical reports, master or PhD theses. While one could retain all publications classified as `article`, `incollection`, `inproceedings`, and `inbook`, this would also include arbitrary documents found on the web, as *Google Scholar* classifies these

[1] Estimated by *Google Scholar* on 29th of September 2016.
[2] A similar effect can be found following the citations of [Zhu and Zhou, 2006].
[3] `https://scholar.google.com`
[4] Note, this will also detect the form "role-based" as the dash is ignored by Google's search engine.

as `article`, as well. Hence, it is more reasonable to only consider literature published by the big publishers, because they individually enforce peer review and scientific standards for their publications. Conversely, the *filter* phase automatically removed all entries that have not been published by one of the following four publishers: *Association for Computing Machinery (ACM)*,[5] *Institute of Electrical and Electronics Engineers (IEEE)*,[6] *Springer*,[7] *ScienceDirect*.[8] This, in turn, should ensure that all considered publications follow scientific practices and have been peer reviewed. Admittedly, there are several other options to automatically filter publication such as *filter by citation count*, *filter by class*, *filter by keywords*, yet, none of them ensures peer review and include most of the related publications. Filtering by citation count, for instance, assuming a citation count above $12 \cdot log_{10}(age)$, would have excluded 38 percent of the relevant approaches (18 of 47). As a result, the *filter* phase retained 1311 publications in April 2014 and 1501 in September 2016. After limiting the number of entries in the data set to a reasonable amount, the *preselection* phase further reduced the number of publications by examining their abstract and occurrences of the word *role* in the document. While the abstract not necessarily indicates the introduction of a role-based language, looking through the occurrences of *role* permits to quickly discern publications that define a role-based language from those that plainly use the form "role-based". In sum, 133 publications have been selected as relevant in the first iteration and 3 additional publications in the second iteration. Accordingly, each of these relevant publications have been retrieved during the *download* phase in order to further evaluate each approach. Up to this point, the data set included role-based languages and approaches from a wide variety of application domains, e.g. conceptual modeling, Multi-Agent Systems (MAS), Business Process Modeling (BPM), Role-Based Access Control (RBAC), Self-Adaptive Systems (SAS). Despite the fact that all these approaches provide definitions for roles in their respective domain, this SLR solely focuses on conceptual modeling languages and programming languages regardless of their use in a particular domain. In general, all publications that either introduce a role-based modeling language (RML) or a role-based programming language (RPL) (extension) have been selected. Furthermore, only those publications have been included that provided enough information on their definition of roles to feasibly evaluate the 27 classifying features of roles. Conversely, most of the formalisms and frameworks for MAS, as well as all role-based workflow languages for BPM and specifications for RBAC have been excluded from the data set. Besides, I also excluded all publications that have been published within the RoSI research training group, to avoid a biased evaluation due to my affiliation to the project and authors. In conclusion, after the preliminary search identified 6 publications of role-based modeling and programming languages, the systematic selection process found 47 additional publications. Naturally, most approaches are described and extended over multiple publications, however, each approach is only evaluated once with respect to all the corresponding publications. Accordingly, the *evaluation* phase evaluates, which of the 27 features of roles are supported by the selected approaches. For each role-based language, the respective (formal) definition of the underlying role concept have been investigated to evaluate whether they fully, partially or fail to support a given feature of roles. In detail, a feature is considered *fully supported* if it is described in a corresponding publication or implemented in a prototypical implementation. If an approach, for instance, explicitly states that roles have their own identity or implement roles as adjunct instances [Steimann, 2000b], then this approach clearly satisfies Feature 15. Conversely, a *partially supported* feature is not directly mentioned or implemented, but could be facilitated by using another feature of the given approach. Consider an approach, that employees arbitrary logical formulas to specify constraints.

[5] `https://www.acm.org`
[6] `https://www.ieee.org`
[7] `https://link.springer.com`
[8] `http://www.sciencedirect.com`

Table 3.1: Statistics of the paper selection process.

Year	Query	Filter	Preselection	Selection
2000	74	32	3	1
2001	93	34	3	1
2002	156	43	3	1
2003	177	57	9	2
2004	223	59	10	5
2005	277	95	11	4
2006	310	111	12	6
2007	343	143	9	4
2008	333	134	8	4
2009	334	137	12	4
2010	351	145	8	2
2011	353	117	9	1
2012	367	131	7	1
2013	303	101	15	1
2014	330	126	9	7
2015	133	61	9	3
2016	26	7	2	1
Sum	4183	1533	139	48

While this approach might not explicitly support intra-relationship constraints, it is still possible to express them as logical formulas. Thus, intra-relationship constraints (Feature 16) are partially fulfilled. Last but not least, a feature is considered *not supported* if it is either stated or implied as such in the corresponding definition. Again, if an approach defines roles as specializations and/or generalizations [Steimann, 2000b] or implements them via static *traits*, then it can be inferred that this approach does not feature roles with their own identity (Feature 15). In accordance, a questionnaire was established comprising the 27 features of roles that was completed for each approach. The results of this evaluation have been collected in a spreadsheet. In addition to that, both the included and the excluded `BibTex` entries have been retained to derive statistical information on the number of publications excluded after each phase of the review process. In sum, the described review process not only identified a broad spectrum of relevant role-based modeling and programming languages, but also elucidated the employed selection and evaluation process. Furthermore, this process is easily reproducible, especially, because the *query*, *filter* and *download* phase can be automated.[9] Hence, the resulting review process fulfills the main quality requirements for SLRs established by Kitchenham [2004].

3.2 RESULTS

Up to this point the discription focused on the methodology of the conducted SLR. Hence, this section discusses the results of the review process. In general, this SLR on contemporary RMLs and RPLs identified 25 distinct approaches. By extension, the second iteration of the review process retrieved 4183 entries from *Google Scholar* in the *query* phase. From them 1533 remained after filtering for publications of the big four publishers. In the remaining data set 139 related publications

[9] `https://github.com/Eden-06/gsresearch/tree/master/workflow`

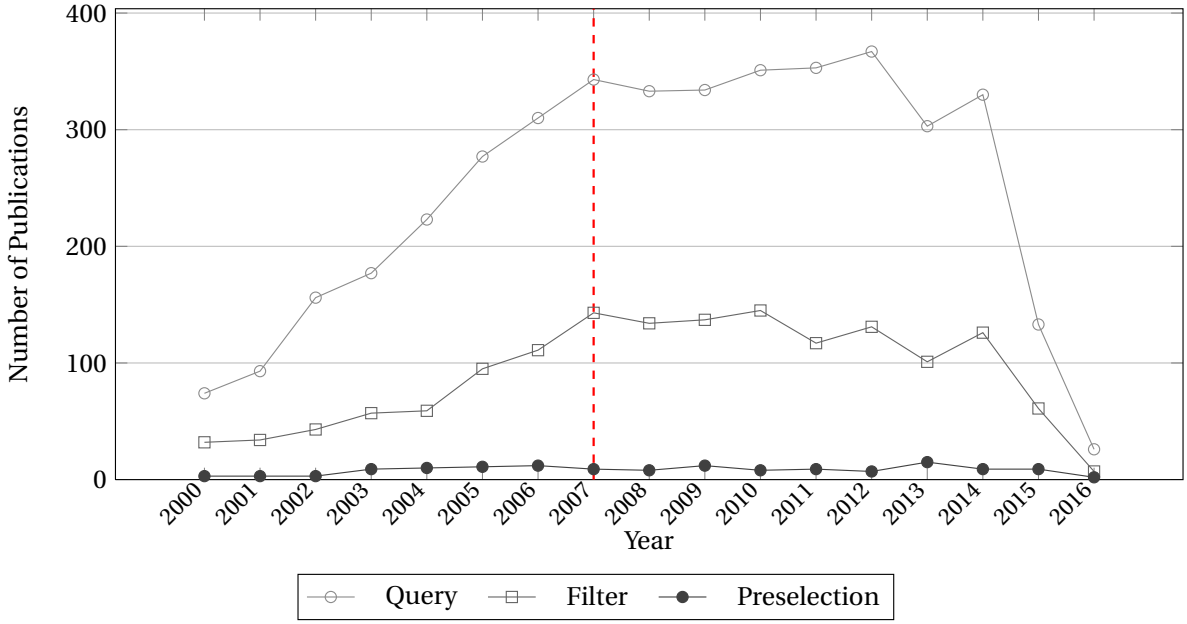

Figure 3.2: Number of publications per year from the query to the preselection phase.

have been identified. After downloading and studying them, 48 publications have been selected for further evaluation in addition to the 7 publications identified in the preliminary search. In sum, these publications describe 25 distinct role-based modeling and programming languages. In turn, Table 3.1 shows the number of publications that have been selected after the *query*, *filter*, *preselection*, and *selection* phase for each year from 2000 until 2016. Nevertheless, the evaluation of these 25 RMLs and RPLs will be deferred to Chapter 4 and Chapter 5, respectively. Thus, the discussion henceforth concentrates on the data collected during the review process, such as the number of publications per year, publications per phase, and overall distribution of relevant publications in the given time frame. This is important, because it unveils the various intricacies of the research field and allows for validating some assumptions present in the research field.

Steimann, for instance, noted that the "interest in roles has grown continuously" [Steimann, 2000b, p.84]. While this was true back then, the data set suggests that the interest has only grown until the year 2007. Since then, the number of new publications found in the query phase remained consistent at a mean of 339.25 (±19.1889) publications per year. Additionally, Figure 3.2 shows that this pattern still persists after the *filter* phase only including publications of the four biggest publishers. Notably, the last two years should be considered outliers as *Google Scholar* does not immediately include all new publications into their database. In conclusion, it is still true that there is a persistent interest in the notion of roles, although, it has not grown in the past 9 years. While this is true for general role-based approaches, the situation is more complex when considering the relevant role-based languages. This complexity is best illustrated by Figure 3.3 highlighting the distribution of publications over the years for both the relevant and the selected publications. In general, the number of publications increased until its peak of 12 publications in 2006, since then the number of publications per year fluctuated around a mean of 9.625 with a rather high standard deviation of 2.615. More surprisingly, the distribution of the selected publications indicates that research on these languages is conducted in waves. In other words, there has been a boom of new RMLs and RPLs between 2003 and 2006 resulting in several corresponding publications.

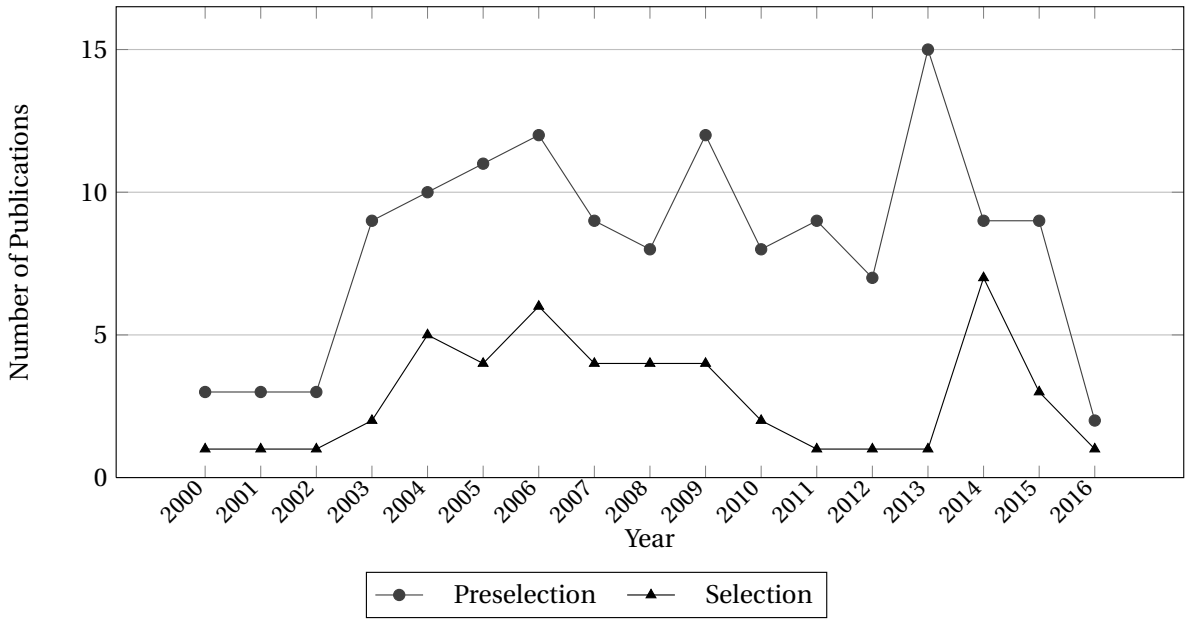

Figure 3.3: Number of publications per year for the preselection and selection phase.

However, most of these languages are only introduced and described in one or two subsequent publications and rarely picked up again afterwards. *EpsilonJ* [Ubayashi and Tamai, 2000], *Object-Role Modeling (ORM)* [Halpin, 1998], and *E-CARGO* [Zhu and Zhou, 2006] represent the only exceptions to this apparent discontinuity in the field of role-based modeling and programming languages. Similarly, the influence of Steimann's seminal paper [Steimann, 2000b] on these role-based languages was rather limited. In fact, only few of the 25 selected approaches reference [Steimann, 2000b]. For instance, only [Herrmann, 2005], [Boella and Van Der Torre, 2007] and [Pradel and Odersky, 2009] employ Steimann's classification scheme. From all this, it becomes evident that the research field on role-based modeling and programming languages suffers from *discontinuity*, i.e., none of the investigated approaches reused results of another related approach [Kühn et al., 2014].

Nevertheless, the main goal was not the examination of the research field, but the systematic selection, identification, and evaluation of contemporary role-based languages. Essentially, this SLR identified the following distinct role-based modeling languages:

- **Lodwick** is a formal RML unifying the behavioral and relational nature of roles and proposing a lightweight extension to UML [Steimann, 2000b].
- The **Generic Role Model** is a modeling language incorporating the dynamic relation between objects and roles [Dahchour et al., 2002].
- **Taming Agents and Objects (TAO)** is a conceptual framework for the design of MAS [Da Silva et al., 2003, Da Silva and De Lucena, 2004, 2007, Adamzadeh et al., 2014].
- The **Role-Based Metamodeling Language (RBML)** is a conceptual metamodeling language dedicated to the specification and identification of design patterns in UML [Kim et al., 2002, 2003, France et al., 2004, Kim and Whittle, 2005, Kim and Shen, 2007, Kim, 2008, Kim and Lu, 2008, Kim and Lee, 2015].
- The **Role Concepts in Patterns** establishes a formal role-based metamodeling language for the specification of design patterns combining UML and Object-Z [Kim and Carrington, 2004, 2005, 2009].

- The **Object-Role Modeling (ORM)** *(Version: 2)* is a well-established, fact-oriented data modeling language including roles as places of relationships and a wide variety of modeling constraints [Halpin, 1998, 2005, 2006, Curland et al., 2009].
- **E-CARGO** is a formal model for role-based collaborative systems featuring environments, agents, and roles. It manly focuses on the context-dependent nature of roles [Zhu and Zhou, 2006, Zhu, 2005, Liu and Zhu, 2006, Zhu, 2007, Zhu and Zhou, 2008a, 2009, Liu et al., 2014, Sheng et al., 2014, Zhu, 2016, Sheng et al., 2016].
- The **Metamodel for Roles** is a formal model proposed to combine the various behavioral and context-dependent definitions of roles [Genovese, 2007].
- The **Information Networking Model (INM)** is a data modeling language designed to store, query, and manipulate context-dependent information [Liu and Hu, 2009a].
- The **Data Context Interaction (DCI)** architecture is a software development methodology introducing roles to capture the context-dependent behavior of objects [Reenskaug and Coplien, 2009, Qing and Zhong, 2012, Zat'ko and Vranic, 2015].
- **OntoUML** is an ontologically well-founded extension of UML incorporating concepts such as role types and relationship types [Guizzardi and Wagner, 2012].
- The **Helena Approach** is novel a methodology for the design of distributed autonomic systems based on a formal RML combining all natures of roles [Hennicker and Klarl, 2014, Klarl et al., 2014].

Additionally, the SLR found the contemporary role-based programming languages listed below:

- **EpsilonJ** is a programming language for role-based evolutionary programming of context-dependent applications [Ubayashi and Tamai, 2000, 2001, Tamai et al., 2005, 2007, Monpratarnchai and Tetsuo, 2011, Tamai and Monpratarnchai, 2014].
- **Chameleon** is a programming language extension to Java solely focusing on the behavioral nature and the dynamic dispatch of roles [Graversen and Østerbye, 2003].
- **RICA-J** is a role-based programming language (RPL) extension developed to support the implementation of MAS by means of communicative roles and interactions [Serrano and Ossowski, 2004, Serrano et al., 2006].
- **JAWIRO** is a Java library for implementing role-based applications establishing behavioral roles [Selçuk and Erdoğan, 2004, 2006].
- **ObjectTeams/Java (OT/J)** is a matured programming language extension adding teams and roles to embrace both the behavioral and context-dependent nature of roles [Herrmann, 2005, 2007, Al-Zaghameem, 2010, Herrmann, 2010].
- **Rava** is a small preprocessor for Java introducing roles and role invocation to facilitate roles encompassing the dynamic behavior of objects [He et al., 2006].
- **PowerJava** is another programming language extension incorporating roles and institutions to bridge the gap from Java to MAS [boella2006ont; Baldoni et al., 2006c,a, 2008].
- **Rumer** is a formally defined programming language introducing first-class relationships, member interposition, and invariants over shared state [Balzer et al., 2007].
- **First Class Relationships** proposes a new programming model combining objects and associations with relationships and roles [Nelson et al., 2008, Pearce and Noble, 2006].
- **Scala Roles** is a lightweight language extension establishing both roles and collaborations as first-class citizens [Pradel and Odersky, 2009].
- **NextEJ** is the successor of the EpsilonJ programming language improving the implementation of context-dependent roles [Kamina and Tamai, 2009, 2010].

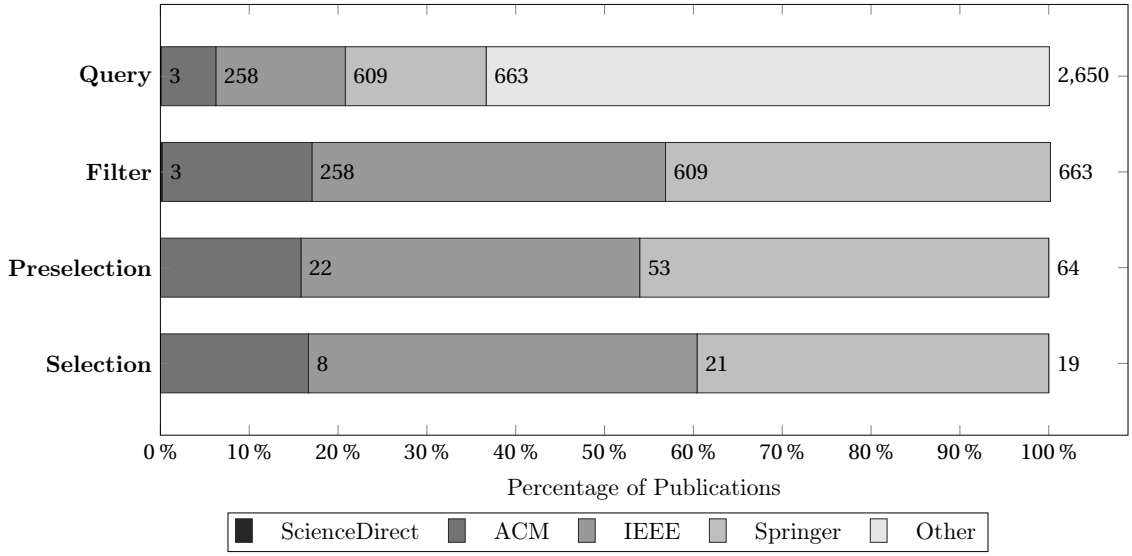

Figure 3.4: Distribution of publications per publishers for particular phases.

- **JavaStage** is a role-based programming language (RPL) extension to Java only supporting static binding of roles [Barbosa and Aguiar, 2012].
- **Relations** is a role-based data modeling language featuring first-class relationships and elevates cardinalities to the type system [Harkes and Visser, 2014].

3.3 DISCUSSION

Although this SLR was carefully performed to identify, select, and evaluate as much of the relevant literature as possible, this section assesses its quality and discusses its limitations. According to Kitchenham [2004], the quality of a study is determined by *"the extent to which the study minimises bias and maximises internal and external validity"* [Kitchenham, 2004, p.10]. Likewise, the quality of an SLR corresponds to the means employed to avoid bias, as well as the methods facilitated to ensure validity.

In general, *bias* refers to the *"tendency to produce results that depart systematically from the 'true' results."* [Kitchenham, 2004, p.11]. Basically, bias is any influence that might invalidate or distort the validity of the obtained results. Considering this literature review, the results can be biased in two ways. First, relevant publications might be excluded from the dataset in the various phases. This might be the case, if a publication did not contain the phrase "role-based", such as [Dahchour et al., 2002, Graversen and Østerbye, 2003, He et al., 2006]. Nonetheless, the aforementioned approaches have been identified in the preliminary search and have also been considered for the evaluation. Similarly, if a publication has not been published via one of the four publishers, the *filter* phase would have automatically excluded it from the dataset. However, these four publishers guarantee a minimum level of quality and, more importantly, peer review for their publications. Admittedly, the review indicates that only *ACM*, *IEEE*, and *Springer* publish relevant role-based languages. As case in point, Figure 3.4 shows the distribution of publications per publisher for particular phase. In contrast to the *query* and *filter* phase, the *preselection* and *selection* phase have been performed manually. Hence, the selection might be biased by the author's prejudice. Unfortunately, the only way to minimize the *exclusion bias*, would be to let multiple other researchers repeat both selection phases with the same inclusion/exclusion criteria and combine the identified publications.

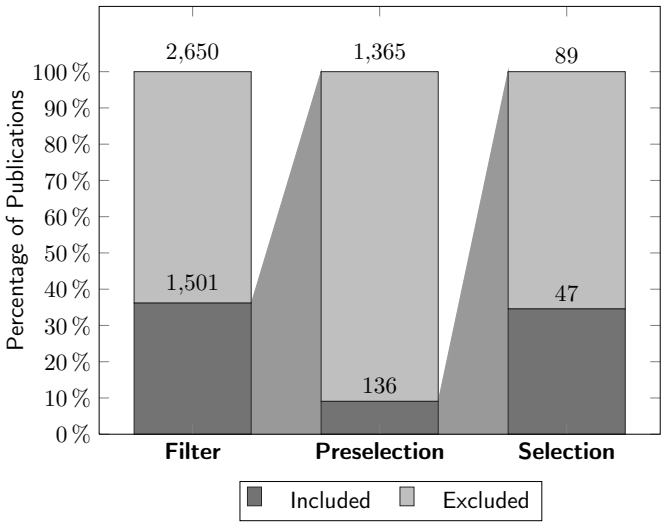

Figure 3.5: Comparison of selectivity of the filter, preselection and selection phase.

Naturally, this is impractical for a literature review within a PhD thesis. As an overview of the publication selection, Figure 3.5 highlights the percentage of publications included and excluded during the *filter*, *preselection*, and *selection* phase. In summary, the performed literature review could identify most of the relevant contemporary literature. Second, different approaches might not be evaluated objectively. Consequently, to ensure objectivity each approach is evaluated with respect to a fixed classification scheme (cf. Section 2.6) taking all corresponding, available publications and prototypical implementations into account. Granted, this only limits the possible *evaluation bias* to the individual features of roles. However, the evaluation bias can only be avoided, if more researchers are involved and approaches can be randomly assigned to researchers for evaluation.

In contrast to minimizing bias, ensuring *internal validity* means that *"the design and conduct of the study are likely to prevent systematic errors"* [Kitchenham, 2004, p.11]. To facilitate internal validity, the review process was performed in a semi-automatic fashion. Basically, the whole review process is implemented in a set of shell scripts that are available on *GitHub*.[10] In particular, it was possible to automate the *query*, *filter*, and *download* phase. Thus, it is very unlikely that publications have been missed or falsely excluded during these phases. Moreover, both the *preselection* and *selection* phase are supported by an interactive tool. In fact, I strictly followed the inclusion/exclusion criteria to grant validity of the selection method. Conversely, the questionnaire for the 27 features of roles was utilized in the *evaluation* phase, to collect and document the investigation of individual approaches. After each phase, all the relevant statistics on the included `BibTex` entries were automatically collected and stored in a dedicated file. This, in turn, allowed for monitoring and adjusting the effects of the employed exclusion criteria. As a result, the implementation of the semi-automatic review process and its disciplined execution facilitate internal validity of this SLR. In conclusion, this report of the conducted SLR accompanied by the implementation of the employed review process not only grant its comprehensibility and reproducibility, but also its quality.

[10] `https://github.com/Eden-06/gsresearch/tree/master/workflow`

"[The] role concept is a truly original one, one that cannot be emulated by any of the better established conceptual or object-oriented modelling constructs."

— Steimann [2000b]

4 CONTEMPORARY ROLE-BASED MODELING LANGUAGES

Despite the fact that most modeling languages already feature roles as named association ends, they cannot capture the full potential and nature of roles. According to Steimann [2000b] quoted above, those languages fail to appropriately represent the role concept. Consequently, the survey and the discussion henceforth focuses on those contemporary modeling languages that feature roles as *first-class citizens* including both formal modeling languages, data modeling languages, and conceptual modeling languages. Accordingly, each of the identified role-based modeling languages (RMLs) is summarized, illustrated with an example, and evaluated by applying the 27 features of roles (cf. Chapter 2.6). In general, this chapter is based on the results published in [Kühn et al., 2014]. For the sake of consistency, however, this chapter is structured in accordance with the supported nature of roles. Thus, Section 4.1 emphasizes languages that feature the behavioral and/or relational nature of roles. Section 4.2 comprises languages focusing on the context-dependent nature of roles. Last but not least, Section 4.3 highlights the few languages combining both the relational and context-dependent nature of roles. In short, this chapter collects and reviews the state-of-the-art in role-based modeling.

4.1 BEHAVIORAL AND RELATIONAL MODELING LANGUAGES

The following section discusses six RMLs that concentrate on the behavioral nature, the relational nature of roles, or both. Specifically, the *Generic Role Model* [Dahchour et al., 2002] (Section 4.1.2) is the only modeling language only incorporating the behavioral nature of roles, and *ORM 2* [Halpin, 2005] (Section 4.1.5) is the only RML solely focusing on the relational nature of roles. Conversely, the other modeling languages, such as *Lodwick* (Section 4.1.1), *RBML* (Section 4.1.3), *Role-Based Pattern Specification* (Section 4.1.4), and *OntoUML* (Section 4.1.6), successfully combine both the behavioral and relational nature of roles.

Listing 4.1: Bank example specified with Lodwick.

```
1  advises: Consultant Customer
2  Person  <NR Consultant
3  Person  <NR Customer  Company <NR Customer  Bank <NR Customer
4  own_ca: CA_Owner CheckingAccount
5  CA_Owner ≤RR Customer
6  Person  <NR CA_Owner  Company <NR CA_Owner  Bank <NR CA_Owner
7  Account <NR CheckingAccount
8  own_sa: SA_Owner SavingsAccount
9  SA_Owner ≤RR Customer
10 Person  <NR SA_Owner  Company <NR SA_Owner  Bank <NR SA_Owner
11 Account <NR SavingsAccount
12 transaction: Source Target
13 Account <NR Source      Account <NR Target
```

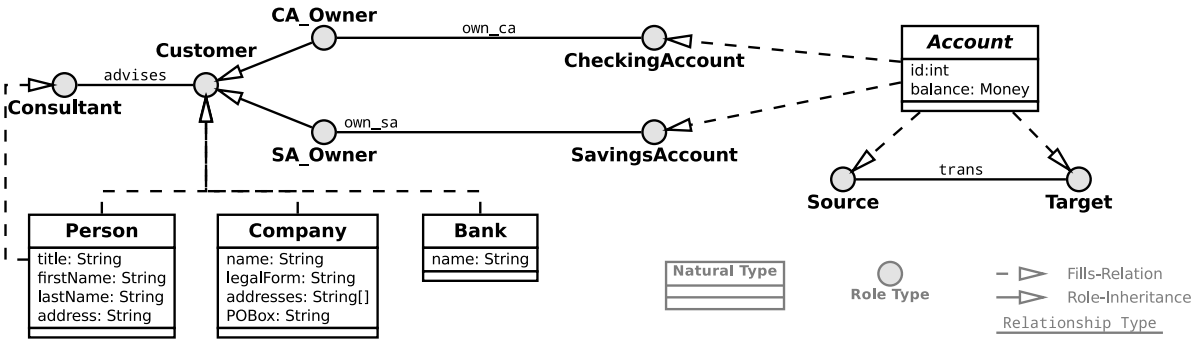

Figure 4.1: Corresponding representation using the revised UML notation.

4.1.1 LODWICK

Lodwick [Steimann, 2000b] is one of the first formal modeling languages for behavioral and relational roles. It is designed by Friedrich Steimann as an attempt to consolidate the various notions of roles in conceptual modeling. Its formal definition includes *natural types* filling *role types*, whereas the latter are placeholders in n-ary *relationships*. While these types do not have properties or methods, they are modeled with an intention $int(x)$, a logical formula for its behavior, an extension $ext(x)$, a set of instances of this type, as well as a disjoint inheritance hierarchy [Steimann, 2000b]. Hence, this limits the definition of the operational semantics to the behavior defined as propositional formulae. Moreover, roles are only represented on the conceptual model and do not carry on to instances of that model [Steimann, 2000b]. In other words, although role types are first-class citizens on the model level, role instances are not represented on the instance level. Consider, for instance, the banking scenario modeled in *Lodwick*, depicted in Listing 4.1. In this model, the *fills* relation is defined with `Player` $<_{NR}$ `Role`, individual relationships with `rel: Role1 ... RoleN` and role inheritance with `SuperRole` $\leq_{RR}$ `SubRole`. Accordingly, this example defines the natural types `Account`, `Person`, and `Company` as well as the relationships `trans`, `advises`, `own_ca`, and `own_sa`. Notably, *Lodwick* enforces that a role type only belongs to one relationship [Steimann, 2000b]. Hence, to specify the various relationships of the `Customer` role type, two subrole types `CA_Owner` and `SA_Owner` must be added for the relationships `own_ca` and `own_sa`, respectively. In addition to the textual representation of *Lodwick*, Steimann [2000c] proposed a revised version of UML class diagrams to allow for specifying role types at the end of UML associations and *fills* relations between role types and UML classes. Figure 4.1 shows the corresponding graphical representation of the banking example using the revised UML notation [Steimann, 2000c]. It depicts

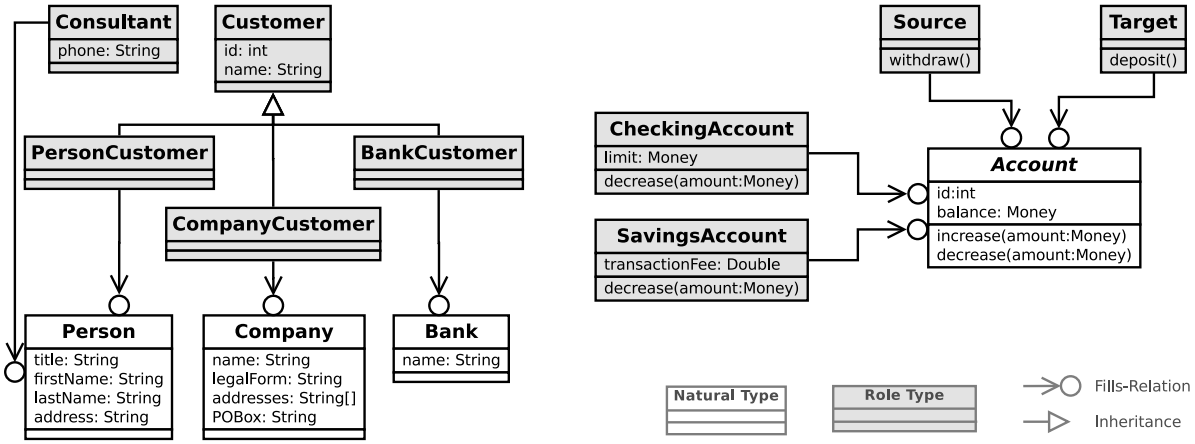

Figure 4.2: Bank example depicted using the Generic Role Model.

role types as circles and the *fills* relation with dashed arrows. However, due to the missing operational semantics and missing tool support for its graphical notation, its influence stayed behind the accompanied list of features of roles. Unsurprising, Steimann already applied the first 15 features of roles to classify *Lodwick* [Steimann, 2000b, Sec. 4.2]. In short, *Lodwick* supports most of these features either fully (Features 1–5, 7, 14) or partially (Features 6, 9–11, 13). Conversely, it only lacks the support for roles playing roles (Feature 8) and roles carrying an identity (Feature 15). Regarding the additional features of roles, *Lodwick* permits the specification of *intra-relationship constraints* (Feature 16). Besides that, because relations can be n-ary, it is possible to model compartments as relations (Feature 19). Similarly, due to the existence of *role inheritance*, a role type can be part of multiple relationships (Feature 21) by deriving a subrole for each new relation. Specifically, this was utilized to specify the various relationships of the `Customer` role type. In sum, *Lodwick* provides a very concise formalization of the behavioral and relational nature of roles that can be quickly adapted to different domains, e.g. *UML* [Steimann, 2000c] or *MAS* [Boella and Van Der Torre, 2007].

4.1.2 THE GENERIC ROLE MODEL

In contrast to *Lodwick*, the **Generic Role Model** proposed by Dahchour et al. [2002] embraces the behavioral nature of roles. In short, they introduce *role relationships* as new inheritance relation. Similar to *object inheritance*, the *role relationship* permits dynamically adapting classes, instantiating multiple classes, and context-dependent access [Dahchour et al., 2002]. In essence, *role relationships* correspond to the *fills* relationship and model that a given class can play the role represented by the subclass. However, this approach mingles the inheritance hierarchy of both natural types and role types, as both are actually classes. Role instances, in turn, are represented as adjunct objects to their respective players that carry their own unique identity [Dahchour et al., 2002]. Nevertheless, their biggest contribution is the formal description of the *fills* relation and its interaction with class-based inheritance. According to Dahchour et al., *"[each] instance of a role class (e.g., Student) is related to exactly one instance of its object class (e.g., Person) but, unlike generalization, each instance of the object class can be related to any number of instances of the role class"* [Dahchour et al., 2002, p.649]. By extension, they establish that objects can dynamically change their super class, if it is specified with a *role relationship*. As an illustration, Figure 4.2 shows the *Generic Role Model* of the banking scenario. In this model both the role types and natural types are represented as classes. Role types, however, are those classes from which a *role relationship* starts, as for instance `Consultant`, `Source`, and `Target`.

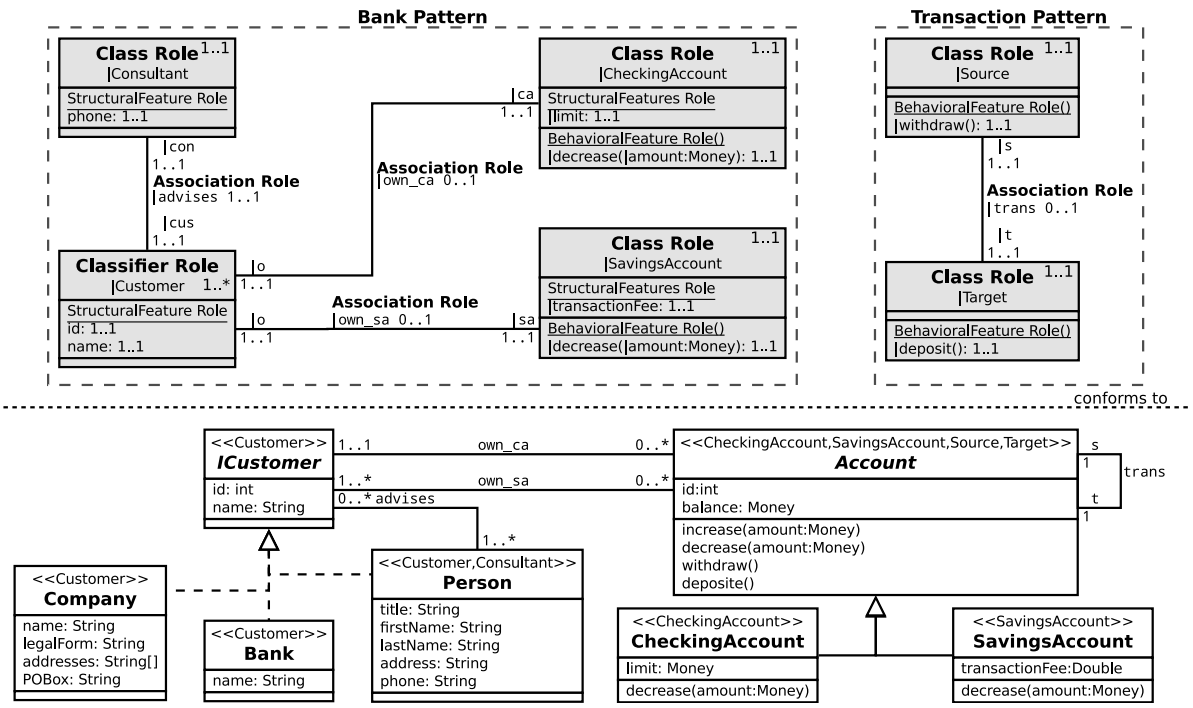

Figure 4.3: Bank example specified as two patterns using the RBML.

Although it appears that the *role relationship* correspond to the *fills* relation, its definition does not support that role types can be played by unrelated objects (Feature 7) [Dahchour et al., 2002, p.649]. Hence, the example model must contain individual subroles for each of the unrelated players of the `Customer` role type to capture the intended semantics of the banking scenario. In conclusion, the *Generic Role Model* embraces the behavioral nature of roles and supports all but four of the initial 15 features of roles, i.e.: Feature 2, 7, 9, 14, but none of the additional features of roles (Feature 16–27).

4.1.3 ROLE-BASED METAMODELING LANGUAGE (RBML)

While the previous approaches employed roles to dynamically extend objects, the **Role-Based Metamodeling Language (RBML)** introduces roles on the metamodel level to vary and extend model elements [France et al., 2004]. Simply put, Kim et al. *"define roles at the metamodel level to specify design patterns where a role is played by model elements (e.g., classes, associations)"* [Kim et al., 2003, p.1]. These roles, denoted *model roles*, are played by UML model elements, e.g.: classes, associations, attributes and methods [Kim et al., 2002]. Conversely, role types are specified on the metamodel level, such that each role type can be played by a corresponding metaclass in the UML metamodel [Rumbaugh et al., 1999]. *Classifier roles*, for instance, can be played by any `Classifier`, *class roles* by any `Class`, *association roles* by any `Association`, and so forth. In particular, *classifier roles* additionally define sets of structural and behavioral features, i.e., *structural feature roles* and *behavioral feature roles*. As a result, *model roles* give rise to instances of the corresponding metaclasses featuring the specified properties, behaviors and constraints. Moreover, they can be played by multiple instances of a metaclass, however, each *model role* is specifically tied to one metaclass as its player type [Kim et al., 2002]. Finally, all *model roles* are collected into a *Role Hierarchy*, which then can be used to verify if a given UML class diagram conforms to the given hierarchy, i.e., whether the class diagram fulfills the structural constraints specified in the *role hierarchy* and

the *model roles* [Kim and Shen, 2007]. As an illustration, Figure 4.3 specifies both the `Bank` and the `Transaction` as *role hierarchies*. In detail, the role types `|Customer`, `|Source`, and `|Target` are defined as *class roles* to ensure that their methods are implemented in an UML class. In contrast, the *customer* role is modeled as a *classifier role* permitting its player to be also an interface or abstract class. Similarly, the relationships `|advises`, `|own_ca`, `|own_sa`, and `|trans` are specified as *association roles* accompanied by individual *association end roles*. In the conforming UML class diagram (below the dashed line), the various classes fulfill the various *model roles*. The `Account` class, for instance, plays the roles `|CheckingAccount`, `|SavingsAccount`, `|Source`, and `|Target`; whereas the classifier role `|Customer` is played by the classes `Person`, `Company`, and `Bank` as well as the equally named interface. Actually, the occurrence constraint `1..*` of the `|Customer` role not only states that it must be played at least once. As a result, the depicted class diagram implements the banking model specified as a *design pattern* on the metamodel level. Regardless of the missing dynamics of roles, the RBML is a well-founded modeling language focusing on the behavioral and relational nature of roles. Conversely, while those features of roles that assume dynamic binding of roles are not applicable (i.e. Features 5, 6, 9, and 10), the behavioral features of roles are fully supported, i.e., Features 1–4 and 12–14. Additionally, it allowed for specifying *intra-relationship constraints* within *association roles* (Feature 16) and was one of the first RMLs featuring *occurrence constraints* for roles (Feature 27). In conclusion, this RBML is best suited for the specification and validation of *Design Patterns* [Gamma et al., 1994] including both structural design patterns [Kim et al., 2002, 2003, France et al., 2004, Kim and Shen, 2007] as well as behavioral design patterns [Kim and Lu, 2008, Kim and Lee, 2015]. Additionally, Kim [2008] showed the applicability of the RBML to pattern-based model refactoring. Ultimately, its ongoing development led to the implementation of the `RBML Conformance Checker` within the *TrueRefactor* software engineering tool suite.[1]

4.1.4 ROLE-BASED PATTERN SPECIFICATION

In the same way as RBML, the *Role-based Pattern Specification* proposed by Kim and Carrington [2004] facilitates roles on the metamodel level. However, in contrast to RBML, their approach *"defines an innovative framework where generic pattern concepts based on roles are precisely defined as a formal role metamodel using Object-Z"* [Kim and Carrington, 2009, p.243]. Basically, they employ the *Object-Z specification language* [Smith, 2012] to formalize both the employed role concept and the individual design patterns.[2] Nonetheless, like RBML, the roles of a design pattern are specified on the metamodel level to fill one of the corresponding metaclasses of *UML class diagrams*, e.g. `Classes`, `Operations`, and `Associations`. In particular, both the *class model* and the *role model* are specified separately [Kim and Carrington, 2009]. Afterwards, a *role binding model* specifies which *model elements* of the *class model* play role instances of the *role model*. In addition, the *role binding model* contains six constraints to ensure that the given *class model* fulfills the structural and behavioral constraints imposed by the *role model*. These constraints, for instance, ensure that each *class role* is bound to `Class` in accordance to its *occurrence constraint*, such that each of its *attribute roles* and *operation roles* are bound consistently to the `Class`'s corresponding features [Kim and Carrington, 2009]. Furthermore, the operations required by a *class role* can be specified as *operation role dependency* to its playing class [Kim and Carrington, 2009]. Due to the fact that the formal notation is rather long, Figure 4.4 depicts the financial transaction modeled as an *UML object diagram* to enhance the readability, as suggested by Kim and Carrington [2009]. In detail, the diagram specifies two `ClassRoles` named `Source` and `Target`, as well as their cor-

[1] `https://app.assembla.com/wiki/show/truerefactor/TrueRefactor`

[2] *Object-Z* is an object oriented extension to the *Z notation* [Spivey, 1998] suitable for the formalization of UML class diagrams [Kim and David, 1999].

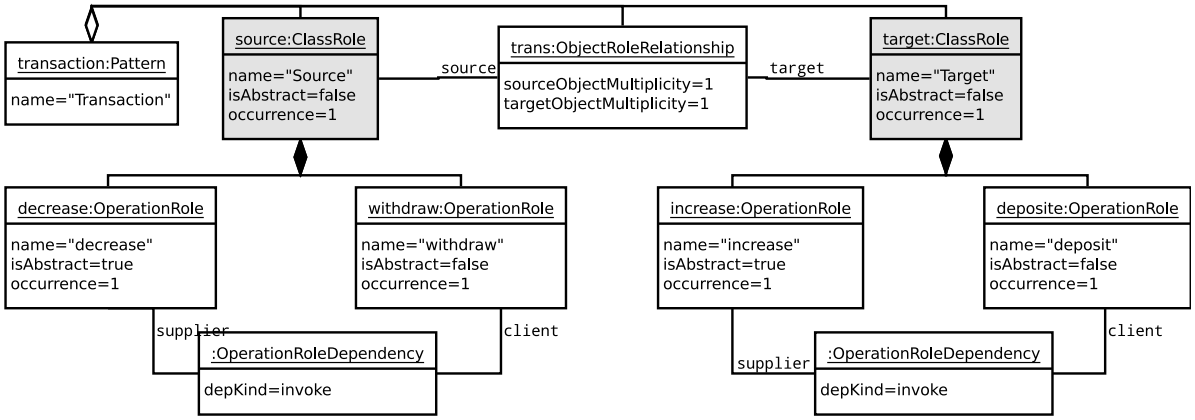

Figure 4.4: Financial transaction specified with the Role-based Pattern Specialization.

responding `OperationRoles` for the methods `withdraw` and `deposit`, respectively. Additionally, this model contains an `ObjectRoleRelationship` to declare the `trans` relationship, as well as, two `OperationRoleDependency` specifying that withdrawals and deposits invoke the `decrease` and `increase` method of the respective player of the `ClassRole`. Finally, the various *occurrence constraints* declared for each element of the *role model* define how often each role can be played. In conclusion, then Kim and Carrington designed the *Role-based Pattern Specification*, as a formal role-based metamodeling language focusing on the behavioral and relational nature. However, because they do not consider dynamic role binding, non of the dynamic features of roles could be applied, i.e. Features 5, 9, and 12. Conversely, the *Role-based Pattern Specification* fully supports the Features 1–3, 11, 13, and 14 facilitating roles as entities with behaviors that can be linked with relationships and can inherit from one another, yet have no individual identity. Furthermore, it incorporates the specification of *role constraints* (Feature 6), *occurrence constraints* (Feature 16) and *intra-relationship constraints* (Feature 27). In sum, the *Role-based Pattern Specification* is able to fully formalize the role-based metamodeling approach based on a well-founded formal theory. However, a design pattern defined with the *Role-based Pattern Specification* can only be applied once in a *class model*, as it does not provide compartments to distinguish different application contexts of individual design patterns. For instance, a *composite pattern* [Gamma et al., 1994] that occurs twice in the class model of a graphical modeling editor can only be bound once.

4.1.5 OBJECT-ROLE MODELING (ORM) 2

Object-Role Modeling (ORM) [Halpin, 2009] is a well-established data modeling methodology. It is an advancement of the *Natural language Information Analysis Method* (NIAM) into a fully fledged data modeling language [Halpin, 1998] supported by an expressive query language [Bloesch and Halpin, 1997] that is updated to *ORM 2*. Halpin [1998] elucidates that *"ORM is so-called because it pictures the world in terms of objects (entities or values) that play roles (parts in relationships)"* [Halpin, 1998, p. 1]. In making this comment, Halpin emphasizes that *ORM* is fact-oriented, as it treats both entities and values just as facts. Moreover, he establishes roles as parts of relationships played by entities or values. Regardless, roles degenerate to unnamed ends of relationships filled by *entity* and *value types* [Halpin, 1998] without properties. Despite that, *ORM* provide numerous available constraints for these relationships including *intra-* and *inter-relationship constraints* [Halpin, 2009]. On the downside, it did not encompass the possible flexibility provided by other role-based approaches. This becomes evident when modeling the banking example with ORM 2, as shown

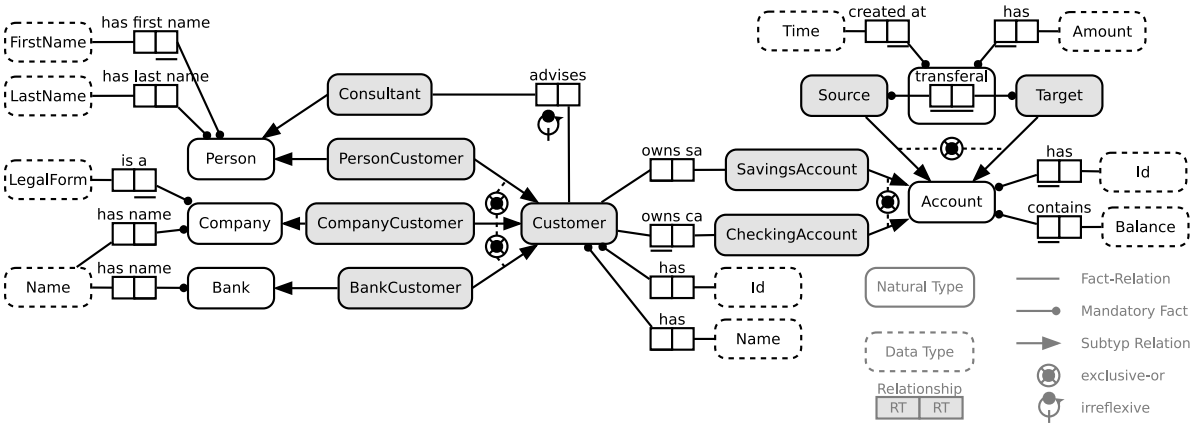

Figure 4.5: Bank example modeled with ORM 2.

in Figure 4.5. In particular, neither the role types *customer* and *consultant* nor *savings account*, *checking account*, *source*, and *target* can be represented as roles in *ORM*, because their individual attributes and relationships would be hidden. Consequently, these role types must be modeled as entity types, as well. Then, however, role types and their player types must be related with generalization or specialization, which, in turn, cannot capture the semantics of roles, according to [Steimann, 2000b]. In response, Halpin [2007] proposed to use a small design pattern for modeling role types (cf. [Halpin, 2007, Fig. 11]). For example, the model contains `PersonCustomer`, `CompanyCustomer`, and `BankCustomer` as subtype of both `Customer` and the respective player type. Additionally, the individual subtype relations of the `Customer` entity type must be defined as complete disjoint, as indicated with the *exclusive or* between the corresponding subtype relations. Similar to `Customer`, all the role types of the `Account` entity types are modeled as subtypes. Moreover, the different relationships could be modeled appropriately including the various cardinalities and the intra-relationship constraint `Irreflexive` for `advises`. Even though *ORM* does not support *compartments*, it still allows for objectifying relationships. In this way, individual *transactions* can be represented as a `transferal` entity type with individual properties [Halpin, 2009]. Unfortunately, *objectification* is limited to a single relationship. Hence, the *bank* cannot be modeled as an objectified relationship. In sum, *ORM* is not only one of the oldest RMLs, but also a language fulfilling a considerable amount of the features of roles. In detail, it fully supports Features 2–5, 14, 16, and 17 establishing roles dependent on relationships such that entities play roles whenever entity or value are related to one another. Moreover, relationships feature a wide variety of *intra-* and *inter-relationship constraints* (Feature 17). Still, roles cannot have individual properties (Feature 1) and cannot be played by unrelated entities (Feature 7). However, if role types are modeled as an *entity type*, their properties can be modeled as facts and they can be played by unrelated entities using subtyping [Halpin, 2007]. While this facilitates the relational nature of roles, objectified relationships can partially establish context-dependent roles. Accordingly, Features 19 and 20 are only partially fulfilled, as objectified relationships are limited to single relationships. Regardless, they can play roles themselves (Feature 22) and as entities have their own identity (Feature 26). However, because *ORM* is a data modeling language, the dynamic features of roles, e.g. Feature 9, 10, and 12 are not applicable. In conclusion, *ORM 2* is a well-engineered data modeling language fully incorporating the relational nature and various constraints of roles. Furthermore, Halpin et al. implemented the graphical editor *NORMA* integrated into *Microsoft Visual Studio* [Halpin, 2005], as well as the query engine *Conquer–II* to query ORM data models [Bloesch and Halpin, 1997].

4.1.6 ONTOUML

OntoUML [Guizzardi and Wagner, 2012] is an ontologically founded conceptual modeling language developed by Guizzardi et al. [2004] to overcome the syntactical and semantic shortcomings of UML. Therefor, they developed a multi-layered Universal Foundational Ontology (UFO) [Guizzardi and Wagner, 2005] that contains not only *kinds* and *role types* but also *role mixins* and *relators* together with rules to specify well-formed models. These types can be used within *OntoUML* to annotate UML classes with stereotypes and the inheritance relation with additional constraints [Guizzardi and Wagner, 2012]. Surprisingly, they have two distinct notions of roles. Role types are *sortals* inheriting their identity and role mixins are mixin types without any identity. Both are necessary to model role types that can be played by unrelated types [Guizzardi et al., 2004, Fig. 5] by employing a design pattern not unlike the pattern proposed for *ORM 2* in [Halpin, 2007, Fig. 11]. Along the same lines, *OntoUML* introduces the concept of *relators* to specify objectified relationships. In particular, Guarino and Guizzardi [2015] define that *"when a relation R is* derived *from a relator type T, then, for every x, y, $R(x,y)$ holds if there is an instance t of T such that $mediates(t,x)$ and $mediates(t,y)$ hold."* [Guarino and Guizzardi, 2015, p.283]. Basically, they describe that a *relator* instance references all the related instances using the *mediates* association. In contrast to ORM, however, *OntoUML* only supports binary relationships and hence can only model *relator types* as objectification of binary relationships. Furthermore, *relators* cannot be used as superclass of a *role type* and as such cannot play roles themselves. As a result, while *relator types* can be used to objectify binary relations, it cannot be used to specify an objectified collaboration. In conclusion, when specifying the financial application using *OntoUML*, shown in Figure 4.6, natural types are annotated with the `<<kind>>` stereotype, role types with `<<role>>`, and relator types with `<<relator>>`. Moreover, the `Transaction` and the `Bank` *relator type* can only mediate the `trans` and the `advises` relation, respectively. Thus, both the `own_ca` and the `own_sa` relationships are not mediated by (or dependent on) the `Bank` relator type. Moreover, as indicated earlier, *OntoUML* requires a design pattern to model that the natural types `Person`, `Company`, and `Bank` can play the `Customer` role. Consequently, the entity `Customer` is declared as a *role mixin* and thus, as abstract super type for the concrete, disjoint role types `PersonCustomer`, `CompanyCustomer`, and `BankCustomer`. Notably though, *OntoUML* prohibits that *kinds* inherits from *relators* and vice versa. As a result, a *bank* must be modeled with two different conceptual entities that are conceptually the same. It follows, then that *OntoUML* is an ontologically founded RML incorporating the relational and behavioral nature of roles. In detail, it fully supports Feature 1, 2, 8, 10, 11, 14 and partially Features 3, 7, 13. On the one hand, roles are classified to have properties and behavior, may depend on relationships, can play other roles (using multiple inheritance), and share their identity with the player type. On the other hand, objects can play multiple roles only if overlapping subtyping is utilized. Similarly, unrelated objects can play the same role only if a special design pattern is used. In addition to that, *OntoUML* provides both *intra-* and *inter-relationship constraints* (Feature 16 and 17), as well as *occurrence constraints* that can be specified at the *mediation* relation of *relator types* (Feature 27). Finally, *relator types* can model *compartment types* in *OntoUML* (Feature 19), as they have properties and behaviors (Feature 20) and permit that a *role type* can be mediated by multiple *relator types* (Feature 21). In sum, the modeling language focuses on the creation of ontologically founded conceptual models, however, is not concerned with their realization on the instance level. Hence, the dynamic features of roles are not applicable, i.e. Feature 4, 5, 9, and 12. In conclusion, while *OntoUML* is a very expressive extension to UML introducing multiple new concepts, it does not embrace the context-dependent nature of roles. Nonetheless, Benevides and Guizzardi [2009] provided a proof-of-concept modeling editor for *OntoUML* that is now available on *GitHub*[3].

[3] `https://github.com/nemo-ufes/ontouml-lightweight-editor`

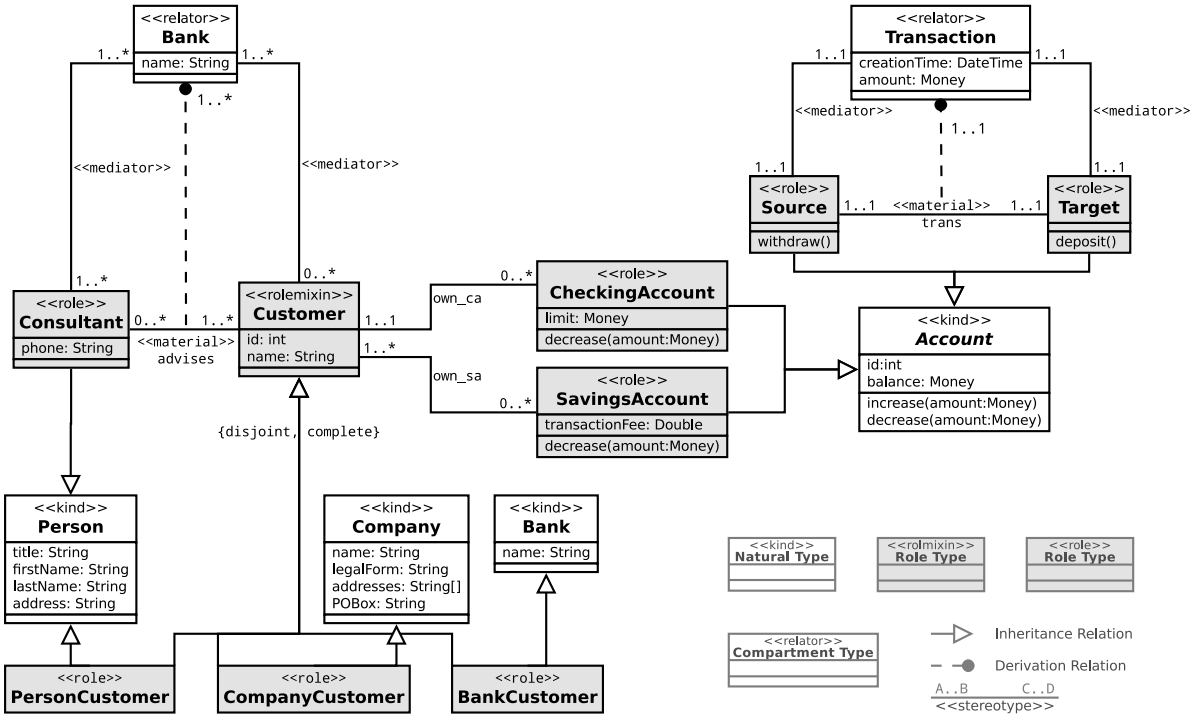

Figure 4.6: Bank example represented in OntoUML.

4.2 CONTEXT-DEPENDENT MODELING LANGUAGES

After the previous approaches only incorporated the behavioral and relational nature of roles, this section elaborates on RMLs that combine the context-dependent and behavioral nature of roles, yet do not feature the relational nature. Moreover, these approaches represent more formal modeling languages that do not provide a graphical notation. In short, the *Metamodel for Roles* [Genovese, 2007] is a formal metamodel for roles proposed to unify the various RPLs (Section 4.2.1). The *E-CARGO Model* [Zhu and Zhou, 2006] is a formal role-based model for computer-supported cooperative work (CSCW) (Section 4.2.2). In contrast to these formal models, the *DCI* approach [Reenskaug and Coplien, 2009] is a general paradigm for the role-based design of software systems (Section 4.2.3). In sum, the section henceforth highlights the different attempts to reconcile the behavioral and context-dependent nature of roles.

4.2.1 METAMODEL FOR ROLES

Similar to *Lodwick*, the **Metamodel for Roles** was proposed by Genovese [2007] in an attempt *"to provide a flexible formal model for roles, which is able to catch the basic primitives behind the different role's accounts in the literature, rather than a definition"* [Genovese, 2007, p.27–28]. In other words, Genovese attempts to reconcile the various definitions of roles by providing a general formal model for roles able to encompass the preceding RPLs. In particular, he introduces *Players*, *Roles* and *Institutions* both on the model (M1) and the instance level (M0) [Genovese, 2007]. All these entities can have attributes, operations and separate inheritance hierarchies. Moreover, the metamodel includes the relation *RO* and *PL* representing which roles belongs to which institution and which player can play them, respectively [Genovese, 2007].

Listing 4.2: Bank example formalized using the Metamodel for Roles.

```
1  < D, Contexts, Players, Roles, Attr, Op, Constraints, PL, RO, AS, OS, RH, PH, CH >
2  D := {Bank, Transaction, Person, Company, Bank, Account, Customer, ...}
3  Contexts := {Bank, Transaction}
4  Players := {Person, Company, Bank, Account}
5  Roles := {Customer, Consultant, CA, SA, Source, Target, MoneyTransfer}
6  Attr := {name, firstName, lastName, phone, id, limit, transactionFee, balance, ...}
7  Op := {increase, decrease, withdraw, deposit, execute}
8  Constraints := {...}
9  PL := {(Person, Consultant), (Person, Customer), (Company, Customer), (Account, Source),
10     (Account, Target), (Account, CA), (Account, SA), (Transaction, MoneyTransfer)}
11 RO := {(Bank, Consultant), (Bank, Customer), (Bank, CA), (Bank, SA), (Bank, MoneyTransfer),
12     (Transaction, Source), (Transaction, Target)}
13 AS := {(Bank, name), (Transaction, amount), (Person, firstName), ...}
14 OS := {(Account, increase), (Account, decrease), (Source, withdraw), ...}
15 RH := ∅
16 PH := ∅
17 CH := ∅
```

The only novel concept are *Sessions* that specify the binding of attributes when roles collaborate with one another in an institution. On the downside, all the entities are collected from the same set and thus allow both roles and institutions to play roles implicitly. Besides that, the only possibility to adjust the metamodel to a target language is to specify additional constraints to both model and instance level as logical formulae [Genovese, 2007], e.g., $Players \cup Roles \neq \emptyset$, to ensure that the sets of players and roles are disjoint. Yet without these constraints, the formal model would permit arbitrary combinations of types and interrelations, e.g., a person that is both a player and a role and inherits from another institution. Nevertheless, when considering the banking example, the resulting formal model appropriately captures the behavioral and context-dependent aspects of the banking domain, as shown in Listing 4.2. In general, the model is specified as a tuple of sets of classes and relations between them, whereas D contains all classes of the domain, $Attr$ all their attributes, and Op all their methods. Furthermore, the sets $Contexts$ and $Roles$ collect all institution types *(e.g. $Bank$, $Transaction$)* and role types *(e.g. $Customer$, $Consultant$, etc.)*, respectively. In contrast, the $Players$ set contains all classes able to play roles, such as $Person$, $Company$, $Account$, $Transaction$ and $Bank$. After declaring all classes, the PL relation collects which player type fills which role type, e.g., that $Person$ player type fills the $Customer$ and the $Consultant$ role type. Similarly, the RO relation defines which role types participate in which institution, such that the $Bank$ contains the $Consultant$, $Customer$, SA, and CA.[4] Finally, because the banking scenario does not feature inheritance, the corresponding inheritance relations PH, RH, and CH are left empty. As a result, the model appropriately captures the context-dependent behavior in the banking domain, yet lacks the notion of relationships. Conversely, the *Metamodel for Roles* fully embraces the context-dependent and behavioral nature of roles. Moreover, it is designed to be tailored towards different variants of the role concept.

Hence, the evaluation focuses on the most general variant representable within the *Metamodel for Roles*, i.e., an unconstrained formal model. Accordingly, the most general variant fully supports the behavioral Features 1,4–5,7,8, and 13 stating that role types have attributes and operations, can be played by multiple unrelated objects, and can inherit properties from other role types. Furthermore, Features 9–11, 14, 15 are only partially supported, as additional constraints are required to specify the desired semantics [Genovese, 2007], e.g., to decide whether role instances have their own (Feature 15) or a shared identity (Feature 14). Besides that, the fact that an object can play

[4] For brevity, savings accounts and checking accounts are abbreviated to *SA* and *CA*, respectively.

multiple roles simultaneously (Feature 3) must be modeled using *sessions*, which can link different role instances by means of their attributes and behaviors. Moreover, Genovese [2007] argued that *sessions* can also be used to represent links of relationships, hence partially fulfilling Feature 2. In contrast, most of the features of roles corresponding to compartments are fulfilled by the *Metamodel for Roles*, i.e. Features 19-23, and 25 are fully supported, whereas Features 26 and 27 are partially supported. On the one hand, Genovese's institution types have attributes and operations, can play roles, and can inherit properties from other institution types. On the other hand, additional constraints determine whether institutions have their own or a session-based identity and can be employed to specify *occurrence constraints* as logical formula. In conclusion, the *Metamodel for Roles* is designed to capture most of the features of roles. However, I would argue, that it is to general to be useful, as it leaves the various assumptions and design decisions of the different definitions of roles unspecified.

4.2.2 E-CARGO MODEL

The **E-CARGO Model** [Zhu, 2005] is another formal role-based model for computer-supported cooperative work (CSCW). In particular, Zhu and Zhou emphasize that *"roles are the key media for human users to interact and collaborate"* [Zhu and Zhou, 2006, p.580]. Hence, to facilitate the role concept for the specification and design of CSCW systems, they argue that a formal model for role-based CSCW is imperative, because *"[it] supports the robustness, efficiency, and correctness of the entire system."* [Zhu and Zhou, 2006, p.581]. Consequently, the *E-CARGO Model* encompasses *agents* playing *roles* collaborating in *groups* working on *objects* in a defined *environment* [Zhu and Zhou, 2006].[5] In particular, it provides two different representations of compartments: groups and environments. Groups are used to arrange collaborating agents by first negotiating the roles assumed by their players [Zhu and Zhou, 2006]. Environments, in turn, specifies the work space of several groups and limits the number of roles playable simultaneously in the groups [Zhu and Zhou, 2006, Appendix]. Thus, groups can be seen as collaborations and environments as their instantiation, which, in turn, coincides with the notion of compartments. More recently, the formal model provided the foundation for a minimal role-playing logic [Liu et al., 2014], dynamic role assignment for adaptive collaborations [Sheng et al., 2016], and a practical approach to resolve conflicting role assignments [Zhu, 2016]. In conclusion, the *E-CARGO Model* proves to be one of the more successful role-based models. Although the target domain is cooperative work, their underlying formal model is applicable to role-based software systems, as well. Admittedly, in contrast to typical modeling languages, the formal model mixes both the type and instance level.

Consequently, only the natural types in the banking domain are defined as classes and collected in C. The $account$ class, for instance, is identified as $Account$, has two attributes id and $balance$, as well as implements at least the methods $increase$, and $decrease$ and exposes them as interface. Moreover, objects and agents are instances of a class collected in the sets O and A, respectively. However, because only agents can play roles, accounts, persons and companies must be modeled as agents. Role types, in turn, are collected in R and specified as a tuple encompassing its identifier, incoming and outgoing messages, the set of agents currently playing this role type, and the set of classes, roles, and objects these players can access. Consider the definition of the $consultant$ role type, which adds the outgoing message $advise$ to its players and permits them access to $customer$ role types and the $bank$ environment. Additionally, the definition collects all the agents currently playing this role in its third component, e.g., person p_1.

[5] E-CARGO is an acronym for Environment, Classes, Agents Roles Groups and Objects.

Listing 4.3: Bank example defined using the E-CARGO model.

$\sum = \langle C, O, A, M, R, E, G, s_0, H \rangle$
$C := \{account, person, company\}$
$O := \{amount, phone, \ldots\}$ /* Object Instances */
$A := \{a_1, a_2, \ldots, p_1, p_2, \ldots, c_1, c_2, \ldots\}$ /* Agent Instances */
$M := \{increase, decrease, withdraw, deposit, advise, \ldots\}$ /* Messages */
$R := \{consultant, customer, ca, sa, source, target, moneytransfer\}$ /* Role Types */
$E := \{bank, transaction\}$
$G := \{bank_1, \ldots, t_1, \ldots\}$
$H := \{\ldots\}$ /* Users */
/* Natural Types */
$account := \langle Account, \{id, balance\}, \{increase, decrease, \ldots\}, \{increase, decrease, \ldots\}\rangle$
$person := \langle Person, \{firstName, lastName, \ldots\}, \ldots, \ldots\rangle$
$company := \langle Company, \{name, legalForm, \ldots\}, \ldots, \ldots\rangle$
/* Role Types */
$consultant := \langle Consultant, \langle \ldots, \{advise\}\rangle, \{p_1, \ldots\}, \{customer, bank\}\rangle$
$customer := \langle Customer, \langle \{advise, \ldots\}, \{increase, decrease, \ldots\}\rangle, \{p_1, \ldots, c_1, \ldots\}, \{consultant, sa, ca\}\rangle$
$ca := \langle CeckingAccount, \langle \{increase, decrease\}, \emptyset\rangle, \{a_2, \ldots\}, \{bank\}\rangle$
$sa := \langle SavingsAccount, \langle \{increase, decrease\}, \emptyset\rangle, \{a_1, \ldots\}, \{bank\}\rangle$
$source := \langle Source, \langle \{withdraw\}, \emptyset\rangle, \{a_1, \ldots\}, \{amount\}\rangle$
$target := \langle Target, \langle \{deposit\}, \emptyset\rangle, \{a_2, \ldots\}, \{amount\}\rangle$
/* Environments */
$bank := \langle Bank, \{\langle consultant, [1, *], \{phone\}\rangle, \langle costumer, [0, *], \ldots\rangle, \langle ca, [0, *], \ldots\rangle, \langle sa, [0, *], \ldots\rangle\}\rangle$
$transaction = \langle Transaction, \{\langle source, [1, 1], \ldots\rangle, \langle Target, [1, 1], \ldots\rangle\}\rangle$
/* Groups */
$bank_1 = \langle AlphaBank, bank, \{\langle p_1, consultant, phone\rangle, \langle p_1, customer, \ldots\rangle \ldots\}\rangle$
$t_1 = \langle t042, transaction, \{\langle a_1, source, amount\rangle, \langle a_2, target, amount\rangle\}\rangle$

In contrast to role types, the $bank$ and the $transaction$ environments represent the type level definition of compartments whereas the individual groups, such as t_1 and $bank_1$, represent the corresponding collaborations instances. In particular, the definition of the $bank$ environment specifies that at least one agent plays the $consultant$ role and that consultants have access to a $phone$ object. The group $bank_1$ declares the $AlphaBank$ instance where the person p_1 plays the consultant and customer role. In conclusion, the modeled banking application σ represents both the domain model and a possible instance in one model. Nonetheless, the *E-CARGO Model* combines the behavioral and context-dependent nature of roles into a concise formal modeling language. Basically, it fully supports the behavioral Features 3–5, 9, 12, 14 and partially the Feature 1, 6, 7, 10, 15. On the one hand, agents can play multiple roles simultaneously, acquire and abandon roles dynamically and they share their identity with their roles. Moreover, a role restricts the environments, agents, objects, and roles the corresponding agent can access. On the other hand, roles do not have properties and behaviors, however they extend the messages an agent can send and respond to, as well as externalizes its state into an object. Furthermore, unrelated agents can only play the same role type, if their interface provides the same messages as the role type specifies as incoming. Last but not least, it is possible to specify all roles as role types duplicating their specification, then each role instance has its own identity. Besides that, *E-CARGO* introduces both environments and groups as representations of compartments and thus, supports the Features 19, 21, 26, and 27 fully. This entails, that role instances depend on groups, their occurrence can be constrained, and groups depend on environments and carry their own identity. In contrast to these features, only Feature 18 is partly fulfilled, as groups allow for combining roles and constrain them together, yet only with *occurrence constraints*. In sum, the *E-CARGO Model* is a very concise formal model for behavioral and context-dependent roles that proofed to be a valuable basis for multiple contributions to the design of role-based systems, e.g. [Zhu, 2007, Zhu and Zhou, 2008a, Liu et al., 2014, Sheng et al., 2016, Zhu, 2016].

Listing 4.4: Bank example implemented with Scala DCI.

```
case class Person(title: String, firstName: String, /*...*/)
case class Company(POBox: String, addresses: String, /*...*/)
case class Account(var balance: Money, id: Integer){/*...*/}
@context
case class Transaction(source:Account,target:Account,amount: Money){
  role source{
    def withdraw(amount:Money):Unit={source.decrease(amount)} }
  role target{
    def deposite(amount:Money):Unit={target.increase(amount)} }
  def execute():Boolean={source.withdraw(amount);/*...*/}
}
@context
case class Bank(name: String){
  var customer = mutable.ListBuffer.empty[customer] /*...*/
  role consultant{/*...*/}        role customer{/*...*/}
  role checkingsAccount{/*...*/}  role moneyTransfer{/*...*/}
  role savingsAccount{/*...*/
    def decrease(amount:Money):Unit=
      {savingsAccount.decrease(amount*transactionFee)} }
}
```

4.2.3 DATA CONTEXT INTERACTION (DCI)

In contrast to the previous approaches, **Data Context Interaction (DCI)** is a new paradigm brought forward by Trygve Reenskaug to overcome the limitations of object-oriented design methodologies [Reenskaug and Coplien, 2009]. To put it bluntly, Reenskaug and Coplien argue that *"[w]hile objects capture structure well, they fail to capture system action. DCI is a vision to capture the end user cognitive model of roles and interactions between them."* [Reenskaug and Coplien, 2009]. In other words, they suggest changing the perspective of software development from objects with data and behavior towards *data* which plays *roles* in *interactions* dynamically connected by a *context* [Reenskaug and Coplien, 2009]. In particular, objects degrade to data containers and the behavior is specified in the roles defined in a specific context. The context itself manages and binds the role instances to data objects and controls their interaction [Reenskaug and Coplien, 2009]. By employing this paradigm, Coplien and Bjørnvig [2010] argues, that software developments becomes lean, i.e., less wasteful, because the *DCI architecture* allows for deferring implementations and more closely resembles the end user's mental model [Coplien and Bjørnvig, 2010]. They describe an agile software development process that captures the structure of the domain in data classes, whereas the behavior of algorithms and use cases are implemented in context classes containing the participating role mixins (or traits). Although Reenskaug and Coplien [2009] introduced a simple graphical notation highlighting classes, objects, methodful roles, methodless roles and contexts [Reenskaug and Coplien, 2009, Fig. 5], their notation only serves as an illustration of the underlying architecture. Despite the lack of a modeling language, both Reenskaug and Coplien [2009] and Coplien and Bjørnvig [2010] show that *DCI* is readily applicable employing state-of-the-art object-oriented programming languages [Reenskaug, 2011], such as Scala, Python, C#, Ruby and Qi4j [Coplien and Bjørnvig, 2010].[6] Consequently, to model the banking domain using the *DCI* architecture, it was implemented utilizing *Scala DCI*,[7] outlined in Listing 4.4. Particularly, the data elements are implemented as `case` classes, i.e., `Account`, `Person`, and `Company`. Conversely, both the transferal and banking use case are implemented as the context classes `Transaction` and `Bank`, respectively.

[6] Qi4J was a Java framework for DCI now embedded in the *Apache Zest*™ composite oriented programming framework.
[7] `https://github.com/DCI/scaladci`

The interacting roles in these use cases are specified using the `role` keyword and are implicitly bound to their player when the corresponding context is instantiated. The context object ultimately manages their players either as a field, in the case of the `Transaction`, or lists, in case of the `Bank`. As a result, this appropriately implements the behavioral and context-dependent aspects of the banking application. Nonetheless, the features of roles supported by the various implementations greatly vary. Hence, the evaluation only takes its proposal [Reenskaug and Coplien, 2009] and implementation guideline [Reenskaug, 2011] into account. Henceforth, features are considered partially fulfilled, if they depend on the implementation rather than the DCI paradigm. In general, roles in *DCI* fulfill Feature 1,3,7, 10–12 completely. Accordingly, roles have their own properties and behaviors that can be assumed by unrelated objects and used to restrict access to the players state. Moreover, objects can play multiple role types simultaneously and used to employ access restrictions. While these features where described in [Reenskaug and Coplien, 2009, Reenskaug, 2011], the fact that roles can be dynamically assigned, can inherit from another one, and have either a shared or unique identity depend on the implementation of the *DCI architecture*. Thus, Feature 2, 5, 13–15 are considered partially fulfilled. In addition, the context classes in *DCI* directly correspond to compartment types. In turn, the *DCI* paradigm satisfies Feature 19, 20, 22, and 26. Like objects, contexts can have properties and behavior, can play roles, and have their own identity, as well. Last but not least, whether context classes can inherit from another one (Feature 25) is again implementation dependent. In sum, *DCI* propagates a paradigm shift from object-orientation to role-orientation by combining the behavioral and context-dependent nature of roles. Furthermore, the underlying architecture is so simple that it spanned multiple implementations in various programming languages.

4.3 COMBINED MODELING LANGUAGES

Up to this point, none of the aforementioned RMLs have combined the behavioral, relational and context-dependent natures of roles. Therefore, this section highlights those three modeling languages that accomplished to incorporate all of them. The *Taming Agents and Objects (TAO) Approach* is one of the first modeling languages for the specification of MAS that augments environments by adding relationships (Section 4.3.1). In contrast to TAO, the *Information Networking Model (INM)* is a data modeling language that introduces contexts to a relationship-based model to model context-dependent information (Section 4.3.2). Most recently, the *HELENA approach* proposes a role-based architecture description language that adds both roles and their relationships to an ensemble-based modeling language (Section 4.3.3). Even though these approaches aim at different domains, they all combine the natures of roles to appropriately model the intrinsic context-dependence and dynamics of the respective domains.

4.3.1 TAMING AGENTS AND OBJECTS (TAO)

One of the early approaches combining the natures of roles is called **Taming Agents and Objects (TAO)**. Da Silva et al. [2003] introduced TAO as a conceptual framework for the development of MAS. In detail, Da Silva et al. [2003] aim to provide *"a definition of an ontology that defines the essential concepts, or abstractions, for developing MASs."* [Da Silva et al., 2003, p.2]. Basically, their goal is to establish a formalization based on a unified terminology of the various MAS including *objects* and *agents* collaborating in *organizations* by playing *roles*. Moreover, these elements can be related by various kinds of relations, such as *ownership*, *play*, *association* and *aggregation*. While *associations* and *aggregations* between objects, agents and roles correspond to the respective UML relations, the *play* relation directly maps agents, objects, and organizations to their corresponding role types.

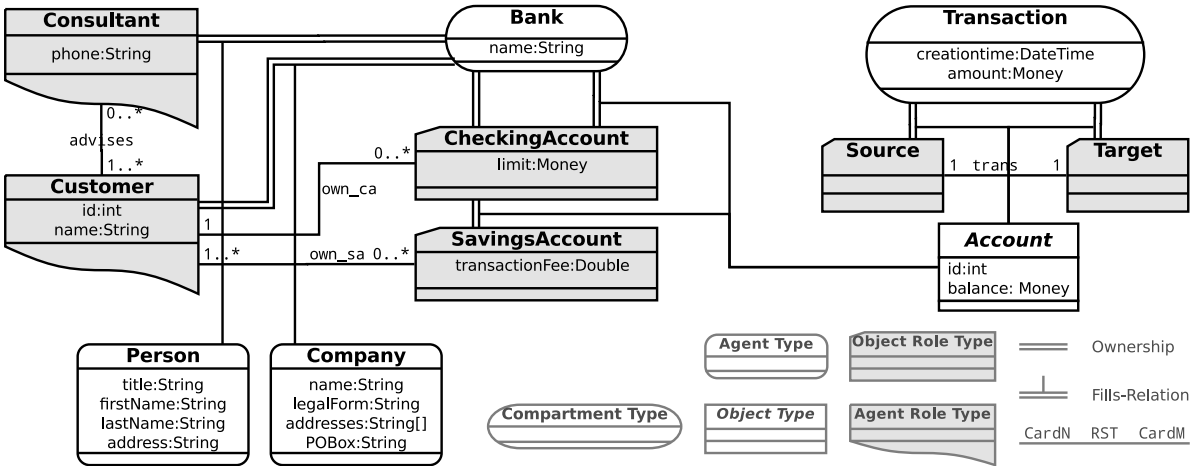

Figure 4.7: Bank example modeled in MAS-ML.

Similarly, the *ownership* relation between organizations and roles specifies which role types participate in which type of organization. Additionally, it captures how an organization is structured into sub-organizations. Nonetheless, TAO distinguishes between object roles and agent roles, that either influence their player's state and behavior or goals and beliefs, respectively. Finally, all objects, agents, and organizations inhabit an *environment* that manages their interrelations. In conclusion, TAO provides a comprehensive set of concepts and relations that not only permits formalizing MAS, but also modeling them using a graphical modeling language. The *multi-agent system modeling language (MAS-ML)* [Da Silva and De Lucena, 2004, 2007], in particular, provides a graphical representation of MAS modeled using the TAO conceptual framework. Accordingly, Figure 4.7 shows the corresponding *MAS-ML* model of the banking application. Here, both the natural types `Person` and `Company` are modeled as *agents*, whereas the natural type `Account` as an *object*. Moreover, all compartment types in the running example are defined as *organizations*. First, the `Bank` organization contains the *agent roles* `Consultant` and `Customer`, as well as the *object roles* `CheckingAccount` and `SavingsAccount` denoted by a double line. Second, the `Transaction` organization owns the `Source` and `Target` *object roles*. Additionally, for each *ownership* relation a simple line denotes the agent types or object types able to play the corresponding role type. Last but not least, the *MAS-ML* model specifies the various relationships, e.g. `advises`, `own_ca`, `own_sa`, including their cardinalities. The `own_ca` relationship, for instance, between `Customer` and `CheckingAccount` is defined as one-to-many relation, to state that a checking account has exactly one owner. In sum, this model appropriately captures the behavioral, relational and context-dependent nature of the banking application. However, as the conceptual framework only supports *cardinality constraints*, the more complex financial regulations could not be specified in the model. Thus, while both *TAO* and *MAS-ML* successfully combine the behavioral, relational and context-dependent nature of roles, they only lack notions to constrain the modeled domain. In detail, both agent and object roles completely fulfill the behavioral Features 1, 3, 4, 7, 10, 11, 13, and 15 states, for instance, that roles have their own properties, behavior, identity and can employ inheritance. Besides that, roles in *TAO* are only transferred (Feature 9) when their player moves from one environment to another. Likewise, agent roles only indirectly restrict access (Feature 12), as agents can only access those relationships defined by the played roles. Thus, these features are considered partially fulfilled. Similar to the behavioral nature, *TAO* also satisfies most of the relational and contextual features of roles, i.e., Feature 2, 16, 19, 20, 22, and 24–26. Thus, roles depend at least on the *ownership* relationship and can depend on various *association* and/or *aggregation* relationships. However, only associa-

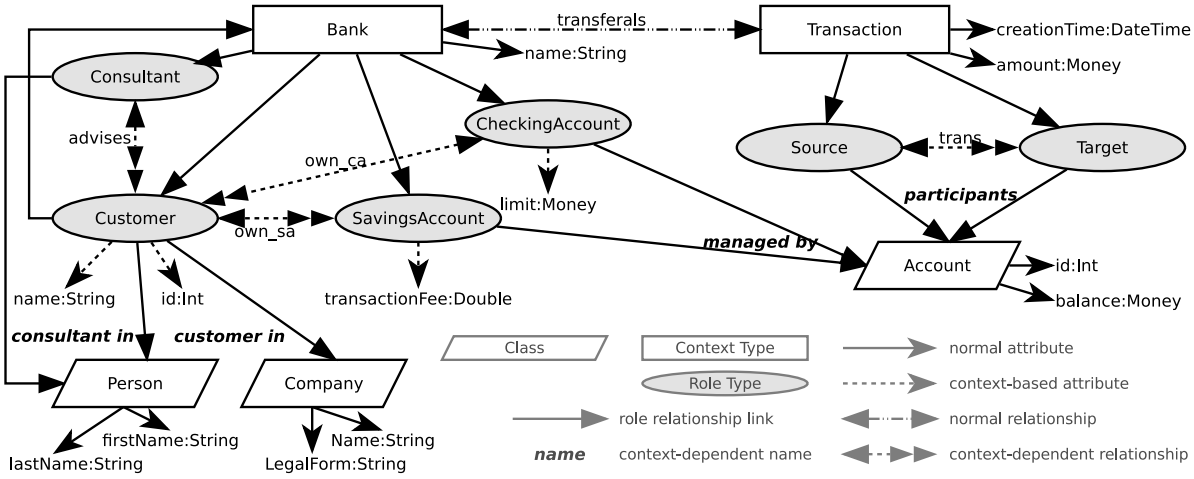

Figure 4.8: Bank example modeled in the INM.

tion and aggregation relationships can be constrained with *cardinalities* in *MAS-ML*. Ultimately, roles depend on organizations (as notion of compartment) that can have rules, laws, and their own identity. Finally, organization can play *agent roles*, can contain sub-organizations, and can inherit properties from other organizations. As it turns out, *TAO* and *MAS-ML* already fulfills most of the features of roles. In fact, *MAS-ML* supports more features than the succeeding approaches until 2015. Nonetheless, Feature 5 and 6 could not be applied to *TAO* as the conceptual framework did not provide an operational semantics. In conclusion, Da Silva and De Lucena [2004] not only provided a solid conceptual foundation for MAS, but also one of the first combined RML for their design. Regardless of its age, both are still applicable for nowadays complex and dynamic domains, e.g., the mitigation phase of *Emergency Response Environments* [Adamzadeh et al., 2014].

4.3.2 INFORMATION NETWORKING MODEL (INM)

Similar to the data modeling language *ORM 2*, **Information Networking Model (INM)** [Liu and Hu, 2009a] is another data modeling approach developed by Liu and Hu [2009a]. It is designed to overcome the limitations of classical data models, object-oriented models, and role models to capture the various natural, complex and context-dependent relationships of real-world objects [Liu and Hu, 2009a]. Specifically, Liu and Hu [2009a] proposed *INM* as a novel model *"to represent not only static but also dynamic context-dependent information regarding objects and various kinds of relationships between objects"* [Liu and Hu, 2009a, p.132]. Hence, Liu and Hu [2009a] suggests enriching the classical ER adding entity types for *contexts* and *roles*, as well as various kinds of relationships, such as *role relationships*, *context-dependent attributes*, and *context-dependent relationships*. On the one hand, *role relationship links* of a role type specifies both the context it is participating in (with an incoming link) and the natural types able to play this role type (with outgoing arrows). On the other hand, *context-dependent attributes* and *context-dependent relationships* specify the individual properties and relationships of roles, respectively. In contrast to *normal attributes* and *relationships* of entity types, these attributes and relationships can only originate from roles and are only accessible in a given context. Nonetheless, context types are treated like classes, as they can have arbitrary attributes, relationships, and super types. Thus, context types directly correspond to the notion of compartment types. Moreover, a class becomes a context if it contains a role, i.e. it has a role relationship to a role. While it is true that, this design decision leads to a very flexible modeling language, I still maintain that mixing these conceptually different entities create a

non-lucid modeling language [Guizzardi et al., 2005]. To put it bluntly, the (graphical) notation becomes hard to comprehend, as both context types and classes are represented by the same language construct. Hence, to distinguish the two entity types when modeling the banking application, Figure 4.8 uses rectangles to denote context types (compartment types) and parallelograms to denote classes (natural types). In particular, both `Bank` and `Transaction` are modeled as context types with individual attributes and the normal relationship `transferal` denoting the owner of a transaction. Conversely, role types, such as the `Customer`, have context-dependent attributes and three context-dependent relationships, e.g., `advises`, `own_sa` and `own_ca`. Last but not least, persons, companies and accounts are specified as classes with role relationship links denoting those role types these classes can fulfill and refer to using the *context-relationship name*. In sum, *INM* is able to sufficiently model the dynamics and context-dependent properties of the banking domain. Similar to TAO, INM only lacks the constraints to capture the financial regulations, e.g. specifying single ownership of checking accounts with *cardinality constraints*. It follows, then that *INM* supports the behavioral, relational, and context-dependent nature of roles [Liu and Hu, 2009a]. By extension for the behavioral nature, it supports Features 1, 3, 4, 8, 10, 11, 13, and 14 completely. As already discussed, roles have properties and can be played by unrelated entities. Moreover, objects may play the same role (type) several times, however roles share their identity with their player. Conversely, *INM* satisfies the relational Feature 2 and contextual Feature 19, 20, 22, and 24–26 completely, as it provides both a notion for relationships and compartment types. In fact, context types directly correspond to compartment types having properties, can inherit from a super type, can play roles and have their own identity. By contrast, *INM* only partially provides that a role can be part of several contexts (Feature 21) by utilizing inheritance. Indeed, only Feature 5, 9, and 12 could not be applied, because a data modeling does not provide an operational semantics. In conclusion, *INM* successfully combines the three natures into an expressive data modeling language, however, lacks the various modeling constraints to be practical. Nevertheless, they not only provided a formal underpinning [Hu and Liu, 2009], but also an implementation of a corresponding database based on a Key-Value store and accessible with their own query language [Hu et al., 2010].

4.3.3 HELENA APPROACH

The most recent RML is the **Helena Approach**. Hennicker and Klarl [2014] proposed it as role-based approach for the design of massively distributed systems by means of *ensembles*, i.e. collections of autonomic collaborating entities [Hennicker and Klarl, 2014]. Their approach, in contrast to other architecture modeling languages, introduce roles to provide *"a rigorous formal foundation for ensemble modeling that can be used during requirements elicitation and as a basis for the development of designs"* [Hennicker and Klarl, 2014, p.360]. In essence, they not only introduce a RML, but also its formal foundation based on *Labeled Transition Systems* [Hennicker and Klarl, 2014]. This, in turn, allows for verifying the correctness of the modeled architecture [Hennicker et al., 2015] as well as generating the corresponding implementation [Klarl et al., 2015]. In particular, the modeling language introduces *ensembles* to capture the dynamic collaborative behavior of distributed *components* by means of *roles*. The *Helena Approach* utilizes *ensemble structures* as reification of collaborations containing *role types* specifying the behavior of components playing these roles [Hennicker and Klarl, 2014]. Additionally, *role connectors* define the communication between two role types, i.e. they specify the role operations accessible by the role type. Similar to relationships, role connectors represent simple directed channels between two role types. Last but not least, the *Helena Approach* introduces *occurrence constraints*, to restrict the number of roles of a given type within an *ensemble*. With these concepts, *Helena's* modeling language is able to capture the complex, dynamic, and collaborative behavior of distributed components.

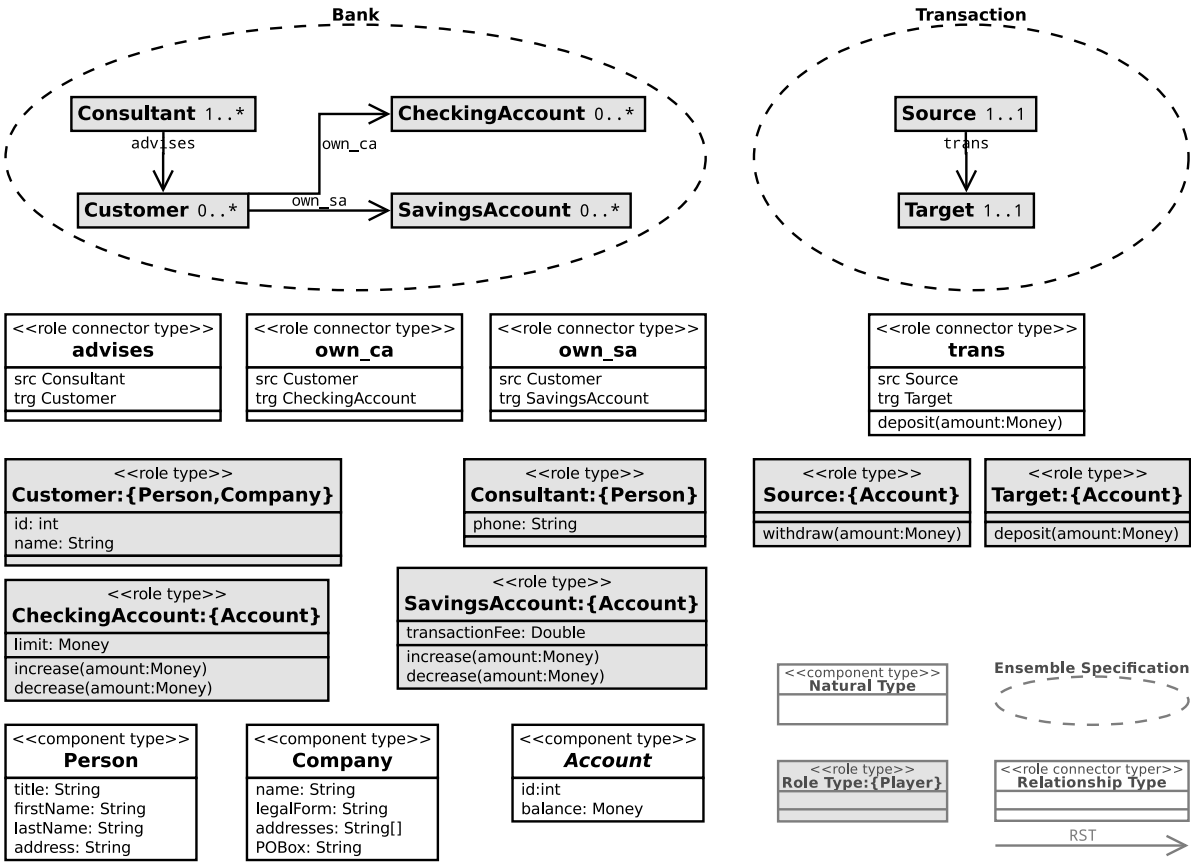

Figure 4.9: Bank example specified with the Helena Approach.

Notably though, as *components* do not have intrinsic behavior, their behavior is solely dependent on the roles they play. Although this simplifies the formalization of the model, I argue that this fails to define how roles adapt the component's behavior, as components are merely data containers. Besides all that, the *Helena Approach* provides a graphical notation based on UML class diagrams and stereotypes [Hennicker and Klarl, 2014] that is henceforth utilized to model the banking application. From top to bottom, Figure 4.9 shows the definition of ensemble structures, role connectors, role types and components. The `Bank`, for instance, contains the role types `Consultant`, `Customer`, `CheckingAccount` and `SavingsAccount`, as well as the role connectors `advises`, `own_ca`, and `own_sa`. Furthermore, the `Consultant` role type is constrained with a `1..*` occurrence constraint to state that any bank ensemble must have at least one component playing the `Consultant` role. Next, the role connectors are defined as classes with the stereotype `<<role connector type>>`. For instance, `trans` declares that the `Source` role can access the `deposit` method of the `Target` role. Afterwards, the corresponding role types of each *ensemble structures* are defined with the corresponding stereotype. Role types, such as `Customer`, denote the set of component types able to fulfill them by listing their names after the role name, e.g. `{Person,Company}`. Last but not least, it defines the natural types `Person`, `Company`, and `Account` as classes annotated with the `<<component type>>` stereotype and the corresponding attributes. In sum, this model captures the dynamics of the banking application, however also lacks more elaborate notions to restrict the modeled domain. Although role connectors are directed relations between role types, they cannot replace first-class relationships, as they suffer from the same problems as relationships implemented with references [Rumbaugh, 1987]. Nevertheless, the *Helena Approach* successfully com-

bines the behavioral, relational, and context-dependent nature of roles [Hennicker and Klarl, 2014]. In particular, it supports Feature 1, 3–5, 7, 10, 11, 12, and 15 of the behavioral nature by introducing roles with properties, behavior, and their own identity that can be played by unrelated components. Moreover, components can acquire and abandon roles dynamically and play multiple roles simultaneously. Likewise, the *Helena Approach* supports the relational nature by including role connectors between role types that can define relationships between roles (Feature 2). Finally, it embraces the context-dependent nature of roles by facilitating ensembles as first-class citizens. The notion of ensembles resemble compartments, as it fulfills Feature 19–21, 26, and 27. Specifically, ensembles can have properties and behaviors in *jHelena* [Klarl et al., 2015] and, generally, have their own identity. Moreover, ensemble specifications permit the definition of *occurrence constraints* for role types and containing role types of another ensemble specification. In conclusion, the *Helena Approach* not only introduced a novel modeling language successfully combining the three natures of roles, but also provided a rigorous formal foundation and operational semantics [Hennicker and Klarl, 2014], as well as tool support for its verification [Hennicker et al., 2015] and implementation [Klarl et al., 2015].

In consequence, the presented RMLs utilize individual natures of roles to cope with the increased complexity, dynamics and context-dependence of current domains. Even though, most approaches focus on a particular modeling domain, e.g., conceptual modeling, data modeling or architecture modeling, they all demonstrate the benefits of the role concept. One would assume that progress in either of these domains would be transferred to another domain. However, after reviewing the various contemporary RMLs, it becomes evident that most approaches simply introduce their own definition for roles and did not build upon previous definition. In fact, Chapter 6 compares the evaluation results of the various approaches and elucidates the apparent fragmentation and discontinuity in the research field on role-based modeling.

"Roles are first-class components of the end user cognitive model, so we want to reflect them in the code."

— Reenskaug and Coplien [2009]

5 CONTEMPORARY ROLE-BASED PROGRAMMING LANGUAGES

Generally, computer scientists tend to believe that modeling and programming are separate processes involving different concepts and constructs. Though I concede that both operate on different levels of abstraction, I still insist that both are tasked to capture the cognitive model of a domain expert and/or end user. Hence, both modeling and programming languages should provide concepts of their user's cognitive model. Undoubtedly, roles are an integral part of our cognitive model, as argued by Reenskaug and Coplien [2009] above. Yet, up to this point the discussion mainly focused on the representation of roles in RMLs. Conversely, this chapter investigates how the various contemporary role-based programming languages (RPLs) define roles and encode the user's domain model. Additionally, the investigation focuses on the language constructs introduced to specify role-based programs. Parts of this investigation have been published in [Kühn et al., 2014] and [Kühn and Cazzola, 2016]. Similar to the previous chapter, the discussion is trisected in accordance to the natures of roles supported by the contemporary RPLs. In detail, Section 5.1 emphasizes those programming languages that only incorporate the behavioral nature of roles. Afterwards, Section 5.2 considers languages that combine the behavioral and relational nature of roles. Finally, Section 5.3 discusses RPLs that successfully combined the context-dependent and behavioral nature of roles. Notably though, none of the contemporary programming languages fully combined all the natures of roles.

5.1 BEHAVIORAL PROGRAMMING LANGUAGES

The first category to consider, is the category of behavioral RPLs – languages that exclusively establish roles as language constructs. In general, these contemporary RPLs extend the *Java* programming language to permit the definition, creation, and binding of roles. In particular, this section covers the following four behavioral RPLs. The first section discusses *Chameleon* [Graversen and Østerbye, 2003], an early compiler and runtime that features aspect-like method overriding. In contrast, Section 5.1.2 presents *JAWIRO* [Selçuk and Erdoğan, 2004], a programming library that permits modeling with roles. Afterwards, Section 5.1.3 elaborates on *Rava* [He et al., 2006], a preprocessor that provides syntactic constructs for easily implementing the *Role-Object Pattern* [Bäumer et al., 1998]. Last but not least, Section 5.1.4 highlights *JavaStage* [Barbosa and Aguiar, 2012], a preprocessor for Java that facilitates static roles.

Listing 5.1: Bank example implemented in Chameleon.

```
1  class Account { Money balance; void increase(amount:Money) /*...*/}
2  role Source roleifies Account{
3    void withdraw(Money amount){ intrinsic.decrease(amount) }
4  }
5  role Target roleifies Account{ void deposit(Money amount){/*...*/} }
6  class Transaction{
7    Source source; Target target; Money amount;
8    Transaction(Account s, Money a, Account t){
9      source=new Source(s); amount=a; target=new Target(t); }
10   void execute(){source.withdraw(amount); target.deposit(amount);}
11 }
12 class Person {String firstName; /*...*/}
13 class Company {String name; /*...*/}
14 role Consultant roleifies Person{/*...*/}
15 role Customer roleifies Object{
16   List<Checking> ca; List<Savings> sa; /*...*/
17 }
18 role Checking roleifies Account{Money limit;
19   constituent void increase(amount:Money){ /*...*/ }
20 }
21 role Savings  roleifies Account{double transactionFee; /*...*/}
22 class Bank{ String name; List<Customer> customers; /*...*/}
```

5.1.1 CHAMELEON

Chameleon is a language extension developed by Graversen and Østerbye [2003] that introduces roles to the *Java* programming language. The *Chameleon* compiler, implemented using the *OpenJava* extensible compiler, is able to generate plain Java code accompanied by small runtime library [Graversen and Østerbye, 2003]. The language adds the keywords `role` and `roleifies` to declare role types and their player type, respectively. As main contribution, *Chameleon* features so called `constituent` methods. Similar to advices in aspect-oriented programming (AOP), these methods are not allowed to be invoked directly, but hook into the natural's method and are executed before, after or instead of the actual method. While a constituent method is invoked, `self` always refers to the most specific type available (the latest possible binding to self). The authors emphasize that constituent methods are key to role-based programming, because a *"role can be attached, and even without being used, it can have some of its code executed."* [Graversen and Østerbye, 2003]. Basically, Graversen and Østerbye highlight that in order to adapt existing objects, roles must be able to override methods of their player. In addition, *Chameleon* introduces *life roles* as roles that are created and bound automatically whenever its player is instantiated [Graversen and Østerbye, 2003]. This enables dynamic aspects, i.e., applying aspects on the perspectives of objects. Graversen and Østerbye [2003] argue that the implementation of aspects with united constituent methods (grouped in roles) makes no difference compared to defining them in separate blocks. However, I would object that mixing aspects and roles is conceptually wrong, because aspects are crosscutting by definition while roles are not. Nonetheless, the major drawback of Chameleon is the fact that roles extend their player. While gaining access to protected properties, due to Javas restriction to single inheritance, it is impossible that a role inherits from another role. Additionally, because roles are implemented as adjunct instances [Steimann, 2000b], all state is kept redundantly in both the player and its roles. Nonetheless, the *Chameleon* language is capable of implementing the banking application, as shown in Listing 5.1. In detail, the natural types `Account`, `Person`, and `Company` are implemented as regular *Java* `class`, whereas the role types `Source`, `Target`, `Consultant` and `Customer` as `role`. For each role type, the `roleifies` keyword defines one natural type allowed to play this role, e.g. that `Source` and `Target` can be played by `Account` objects. Of course, this

entails that a role type cannot be played by unrelated objects. Hence, to model that customers can be either persons or companies, the `Customer` role type is filled by `Object`. However, this includes not only `Person` and `Company` objects, but also any other *Java* class. In addition to that, *Chameleon* facilitates access to a role's player object either explicitly using the `intrinsic` keyword (Listing 5.1, Line 3) or implicitly using `constituent` methods (Listing 5.1, Line 19). Finally, both the `Transaction` and `Bank` compartment types must be implemented as regular classes. In sum, although *Chameleon* provides important language constructs to define roles and their dynamic behavior, its underlying role model is limited to the behavioral nature [Graversen and Østerbye, 2003]. Accordingly, the language extension supports most of the behavioral features of roles, such as Feature 1, 3–5, 9–12, and 15 completely and only Feature 14 partially. In general, *Chameleon* facilitates roles as first-class citizens of the programming language with individual properties, behavior, and identity that can be attached, detached, and transferred dynamically. Besides that, constituent methods grant roles the ability to adapt their player's state and behavior and, thus, to restrict access. Moreover, constituent methods can be utilized to avoid *object schizophrenia* by overriding the `equals` method, as proposed in [Herrmann, 2007]. Consequently, both a role and its player object would be considered equal and, therefore, would have a shared identity. In conclusion, *Chameleon* is one of the earliest approaches that investigates the intricacies of dynamic dispatch in RPLs. Unfortunately, neither the grammar for their language extension nor a proof-of-concept implementation of their compiler has been made available [Graversen and Østerbye, 2002, 2003, Graverson, 2003]. Besides all that, Graversen [2006] later published a thorough investigation of the features of compilers and runtime environments for RPLs in his PhD thesis, not unlike this thesis.

5.1.2 JAVA WITH ROLES (JAWIRO)

In contrast to *Chameleon*, **Java with Roles (JAWIRO)** does not rely on a compiler, as Selçuk and Erdoğan [2004] provide the notion of *roles* and *role hierarchies* with a simple application library. Their goal was *"to implement an extendible, simple yet powerful role model without the restrictions [...] imposed by previous work on role models."* [Selçuk and Erdoğan, 2004, p.928]. In particular, this role model establishes that *actors* can dynamically assume, abandon, suspend, resume, and transfer *roles*. Moreover, Selçuk and Erdoğan [2004] introduce *aggregate roles* to enable *actors* playing multiple roles of the same type simultaneously. In fact, they highlight that playing roles corresponds to object-level inheritance. Hence, roles are able to dynamically intercept and either forward or delegate messages to their player [Selçuk and Erdoğan, 2006]. Furthermore, their role model permits that roles can play roles and that both object- and class-level inheritance can be mixed. This, in turn, gives rise to *role hierarchies* for each actor. In fact, a role hierarchy is a tree of all roles of an actor acquired directly or indirectly (via one of its roles). As a result, Selçuk and Erdoğan [2004] provided the three super classes `Actor`, `AggregateRole`, and `Role` as well as the two interfaces `RoleInterface` and `ConstraintStrategy` to establish their role model. Accordingly, all types able to play roles must be implemented as subclass of `Actor` and all role types as subclass of `Role` or `AggregateRole`. In the implementation of the banking application shown in Listing 5.2, for instance, the classes `Account`, `Person`, `Company` and `Bank` inherit from `Actor`, whereas the role types `Source`, `Target`, `Savings`, and `Checking` from `Role`. However, to ensure that a person can be a customer multiple times, the `Customer` class is specified as an `AggregateRole`. Additionally, the example highlights the delegation of the `decrease` method from the `Source` role to its actor using `executeMethod` (Line 4) and accessing a specific aggregate role using the library method `as` (Line 24). Nonetheless, neither the relationships (e.g. advises, own_ca) nor compartment types (e.g. bank, transaction) can be represented as distinct entities.

Listing 5.2: Bank example implemented in JAWIRO.

```
1  class Account extends Actor{ Money balance; /*...*/}
2  class Source extends Role{
3    void withdraw(Money amount){
4      getActor().executeMethod("decrease",amount);
5    }
6  }
7  class Target extends Role{ void deposit(Money amount){/*...*/} }
8  class Transaction{
9    Source source; Target target; Money amount;
10   Transaction(Account s, Money a, Account t){
11     source=new Source(); s.addRole(source); amount=a;
12     target=new Target(); t.addRole(target);
13   }
14   void execute(){source.withdraw(amount); target.deposit(amount);}
15 }
16 class Person     extends Actor{String firstName; /*...*/}
17 class Company    extends Actor{String name; /*...*/}
18 class Customer   extends AggregateRole{String cid; /*...*/}
19 class Consultant extends Role{/*...*/}
20 class Checking   extends Role{Money limit; /*...*/}
21 class Savings    extends Role{double transactionFee; /*...*/}
22 class Bank extends Actor { String bank; /*...*/
23   Money getBalance(Actor a){
24     return ((Customer) a.as("Customer",bank)).balance();
25   }
26 /*...*/}
```

Although this implementation sufficiently supports the behavioral aspects of the banking application, both compartment types and relationships must be manually recreated using *Java* primitives. Nonetheless, *JAWIRO* fully embraces the behavioral nature of roles [Selçuk and Erdoğan, 2004] by establishing Feature 1, 3–5, 7–13, and 15. In short, roles have properties, behavior, inherit from another and have their own object identity. Actors, in turn, can acquire, abandon, and transfer roles dynamically and can play multiple instances of an aggregate role type simultaneously. Finally, roles adapt their player's behavior and state, and thus can be used to implement access restrictions. Besides all that, to restrict the sequence of role acquisitions and removals (Feature 6), a custom `ConstraintStrategy` must be implemented [Selçuk and Erdoğan, 2006].[1] Similarly, to ensure that an actor and its roles share identity (Feature 14), both `Actor` and `Role` classes must implement a role agnostic `equals` method [Herrmann, 2007]. In conclusion, *JAWIRO* presents an early approach to a behavioral RPL introducing the notion of roles to the *JAVA* programming language without requiring a custom compiler or preprocessor. Regardless of the promising results, *JAWIRO's* implementation appears to be unavailable even upon personal request.

5.1.3 RAVA

Likewise, **Rava** [He et al., 2006] is a lightweight language extension to *Java* solely focusing on the behavioral nature of roles. In contrast to theses, *Rava* employ the *Role-Object Pattern* [Bäumer et al., 1998] to support forwarding and delegation and the *Mediator Pattern* [Gamma et al., 1994] to manage the dynamic role bindings. He et al. [2006] believes that this *"separates the binding relationship from core object and role [...], which reduces the coupling between object and role, and improves their reusability."* [He et al., 2006, p.7]. In making this comment, He et al. [2006] argues that neither the player nor the role should manage the *plays* relationship, as this would lead to in-

[1] For brevity, the example omitted the definition of a `ConstraintStrategy`.

Listing 5.3: Bank example implemented in Rava.

```
1  class Account { Money balance; /*...*/}
2  ROLE Source roleof Account{
3    void withdraw(Money amount){ @core Account().decrease(a); }
4  }
5  ROLE Target roleof Account{/*...*/}
6  class Transaction{ Account source,target; Money amount;
7    Transaction(Account source, Money amount, Account target){/*...*/}
8    boolean execute(){
9      @INVOKEROLE(source,"Account","Source","withdraw",amount); /*...*/
10   }
11 }
12 class Person     extends Actor{String firstName; /*...*/}
13 class Company    extends Actor{String name; /*...*/}
14 ROLE Consultant roleof Person{/*...*/}
15 ROLE Customer   roleof Person,Company,Bank{/*...*/}
16 ROLE Savings roleof Account{ /*...*/
17   void decrease(double a){ @core Account().decrease(a*fee); }
18 }
19 ROLE Checking roleof Account{ /*...*/
20   void decrease(double a){ if(a<limit)@core Account().decrease(a); }
21 }
22 class Bank{ String bank; /*...*/
23   Money getBalance(Person p){
24     return (Money) @INVOKEROLE(p,"Person","Customer","balance");
25   }
26 /*...*/}
```

creased coupling of the two and would hinder reuse of either of them. In accordance, He et al. [2006] proposes to use individual *mediator objects* to manage the dynamic bindings between objects and roles. However, to reduce the implementation overhead of the two design pattern, they implemented a preprocessor that transparently translates, 4 new keywords by means of 4 grammar rules to plain *Java* code. In Listing 5.3, `ROLE` is a prefix indicating the definition of role types, e.g. `Source`, `Target`, `Customer`, and `Consultant`, whereas `roleof` specifies the set of classes (including role types) able to fulfill this role type. In addition, the keywords `@core` and `@INVOKEROLE` are available within role types to permit access to its player and explicitly invoke a method from a bound role, respectively. Consider, for instance, the `withdraw` method (Line 3) use `@core Account()` to obtain a reference to its player. Conversely, the `getBalance` method (Line 24) explicitly calls the method `balance` of the `Customer` role attached to the `Person p`. Nonetheless, both natural types (i.e. `Account`, `Person`, `Company`) and compartment types (i.e. `Bank`, `Transaction`) must be implemented as regular classes. Moreover, to add and remove roles from a player object, the implementation has to consult the corresponding mediator object from a singleton `MediatorFactory` [He et al., 2006, Fig.12]. Overall, the implementation in Listing 5.3 follows the syntax of *Rava* and covers the behavioral nature of the banking application.

Even though there is no compiler available, the evaluation relies on the detailed description of its implementation in [He et al., 2006]. In general, *Rava* supports the behavioral nature of roles [He et al., 2006] by fulfilling Feature 1, 3, 5, 7, 10–13, and 15. Specifically, the language facilitates roles with properties, behaviors, inheritance and independent identity. Furthermore, roles can modify the state and behavior of their player and thus enforce access restrictions. Using the mediator object, roles can be dynamically attached and detached to unrelated objects. Finally, in contrast to *JAWIRO*, *Rava* also permits legacy objects to play roles. In conclusion, *Rava* is a very lightweight approach introducing roles to *Java*. It focuses on the key aspects of the behavioral nature of roles, namely definition of role types and the *fills* relation as well as access to its player and role methods.

Listing 5.4: Bank example implemented with JavaStage.

```
1  class Account {plays Source s; plays Target t;
2      Money balance; /*...*/
3  }
4  role Source {requires Perform implements void decrease(Money amount);
5    void withdraw(Money amount){performer.decrease(a); }
6  }
7  role Target {/*...*/}
8  class Transaction{Account source,target; Money amount; /*...*/
9    void execute(){source.withdraw(amount); target.deposit(amount);}
10 }
11 class Savings  extends Account {/*...*/
12   void decrease(double a){ super.decrease(a*fee); }
13 }
14 class Checking extends Account {/*...*/}
15 class Person  {plays Customer<Bank> bc; plays Consultant con; /*...*/
16   String getName(){ return firstName +" "+ lastName; }
17 /*...*/}
18 class Company {plays Customer<Bank> bc; /*...*/
19   String getName(){ return name +" "+ legalForm; }
20 /*...*/}
21 role Consultant  {/*...*/}
22 role Customer<T> {requires T implements String getName();
23   Money balance() {/*...*/}
24 }
25 class Bank    {plays Customer<Bank> bc; /*...*/
26   String getName(){ return name}; }
27   Money getBalance(Person p){ return p.bc.balance(); }
28 }
```

5.1.4 JAVASTAGE

While the previous RPLs mainly focused on dynamic roles, **JavaStage** features *static roles*, i.e., roles that are statically bound to a class and played by all its instances at runtime. In particular, Barbosa and Aguiar [2012] complain that *"[even] the languages that deal with the dynamic aspects of roles neglect this static nature"* [Barbosa and Aguiar, 2012, p.125]. They argue that static roles already benefit software engineers by fostering code reuse, comprehension, development and documentation [Barbosa and Aguiar, 2012]. Accordingly, Barbosa and Aguiar [2012] propose a small extension to *Java* that adds the notion of roles. It adds five additional keywords as well as a renaming mechanism (using the "#" character). First, the keyword `role` is introduced to declare static roles similar to classes. However, role types can contain a `requires` clause specifying the method(s) the classes playing this role must provide. Within this clause the keyword `Performer` refers to an anonymous class playing this role type. Accordingly, within role methods `performer` permits access to the actual player instance. Last but not least, the `plays` declaration is used within the class body to declare and instantiate a given role type. Besides that, the renaming mechanism permits the instantiation of generic roles, as it can be used to avoid naming conflicts.

To further elucidate these language constructs, the banking application is considered, again. As shown in Listing 5.4, the role types `Source`, `Target`, `Customer`, and `Consultant` are declared with the prefix `role`, however their players are only declared by specifying the required methods, e.g. any class (noted as `Performer`) can play the `Source` role type, as long as it implements the proper `decrease` method (Line 4). In contrast, `Customer` is a generic role type with the type parameter `T` that only requires the `getName` method from its player (Line 22). While the list of requirements avoids explicit declaration of the *fills* relation, generic role types allow for playing multiple kinds

of Customer, e.g. the customer of a bank Customer<Bank> and a shop Customer<Shop> simultaneously. Nonetheless, one needs to take special care to conceive unambiguous method names or employ a sufficient renaming strategy. For instance, the requirement of the Customer role type is far to broad, as it would permit any class with a proper getName method to play the customer role. Afterwards, the classes Account, Savings, Checking, Person, Company and Bank use the play declaration to bind a specific role type by providing its type parameter and a field name to access its instance. The Person class, for instance, binds the Customer role type providing the Bank as generic type parameter and bc as field name (Line 25). This field name can be used later to directly access the role of a player, e.g. with p.bc in Line 27. Similarly, the performer keyword permits the Source role type access to the instance of its player (Line 5). From the technical side, *JavaStage* is implemented as a preprocessor that generates and compiles *Java* and *JavaStage* code to *JVM* compatible *Java* source code. Specifically, all role types are generated as inner classes of their corresponding player class after applying the renaming. Moreover, all visible role methods are mirrored to the player class forwarding their calls to the corresponding role instance. Although *JavaStage* fails to model the dynamics of the banking application, it still manages to improve the reusability of roles by facilitating both a requirements list and generic role types. Moreover, their case study indicates the ability of roles to feasibly resolve crosscutting constraints [Barbosa and Aguiar, 2013]. Ultimately, *JavaStage* supports the behavioral nature of roles [Barbosa and Aguiar, 2012]. To put it bluntly, its static roles support Feature 1, 3, 6–8, 10–13, and 15 completely. They have properties, behavior, and their own identity. Moreover, they can use single inheritance and can be played by unrelated classes. In contrast, *JavaStage* only partially fulfills Feature 4. While generic role types allow for binding similar role types to a class, it is still impossible to specify that a person is a customer of multiple banks. In conclusion, Barbosa and Aguiar [2012] introduced the notion of static roles to programming by providing a lightweight language extension to Java. They insist that *JavaStage* is the only RPL supporting static roles [Barbosa and Aguiar, 2012]. Unfortunately, the authors made neither a complete grammar nor the implementation of their language extension available.

5.2 RELATIONAL PROGRAMMING LANGUAGES

Undoubtedly there are a multiple programming languages that introduce relationships as first-class citizens, e.g. *RelJ* [Bierman and Wren, 2005], *Relationship Aspects* [Pearce and Noble, 2006], yet only few of them additionally establish roles, as well. Consequently, the literature review only considers those approaches that support the relational nature of roles, i.e. RPLs that provide both relationships and roles as first-class language constructs. Hence, the discussion revolves around the following three relational programming languages. First, *Rumer* [Balzer et al., 2007] is a full-fledged programming language for modular verification over shared state using relationships (Section 5.2.1). In contrast, *First-class Relationships* [Nelson et al., 2008] proposes a three-tier programming model encompassing objects, links, and relationships (Section 5.2.2). Last but not least, *Relations* [Harkes and Visser, 2014] is a novel programming language for data models featuring entities and relationships with roles (Section 5.2.3).

5.2.1 RUMER

While the previous RPLs have been developed as a language extension, **Rumer** [Balzer et al., 2007] is a full-fledged programming language focusing on the relational nature of roles. Balzer et al. [2007] argues that *"an object-oriented programming language with explicit support for relationships enjoys properties that facilitate the verification of real-world programs with invariants"* [Balzer and Gross,

Listing 5.5: Bank example implemented in Rumer.

```
1  entity Account{float balance=0.0; /*...*/}
2  entity Person {string firstName; /*...*/}
3  relationship Transaction participants(Account source,Account target){
4    extent boolean transfer(Account s, Account t){
5      return these.add(new Transaction(s,t));
6    }
7    boolean execute(float amount){ /*...*/ source.decrease(a); /*...*/}
8  /*...*/}
9  relationship Advises participants(Person consultant, Person customer){
10   extent invariant these.isIrreflexive();
11 /*...*/}
12 relationship Own_ca participants(Person customer,Account checking){
13   void > range limited(float a){ if(a<limit) this.decrease(a); }
14 /*...*/}
15 relationship Own_sa participants(Person customer,Account savings){
16   float > range fee;
17   void  > range withfee(float amount){ this.decrease(amount*fee); }
18 /*...*/}
19 application Bank{ Extent<Transaction> trans; Extent<Advises> advises;
20   main(){/*...*/}
21 /*...*/}
```

2011, p.360]. In accordance, they introduce binary *relationships* to capture the collaboration between objects [Balzer et al., 2007], as well as to allow for employing a *visible states verification* technique [Balzer and Gross, 2011]. Additionally, *Rumer* prohibits the use of references as attributes of objects, such that every object collaboration must be implemented using relationships. In general, *Rumer* is the only fully formally specified RPLs including a complete grammar with over 30 keywords and 67 rules as well as a formal operational semantics that has been proven to be sound with respect to the modular verification of multi-object invariants [Balzer, 2011].

To showcase its syntax, Listing 5.5 shows a partial implementation of the banking domain. The natural types *account* and *person*, for instance, are declared with the `entity` keyword (Line 1–2), similar to *Java* classes. Conversely, relationships, such as *advises*, *own_ca* and *own_sa*, are specified with the prefix `relationship` followed by a `participants` clause declaring the role types played by the collaborating naturals types. The `Advises` relationship type, for example, is declared between the `consultant` and `customer` role type, whereas both can be played by `Person` objects (Line 9). Moreover, it is augmented with an `extent invariant` to declare that the set of `Advises` links (referred to with `these`) is irreflexive. Besides, the *transaction* compartment type is also declared as a `relationship` to collect the various money transferals (Line 3–8). Granted, role types appear to be named places, so far. However, Balzer et al. [2007] proposed the concept of *member interposition* to allow for adding fields and methods to the role types of a particular relationship. Specifically, the member declaration with `> domain` and `> range` denote interposed members of the first and the second role type, respectively. Accordingly, Line 15–18 add the field `fee` and the method `withfee` to the `savings` role type of the `Own_sa` relationship. In addition to member interposition, *Rumer* supports the specification of `extent` methods, for instance, to add links to the relationship (cf. Line 5). These methods are available to the relationship extent, e.g. `Extent<Transaction>`, an objectified relationship that collects and manages all its links. Last but not least, the `Bank` itself is declared as `application` containing the various relationship extents, e.g. the set of transactions, and the application entry point, i.e. the main method. Even though this example showcases *Rumer's* language constructs, it can only hint the languages rich syntax and semantics. In sum, *Rumer* not only establishes the relational nature of roles by introducing relationships and a wide variety of *intra-relationship constraints*, but also the behavioral nature of roles by

utilizing member interposition [Balzer et al., 2007]. By extension, it supports most of the behavioral features, i.e. Feature 1, 3–6, 10–12, and 14. Specifically, role types are defined with properties and behavior. Moreover, although objects can play multiple roles simultaneously (Feature 4), roles can only be played by unrelated types, if the most common type is declared as player (Feature 7). Finally, roles and their player share their identity (Feature 14), as *Rumer* transparently creates, attaches, and removes roles. Even though member interposition implicitly declares role types, interposed members cannot be shared across several relationship types. For instance, any member added to the `customer` role type is only accessible within the declaring relationship type. Because *Rumer* supports almost arbitrary predicates as invariants, application invariants can be utilized to constrain multiple role types at once (Feature 18). Furthermore, *Rumer* satisfies Features 2 and 16 of the relational nature, because roles depend on relationships and relationships can be constrained. Indeed, Balzer et al. [2007] also incorporates parts of the contextual nature of roles by supporting Feature 19 partially and Feature 20, 22, 24, and 26 fully. In general, *Rumer's* notion of relationship extents as objectified relationship comes close to the notion of compartments. Although they are limited to one binary relationship, they can partially establish the dependence between role types and relationship extents (Feature 19). Additionally, relationship extents can have properties, behaviors and their own identity. Moreover, they can play roles like objects and can contain (own) other relationship extents. It follows, then that *Rumer* not only combined the relational nature and behavioral nature of roles, but also includes the essential features of the context-dependent nature of roles. Notably though, Balzer et al. [2007] designed *Rumer* specifically to show the feasibility of modular verification and formally proving the correctness of programs [Balzer and Gross, 2011]. Hence, providing a running compiler was none of their immediate goals.

5.2.2 FIRST CLASS RELATIONSHIPS

While the other RPLs discussed in this chapter introduce or extend a programming language, Nelson et al. [2008] argues that *"[rather] than adding relationships to existing language models [...] existing language models should be re-factored to support relationships as a primary metaphor"* [Nelson et al., 2008, p.34]. In particular, the **First Class Relationships** approach proposes a three-tier relationship model as a novel foundation for object-oriented programming languages. The *object tier* encompasses the definition of *classes* with fields and methods. These can be instantiated to *objects*, however, their attributes can only carry values, i.e. instances of value types or classes that exclusively belong to the carrying object [Nelson et al., 2008]. Moreover, each object has a unique identity and has exactly one type. The *link tier*, in turn, uses objects and classes to define *links* (tuples) between objects and *associations* between participating classes, respectively [Nelson et al., 2008]. Simply put, links are immutable tuples of objects belonging to an association that represents a cross product of classes. While this tier collects the individual links between objects, the *relationship tier* captures and models groups of links as *relationships* and *relations*. In particular, each *Relationship* contains a subset of all links of an association. Moreover, a *relationship* is defined by a *relation* that objectifies exactly one association and defines *role types* for each participant of an association. In short, Nelson et al. [2008] augments the classical object model with links, relationships, and roles to reconcile object-oriented design and implementation [Nelson et al., 2008]. Besides providing the relationship model, Nelson et al. [2008] additionally sketched a possible relational programming language by providing a partial grammar. This grammar features 14 keywords to define *classes*, *associations* between classes and *relationships* for a particular association, as well as separate inheritance relations for those types.

Listing 5.6: Bank example implemented using First Class Relationships.

```
1  /* object tier */
2  class Account {const int id; Money balance; /*...*/}
3  class Party   {/*...*/}
4  class Person  extends Party {const String firstName; /*...*/}
5  class Company extends Party {const String name; /*...*/}
6  /* link tier */
7  association trans   {
8    participant Account Source;    participant Account Target;   }
9  association advises {
10   participant Person Consultant; participant Party Customer;   }
11 association own_ca  {
12   participant Party Customer;    participant Account Checking; }
13 association own_sa  {
14   participant Party Customer;    participant Account Savings;  }
15 /* relationship tier */
16 relation Transaction contains trans{
17   role Source as SourceRole{ void withdraw(){/*...*/} }
18   role Target as TargetRole{ void deposit(){/*...*/} }
19   role trans  as TransRole { Money amount; /*...*/}
20 }
21 relation Bank contains advises{
22   role Consultant as ConsultantRole{String phone; /*...*/}
23   role Customer   as CustomerRole  {String name;  /*...*/}
24 }
```

Listing 5.6 showcases the language constructs by specifying the banking domain with *First Class Relationships*. Following the three tiers, the example first declares the natural types `Account`, `Person`, `Company` as classes (Line 1–4). Afterwards, four associations are specified using the keyword `association`, e.g. `trans`, `advises`, and `own_sa`. The `advises` association, for instance, is established between the class `Person` as `Consultant` and the class `Party` as `Customer` (Line 7–8). The `Party` class must be introduced to permit the otherwise unrelated classes `Person` and `Company` to participate in the same associations. Finally, the relationship tier declares the `Transaction` and the `Bank` relation derived from the `trans` and the `advises` association, respectively. Besides that, relations have their own fields and methods, they can also declare additional members for individual participants as well as links of the corresponding association. Consider the `Transaction` relation that declares the participant `Source` as `SourceRole` type adding the `withdraw` method (Line 14). Similarly, Line 15 attaches the field `amount` to the links of the `trans` association. As a result, the presented implementation is able to appropriately model most of the banking domain.

Surprisingly, the relationship model not only captures the behavioral and relational nature of roles, but also key features of the context-dependent nature [Nelson et al., 2008]. On the one hand, it establishes both role types and relations with properties and behaviors. Roles depend on relationships and are dynamically assumed or discarded whenever an object enters or leaves an association in the corresponding relationship. Thus, an object can play multiple roles at the same time and share its identity with all of them. However, unrelated objects cannot play the same role type (Feature 7) and relationships cannot be constrained (Feature 16, 17) As a result, *First Class Relationships* support Feature 1–5, 10–11, and 14 of both the behavioral and relational nature. On the other hand, the context-dependent nature of roles is implicitly supported. Especially, when relations and their relationships are considered as compartment types and compartment instances, respectively. Similar to *Rumer*, relationships represent the extent set of an association and represent entities in their own right. Relations, in particular, define the properties and behavior of a set of relationships, just like compartment types do. Moreover, relations can inherit both members and roles from another relation. Even though relationships can be seen as an objectification of a group of links, their

Listing 5.7: Bank example implemented in Relations.

```
module bank
model
  entity Account {balance : int = 0 /*...*/}
  entity Person  {firstName: String /*...*/}
  entity Company {name: String /*...*/}
  relation Transaction{
    Account 1 source    Account 1 target
    amount : int = 0
  }
  relation Bank{
    Person  * customer  Person  + consultant
    Account * savings   Account * checking
    customer.adviser <-> consultant.advisee
    customer.ca      <-> checking.owner
    customer.sa      <-> savings.owner
  }
data
  Person p1{ firstName="Doreen" /*...*/ }
  Account a1{ balance=1000 /*...*/ }
  Transaction t1{ source:a1 target:a2 amount=100 }
  Bank { customer:p1 consultant:p2 checking:a1 savings:a2 }
  Bank { customer:p3 consultant:p2 checking:a3 }
  /*...*/
execute sum(p2.advisee.ca.balance)
```

definition is limited to groups of exactly one association. In accordance to that, *First Class Relationships* fulfills Feature 19 partially and Feature 20, 25, and 26 completely. In conclusion, Nelson et al. [2008] emphasize that relational programming languages must provide links between objects as well as relationships grouping individual links. Moreover, their notion that roles locally affect their player within a particular relationship entails that roles are context-dependent with respect to a relationship. Regardless of the power and simplicity of the proposed relationship model, apparently none of the authors followed this research direction by publishing a proof-of-concept compiler or an operational semantics for the proposed programming language.

5.2.3 RELATIONS

As one of the most recent approaches, **Relations**[2] is an RPL developed by Harkes and Visser [2014]. However, in contrast to other RPLs, *Relations* is a role-based data modeling language featuring first-class *relations*, *native multiplicities*, and a concise navigation language for *derived attributes*. Although it is formally specified including its syntax, a multiplicity-aware type system, and operational semantics, Harkes and Visser [2014] additionally provided a running proof-of-concept implementation based on the language workbench *Spoofax* [Kats and Visser, 2010]. Compared to *Rumer*, the syntax of *Relations* is rather concise introducing 17 keywords in 9 grammar rules [Harkes and Visser, 2014]. However, *Relations* programs can be directly translated to executable *Java* source code. Of course the language is still under active development [Harkes and Visser, 2014, Harkes et al., 2016].[3] However, because only [Harkes and Visser, 2014] featured roles, the discussion henceforth only takes the corresponding release *v0.2.0* into account.[4] Considering the underlying data model, *Relations* encompasses *entity types* declared with `entity` and *n*-ary relationships defined with `relation` [Harkes and Visser, 2014].

[2] Now called *IceDust* [Harkes et al., 2016].
[3] `https://github.com/metaborg/relations`
[4] `https://github.com/metaborg/relations/releases/tag/v0.2.0`

Both entities and relations have fields, these however are limited to primitive types, i.e. `boolean`, `int`, and `string` [Harkes and Visser, 2014]. Moreover, relations can define multiple participating role types, whereas each one is defined by stating its player type, its name, a multiplicity, and the name of the inverse relation (from its player). In addition to that, relations can establish bidirectional navigation between two participating role types by providing reference names for both ends. Finally, the multiplicity given for each role type specifies the number of roles (of this type) a corresponding relation instances can contain. Specifically, the following four multiplicities are supported: arbitrary `*`, at least one `+`, exactly one `1`, and at most once `?`. In sum, a *Relations* program contains a `model` section providing the domain model, a `data` section adding an instance of this model and finally an `execute` section querying this instance model.

Accordingly, Listing 5.7 outlines the banking application implemented as *Relations* program. For each of the natural types *account*, *person*, and *company* the model section contains the declaration of a corresponding entity type. Afterwards, the `Transaction` relation is declared between exactly one `Account` as `source` and one `Account` as `target` and with the field `amount` (Line 6–9). More importantly, the `Bank` relation is defined encompassing four participating role types, i.e. `customer`, `consultant`, `savings`, and `checking`, whereas the former two are played by persons and the latter two by accounts. Additionally, navigations are defined for the three relationship in the banking domain, such as *advises*, *own_sa*, and *own_ca*. *Advises*, for instance, is defined in Line 13 as bidirectional navigation from `customer` with `adviser` to `consultant` with `advisee`. Undoubtedly, *Relations'* syntax leads to a very concise specification of the banking domain, however, it can only support a limit number of features of roles. To put it bluntly, *Relations* only supports Feature 2, 3, 4, and 14 stating that roles depend on relations and that entities can assume multiple roles simultaneously. Still, roles have neither properties, behaviors, nor their own identity (Feature 1, 15) and cannot be played by unrelated objects (Feature 7). Additionally, because *Relations'* operational semantics does not include operations to dynamically instantiate entities and relations, Feature 5 and 12 are not applicable. Regardless, it is still possible to restrict the sequence of role acquisition using multiplicities (Feature 6). Besides all that, *Relations* features the context-dependent nature of roles, as well. Granted one could argue that relations correspond to relationships and multiplicities to cardinalities, yet an instance of a relation is not a tuple of objects rather than a group of links. In fact, the bidirectional navigation established between two participants resembles a binary relationship [Chen, 1976]. Accordingly, I would argue that relations denote *compartment types* and multiplicities *occurrence constraints*. Following this argument, relations fulfill Feature 19, 20, 22, 26, and 27; if they are considered as compartments. Like entities, relations have properties, their own identity, and can play roles. It follows then, that Feature 24 is partially satisfied, because a relation playing a role in another relation implicitly states that the latter contains the former. In conclusion, Harkes and Visser [2014] designed a very lightweight role-based data modeling language that is not only formally specified, but also publicly available. Granted *Relations* does not fully support the natures of roles, yet it emphasizes the key connections between navigation links and binary relationships as well as n-ary relations and compartment types. In particular, this approach shows that relationships and compartments are not synonymous, but can complement each other.

5.3 CONTEXT-DEPENDENT PROGRAMMING LANGUAGES

After highlighting both behavioral and relational programming languages, this section finally elaborates on RPLs that establish the context-dependent nature of roles. Basically, these languages provide a notion of compartment to specify context- or collaboration-dependent behavior. First and foremost, *EpsilonJ* [Ubayashi and Tamai, 2001] is the earliest programming language extension

featuring environments and roles as language constructs. Both *EpsilonJ* and its successor *NextEJ* [Kamina and Tamai, 2009] are discussed in Section 5.3.1. Afterwards, Section 5.3.2 describes *RICA-J* [Serrano and Ossowski, 2004], a software framework designed for the development of *Java*-based multi-agent applications. Third, Section 5.3.3 elucidates *Object Teams / Java* [Herrmann, 2007]. A language extension for *Java* that, in contrast to all other RPLs, matured to a stable programming language providing a development environment, compiler, and debugger. Similar to *RICA-J*, *powerJava* [Baldoni et al., 2006c] is designed to bridge the gap between MAS and programming languages. However, *powerJava* is a programming language extension featuring both roles and institutions (Section 5.3.4). Last but not least, Section 5.3.5 presents *Scala Roles* [Pradel and Odersky, 2009], a small library that seamlessly adds both roles and collaborations to the *Scala* language.

5.3.1 EPSILONJ AND NEXTEJ

EpsilonJ is not only the oldest programming language included in this literature review, but also one of the few that managed to continuously provide new results throughout the years, i.e. [Ubayashi and Tamai, 2000, 2001, Tamai et al., 2005, 2007, Monpratarnchai and Tetsuo, 2011, Tamai and Monpratarnchai, 2014]. Ubayashi and Tamai [2000] first introduced *EpsilonJ* as a software framework for the flexible development of *cooperative mobile agent applications*. Their aim was to provide *"1) a mechanism for separating concerns about mobility/collaboration including traveling, task executions, coordination constraints, synchronization constraints, security-checking strategies and error-checking strategies; 2) a systematic and dynamically evolvable programming style."* [Ubayashi and Tamai, 2001, p.96]. Later on, Tamai et al. [2005] factored out the foundations of this framework into the language independent *Epsilon Model*. The *Epsilon Model* specifies collaborations both on the model and the programming level by means of *objects*, *actors*, *roles* and *environments* [Tamai et al., 2007]. Similar to conventional object models, objects are instances of a class that has properties, behavior, and a unique identity. However, an object can participant in an environment by playing one of the defined roles as an actor. Conversely, environments declare a set of collaborating role types. Roles, in turn, specify the collaboration-dependent properties and behaviors of their actors and can only be accessed from within their environment. By entering and leaving an environment, objects can dynamically acquire and lose roles, respectively. Based on this model, Tamai et al. [2005] reintroduced *EpsilonJ* as an extension to the *Java* programming language. Thus, the authors managed to overcome the verbosity of the original framework by providing both environments and roles as language construct. In particular, it permits the declaration of *environment types* with the keyword `context` and *role types* within an environment type using the keyword `role` [Tamai et al., 2005]. Moreover, role types do not declare the type of their player directly. Similar to *Rava's* requirements, each role type provides a set of methods that a possible player type must implement. Additionally, a role type can be specified as `static` to denote that only one object can play the corresponding role in an environment instance. By default, however, an environment instance can contain an arbitrary number of role instances. Moreover, the language features explicit operators for binding objects to roles with `newBind` as well as for lifting an object to its role in a given environment `(context.RoleType)object`. In sum, Monpratarnchai and Tetsuo [2008] presented the implementation of the *EpsilonJ* language accompanied with a proof-of-concept preprocessor that generates annotated *Java* source code.[5]

[5] `http://tamai-lab.ws.hosei.ac.jp/pub/epsilon/epsilonj/index.html`

Listing 5.8: Bank example implemented with NextEJ.

```
1  class Account { Money balance; /*...*/}
2  class Person  { String firstName; /*...*/}
3  class Company { String name; /*...*/}
4  context Transaction{ Money amount; /*...*/
5    static role Source requires { void decrease(Money amount); }{
6      void withdraw(){  decrease(amount) }
7    }
8    static role Target requires { void increase(Money amount); }{
9      void deposit(){ increase(amount) }
10   }
11   void execute(){ Source.withdraw(); Target.deposite()  }
12 }
13 context Bank{
14   role Consultant requires {/*...*/}{String phone; /*...*/}
15   role Customer   requires {String getName();}{/*...*/}
16   role Savings    requires {void decrease(Money amount);}{/*...*/}
17   role Checking   requires {void decrease(Money amount);}{/*...*/} }
18   static int main(String args[]){
19     final Transaction t1=new Transaction(100.0);
20     final Account a1=new Account(1000.0); /*...*/
21     bind a1 with bank1.Savings(), a2 with bank1.Checking(), /*...*/ {
22       bind a1 with t1.Source(), a2 with t1.Target() { t1.execute(); }
23     }
24   }
25 /*...*/}
```

More recently, Kamina and Tamai [2009] introduced **NextEJ** as an extension of *EpsilonJ* that incorporates common features of context-oriented programming, such as *multiple context activation* and *composite contexts*, to develop context-aware adaptive applications more easily. In particular, it introduces *context activation scopes* as lexical scopes of the program. In detail, they atomically create an environment instance, bind the corresponding roles to the given objects, and perform the enclosed operations. Afterwards the end of the scope is reached, the environment and all contained roles are deactivated. Consequently, the context-dependent behavior of roles is only active within its corresponding activation scope. Kamina and Tamai [2010] formalized *NextEJ's* core calculus *FEJ* and proved its type soundness, Even though, *FEJ's* syntax introduces 5 new keywords and 14 productions, neither of the authors have published a prototypical compiler for *NextEJ* yet. Despite that, *NextEJ* clearly improves several features of *EpsilonJ*. Hence, Listing 5.8, employs the syntax of *NextEJ* [Kamina and Tamai, 2009] to implement the running example. In detail, Line 1–3 defines the natural types `Account`, `Person`, and `Company`, as regular classes, whereas the compartment types `Transaction` and `Bank` are implemented as environments using the keyword `context`. In particular, the `Transaction` is defined in Line 4–12 and declares both `Source` and `Target` as `static` role types, whereas the former requires a `decrease` and the latter an `increase` method. Consequently, a transaction requires one source and one target role played by objects providing the required methods. In contrast, the `Bank` environment declares all roles without this *occurrence* restriction. Last but not least, the `main` method (Line 21–23) showcases two nested context activation scopes denoted with the keyword `bind`. The outer scope activates the `bank1` environment and binds, among others, the account `a1` to the `Savings` role and `a2` to the `Checking` role. The inner scope, shown in Line 22, activates a particular transaction `t1` tasked to transfer money from account `a1` to `a2`, which is finally executed. Please note, that this activation order ensures that the `Source` role invokes the `decrease` method of the `Savings` role before referring to the account's implementation. In sum, the implementation appropriately models the context-dependent behavior of the banking application.

Like the example, the evaluation focuses on *NextEJ* as successor of *EpsilonJ*, as they equally combine the behavioral and context-dependent nature of roles [Kamina and Tamai, 2009, 2010]. In case of the behavioral nature, *NextEJ* fulfills Feature 1, 3–5, 7–12, and 15 completely. By extension, *NextEJ* facilitates role types with properties, behavior, and individual identity. In general, objects can play multiple different roles simultaneously and a role can be played by unrelated objects and roles, as long as they provide the required methods. Moreover, a role can be transferred to another player using the `from` keyword in a bind statement [Kamina and Tamai, 2010]. Although, *NextEJ* handles roles as adjunct instance, it successfully hides this fact, within context activation scopes. Still, the necessary equality operators must be implemented to avoid *object schizophrenia* [Herrmann, 2010] and fulfill Feature 14. In case of the context-dependent nature of roles, *NextEJ* incorporates the Feature 19–20, 22, 24, and 26 by establishing environments as first-class citizens. In short, roles depend on environments as they are defined as inner classes. Regardless, environment instances are objects with properties, behavior, and their own identity. Additionally, environment instances can play roles, as well. However, the type system prevents that an environment can play a role in itself (Feature 23) [Kamina and Tamai, 2010]. Moreover, the modifier `static` for role declarations can be considered as *occurrence constraint*. Yet, as it only covers one case, Feature 27 is only partially fulfilled. Furthermore, when comparing *EpsilonJ* and *NextEJ*, it becomes apparent that *NextEJ* only adds nested environments (Feature 24) to the feature set of its predecessor. Besides that, it is possible to implement relationships as environments, partially satisfying Feature 2. However, such an environment must solely contain static roles and must be bound using *EpsilonJ's* `newBind` operator. It follows then, that the environments in *EpsilonJ* and *NextEJ* directly correspond to the notion of compartments and indirectly represent relationships. Consequently, *EpsilonJ* presents one of the earliest approaches to successfully introduce compartments to implement the context-dependent behavior of objects by means of roles. Since then, it has become an important research prototype for the investigation and application of role-based programming to the creation of *context-aware SAS* [Monpratarnchai and Tetsuo, 2011, Tamai and Monpratarnchai, 2014].

5.3.2 ROLE/INTERACTION/COMMUNICATIVE ACTION (RICA)

Similar to *EpsilonJ*, **RICA-J** [Serrano and Ossowski, 2004] is a framework for the development of MAS. It is based on the *Role/Interaction/Communicative Action (RICA)* theory [Serrano and Ossowski, 2004], a conceptual framework for the design of agent application that combines the aspects of *Agent Communication Languages* and *Organizational Models*. In essence, *RICA* provides the underlying metamodel for *RICA-J*. However, in contrast to *EpsilonJ*, it extends *Java* by means of an application library rather than a preprocessor. In this way, Serrano and Ossowski [2004] emphasizes, it was possible *"to reuse those middleware aspects required for supporting agent interactions, as well as the different agent abstractions provided"* [Serrano and Ossowski, 2004, p.97]. Specifically, *RICA-J* employs the *JADE* framework [Bellifemine et al., 1999] – a well-established *Java* framework for the design of MAS. It extends the framework by introducing *communicative actions*, *roles* and *interactions* as first-class citizens. By extension, the *RICA* metamodel (cf. [Serrano and Ossowski [2004],Fig. 1]) defines *agent types* that can play *role types*. *Role types*, in turn, declare a set of *action types* to denote those actions a player of this role can perform. In contrast to standard MAS metamodels, *RICA* further introduces *social* and *communicative interaction types* as main organizational abstraction. On the one hand, *social interactions* encapsulate the functionality, protocol, and actions performed by agents participating in this interaction, e.g. paper reviews, group meetings [Serrano et al., 2006]. Additionally, each *social interaction* specifies dedicated engagement, closing, joining, and leaving rules [Serrano and Ossowski, 2004] to declare when an interaction is initiated, finished, joined by an agent, and left by a participating agent.

Listing 5.9: Bank example implemented in RICA.

```
1  class Account extends Agent { Money balance; /*...*/}
2  class Person  extends Agent { String firstName; /*...*/}
3  class Company extends Agent { String name; /*...*/}
4  class Source  extends InteractiveRole {
5    static final CommRoleType type=new CommRoleType(Source.class);
6    static final CommInteractionType interaction=Transaction.type;
7    static final Agent player=Agent.type;
8    static final CommActType withdraw=/*...*/
9  /*...*/}
10 class Target  extends InteractiveRole {/*...*/}
11 abstract class Transaction extends CommunicativeInteraction{
12   static final CommInteractionType type=/*...*/
13   static final CommRoleType[] participants=
14      new CommRoleType[]{Source.type, Target.type};
15   Money amount;
16 /*...*/}
17 class FinancialTransaction extends Transaction{/*...*/}
18 class Customer   extends SocialRole {/*...*/}
19 class Consultant extends SocialRole {/*...*/}
20 class Banking extends SocialInteraction {/*...*/
21   static final CommRoleType[] participants=new CommRoleType[]{
22     Customer.type, Consultant.type, Savings.type, Checking.type };
23 /*...*/}
```

Accordingly, *RICA* facilitates both *social role types* and *social action types* to emphasize their dependence on a social interaction. On the other hand, Serrano and Ossowski [2004] motivates *communicative interactions* as reusable abstraction from *social interactions* [Serrano et al., 2006]. In fact, Serrano et al. argues that *"communicative interactions provide the pragmatic features of application-dependent social interactions, which basically differ at the semantic level"* [Serrano et al., 2006, p.107]. Simply put, they define *communicative interactions* as application-independent abstractions of *social interactions* that define the common participants and actions as *communicative role types* and *communicative action types*, respectively. However, communicative interactions omit application-dependent features as well as management rules. As a result, the *RICA* metamodel features agent types as natural types and interaction types as compartment types. [Serrano and Ossowski, 2004]. However, in order to integrate these concepts into a *Java* based framework, *RICA-J* separates the declaration of classes from the declaration of metaclasses. Similar to the distinction between `Object` and `Class`, *RICA-J* provides a base class for each instance, e.g. `SocialInteraction`, `SocialRole`, and `SocialAction`, as well as a class representing the corresponding type, for instance, `SocialInteractionType`, `SocialRoleType`, and `SocialActionType`. However, in contrast to *Java*, the application developer must manually provide the corresponding metamodel definitions as `public static` fields. Moreover, as `Actions` are the main abstraction of behavior, each operation must be specified in an individual class. Furthermore, the order of execution of these actions is controlled by a `Protocol` associated to each `Interaction`. Notably though, an `Agent` does not provide any `Action` on its own, rather than its behavior is solely determined by the actions provided by the roles it is currently playing.

Although this model directly mirrors the *RICA* metamodel, it is very cumbersome to declare the numerous entities of a concrete social application, such as the banking application. Thus, for the sake of brevity, Listing 5.9 excludes the definition of the individual `Action` classes and the management code required to implement a particular type. In short, the example shows the definition of social and communicative interactions and their corresponding role types. In general, the framework provides individual base classes for each kind of the metamodel. The declaration of the `Source` role (Line 4–9), for instance, extends the `InteractiveRole` class provided by *RICA-J* framework

for communicative role types. As mentioned above, the `Source` class must contain the specification of its `type`, the type of its `player`, the type of the communicative interaction it belongs to, and the communicative actions it provides to its players. Conversely, the `Transaction` class is modeled as a communicative interaction (Line 11–16) by extending the `CommunicativeInteraction`, declaring its type, and specifying the types of its participants, i.e. `Source` and `Target`. Due to the fact that the `Transaction` is defined as a reusable interaction, it must be further refined to the `FinancialTransaction` (Line 17) by specifying the engagement, closing, joining, and leaving rules, not shown in the listing. In contrast, the role types `Consultant`, `Customer`, `Savings` and `Checking` are all defined as subclass `SocialRole` that all participate in the social interaction `Banking`. Admittedly, this example hides several fundamental aspects of an actual implementation of the banking application, for instance, the implementation of the `shouldBePlayed` method of role types or the protocol of the `Transaction` interaction. However, these aspects are buried in *Java* source code and hence, not part of the implemented domain model. Thus, while *RICA-J* provides an unconventional approach to define role-based applications, it captures the context-dependent or rather organizational aspects of roles by introducing *social interactions*, *social roles*, and *social actions*. To put it succinctly, the *RICA-J* framework supports the behavioral and context-dependent nature of roles [Serrano and Ossowski, 2004]. In case of the behavioral nature, it supports Feature 1, 3, 5, 7, 10–13, and 15 completely.

In general, both social and communicative role types have properties, associated actions, and a super type. Moreover, when instantiated, roles carry their own identity and can be played by unrelated agents or other roles. Agents, in turn, acquire and abandon roles dynamically whenever they join or leave an interaction. Specifically, an agent can play multiple different roles simultaneously. Besides that, *RICA-J* only partially allows for restricting the sequence of role acquisition and removal (Feature 6), as it must be implemented in both the `shouldBePlayed` method of the role type and the joining respectively leaving rule of the corresponding interaction. Nonetheless, it can be argued that *RICA-J* also incorporates Feature 19–20, 25 and 26. Especially, when considering *social interactions* as compartments. In general, *social roles* depend on the *social interaction* they participate in [Serrano and Ossowski, 2004] Moreover, the state of *social interactions* is stored in its parameters, whereas its behavior is specified in the protocol controlling the execution of its participants [Serrano et al., 2006]. Furthermore, *social interaction* implemented as *Java* classes automatically enable (single) inheritance and guarantee a unique identity. In sum, *RICA-J* manages to augment the typical notion of roles in MAS with the notion of social interactions to model the various organization- or collaboration-dependent behavior typically found in social application domains. Although it is true that the *RICA* theory adequately models the behavioral and context-dependent behavior of roles, the *RICA-J* framework imposes so much implementation overhead that it becomes infeasible to implement even small application. Granted, providing a custom language to specify *RICA* models and a compiler to generate the corresponding source code, would solve this problem immediately.

5.3.3 OBJECTTEAMS/JAVA

In contrast to all other RPLs, **ObjectTeams/Java (OT/J)** is the only RPL that matured from a research prototype [Herrmann, 2002] to a featured extension of the *Eclipse integrated development environment (IDE)*.[6] As the name suggests, it is a heavy-weight extension to *Java* that introduces the notion of *teams* to capture context- and collaboration-dependent behavior of objects. Specifically, it adds 16 keywords and 33 grammar rules to the *Java* syntax [Herrmann and Hundt, 2013].

[6] `http://www.eclipse.org/objectteams`

Listing 5.10: Bank example implemented in Object Teams/Java.

```
1  class Account { Money balance; /*...*/}
2  class Person  { String firstName; /*...*/}
3  class Company { String name; /*...*/}
4  team class Transaction{ Money amount; /*...*/
5    class Source playedBy Account
6      when base( ! hasRole(base,Target) ) {withdraw -> decrease}
7    class Target playedBy Account
8      when base( ! hasRole(base,Source) ) {deposit -> increase}
9    boolean execute(Account as Source f,Account as Target t){
10       /*...*/ f.withdraw(amount); t.deposit(amount); /*...*/
11     }
12 }
13 team class Bank{
14   String name; /*...*/
15   class Checking playedBy Account{/*...*/
16     callin void limited(Money a){ if(a<limit) base.decrease(a); }
17     void limited(Money a) <- replace decrease(Money a);
18   }
19   class Savings playedBy Account{/*...*/ }
20   class Consultant playedBy Person{/*...*/}
21   abstract class Customer{/*...*/}
22   class PersonCustomer  extends Customer playedBy Person{/*...*/}
23   class CompanyCustomer extends Customer playedBy Company{/*...*/}
24   class BankCustomer    extends Customer playedBy Bank{/*...*/}
25 /*...*/}
```

Accordingly, Herrmann [2005] had to implement a custom compiler that translates an *OT/J* program directly to *Java*-bytecode. However, unlike other language extensions, he employs an *aspect weaver* to augment existing classes, e.g. legacy or third-party classes. Consequently, *OT/J* allows for adapting any *Java* application and library with roles. Conversely, the *role types*, their behavior and *player type* must be declared within a *team*. In detail, the `team` keyword declares a special class, whose inner classes become role declarations if their declaration contains the `playedBy` clause. This clause, in turn, denotes exactly one class whose instances can play this role type. Within role declarations, the language supports the declaration of *callin* and *callout* bindings. Comparable to *constituent methods* [Graversen and Østerbye, 2003] or *AOP advices*, *callin* methods can be declared to be invoked `before`, `after` or instead (keyword `replace`) of a given player method. In contrast, *callout* methods are directly forwarded to the player object. Moreover, *OT/J* permits the definition of *guard predicates* using the keyword `when` to restrict the activation of teams, roles, and `callin` methods. Although both role and team declarations can inherit properties and behavior from a super class, a class cannot inherit from a role declaration. This ensures that a role declaration is confined within the team declaration and only accessible within the corresponding team [Herrmann, 2013]. At runtime, roles are implicitly created and bound whenever an object is passed to a method of an active team. In detail, *OT/J* employs lifting [Herrmann, 2007] to detect whether the object already plays the requested role or to instantiate and bind a suitable role type. Admittedly, these language constructs represent a small fraction of the *OT/J* language definition [Herrmann and Hundt, 2013].

Nonetheless, they are sufficient to model the running example. After all, Listing 5.10 outlines the implementation of the banking application. Basically, the natural types `Account`, `Person` and `Company` are declared as regular *Java* classes (Line 1–3), whereas the compartment types `Bank` and `Transaction` are specified as teams. Within each of the `team` declarations, the corresponding role types have been declared. On the one hand, `Transaction` defines the two role types `Source` and `Target` played by `Account` that declare a `callout` binding from `withdraw` to `decrease` and from `deposit` to `increase`, respectively. Moreover, both role types specify a guard predicate with `when`

to prohibit a player of the `Source` role to assume a `Target` role in the same `Transaction` instance, and vice versa. Finally, the `execute` method implicitly creates and binds the given `Account` objects as `Source` and `Target` before performing the actual money transferal. On the other hand, the `Bank` declaration comprises the role declarations for `Savings`, `Checking`, `Consultant` and a subclasses of `Customer` for each player type. Because a role declaration can only specify one player type, the example defines a common base class `Customer` that is extended by three role types `PersonCustomer`, `CompanyCustomer`, and `BankCustomer`, which are played by the corresponding class (Line 21–23). Last but not least, the definition of the `Savings` role type, for instance, demonstrates the use of callin methods, as it redirects any invocation of the `decrease` method to the callin method `limited` (Line 15 and 16). In conclusion, *OT/J* is able to fully capture the context-dependence and dynamics of the banking application.

Moreover, it successfully combines the behavioral and context-dependent nature of roles in an unprecedented way. As evaluated by Herrmann [2007], *Object Teams* fully supports Feature 1, 3–6, 8, 10–13, 15 and only Feature 2 and 14 partially.[7] In the same way, *OT/J* also satisfies most of the contextual features of roles, i.e. Feature 19–20 and 22–26. Indeed, roles can only be declared and exist within a team declaration and corresponding instance (Feature 19). Besides that, teams resemble objects in that they have properties, behavior, and their own identity. Moreover, they can inherit from another class and play roles themselves (including its own). Despite all that, *OT/J* only partially supports *occurrence constraints* (Feature 27). Granted, they could be implemented using guard predicates, however, the compiler would not enforce the lower bounds. As a result, *OT/J* fulfills more features of roles than any other RPL considered in this literature review. Admittedly, it does not provide a formalized metamodel, type system, or operational semantics like other role-based languages, yet a real programming language has the additional benefit of practical applicability [Herrmann, 2007]. In particular, Herrmann emphasizes that *"[his] definition of roles can directly be validated by writing programs in ObjectTeams/Java and evaluating the results"* [Herrmann, 2007, p.181]. In essence, Herrmann argues that a practical implementation of the role concept is equally suitable for its formal specification, especially, when the language is open source and publicly available via the *Eclipse Foundation*.[8] Though I concede that a practical implementation is crucial to show the applicability of roles, I still insist that only a formalization of the role concept enables researchers to investigate and improve different variants of the role concept without being tied to one implementation.

5.3.4 POWERJAVA

Like *EpsilonJ* and *RICA-J*, **powerJava** was developed by Baldoni et al. [2006a] to bridge the gap between agent theory and its object-oriented implementation. Specifically, Baldoni et al. [2006a] believes that *"introducing theoretically attractive agent concepts in a widely used language can contribute to the success of the Autonomous Agents and Multiagent Systems research in other fields"* [Baldoni et al., 2006a, p.73]. Consequently, Baldoni et al. designed *powerJava* as a lightweight extension to *Java* that introduces *social roles* and *institutions* as language constructs [Baldoni et al., 2006a]. In general, Baldoni et al. [2006c] establish *roles* as entities that are existentially dependent on an *institution* and able to grant powers to their players, i.e. by providing methods that permit access to the private state of the institution, as well as participating roles. To achieve this, *powerJava* augments *Java* by adding 5 new keywords within 7 grammar rules [Baldoni et al., 2007]. In contrast to other RPLs, however, *powerJava* separates the declaration of role types into an interface specification (independent of an *institution*) and a definition within a specific *institution*.

[7] In fact, *OT/J* suffers from *object schizophrenia*, although it can be easily cured [Herrmann, 2010].
[8] `http://git.eclipse.org/c/objectteams/org.eclipse.objectteams.git`

Listing 5.11: Bank example implemented in powerJava.

```
1  class Account   { Money balance; /*...*/}
2  interface Party { String getName(); }
3  class Person  implements Party  {String firstName; /*...*/ }
4  class Company implements Party  {String name; /*...*/ }
5  role Source playedby Account{ void withdraw(Money amount); }
6  role Target playedby Account{ void deposit(Money amount); }
7  class Transaction{ Money amount;
8    definerole S realizes Source{/*...*/}
9    definerole T realizes Target{
10     void deposit(Money amount){that.increase(a); }
11   }
12   boolean execute(Account f, Account t){
13     this.new S(f); this.new T(t); ((this.S) f).withdraw(a); /*...*/
14   }
15 }
16 role Customer   playedby Party  {/*...*/}
17 role Consultant playedby Person {/*...*/}
18 role Checking   playedby Account{ void decrease(Money amount); }
19 role Savings    playedby Account{ void decrease(Money amount); }
20 class Bank    implements Party  { String name; /*...*/
21   definerole Con realizes Consultant{String phone; /*...*/}
22   definerole Cus realizes Customer  {List<CA> cas; /*...*/}
23   definerole CA realizes Checking   {Customer owner; /*...*/}
24   definerole SA realizes Savings    {/*...*/}
25 /*...*/}
```

While the prefix `role` specifies the methods (or powers) a role provides to the `playedby` class, `definerole` and `realize` denote its implementation within an institution, granting its player access to the institution's private members. Additionally, *powerJava* introduces the notion of a *role cast* as additional expression of the form `(institution.RoleType)object` to retrieve a specific role, a given object plays in an institution. Conversely, the keyword `that` is available within role implementations to refer to its current player. Regardless, the language does not introduce a keyword for institutions. Hence, any class containing role definition implicitly declares an institution. Technically, *powerJava* utilizes a preprocessor implemented in *Java* that reads *powerJava* source code and generates plain *Java* source code [Arnaudo et al., 2007] that is still publicly available.[9] Notably though, it injects the interface `ObjectWithRoles` and the corresponding management methods into each class able to play a role. In addition, the preprocessor translates role definitions to inner classes of the corresponding institution. *Java's* type system, in turn, ensures the existential dependence of role instances and grants access to the private members of the outer class. In conclusion, Baldoni et al. demonstrate that *powerJava* is sufficient to implement MAS and, more generally, role-based applications [Baldoni et al., 2006c,b, 2007].

To illustrate this, Listing 5.11 sketches the implementation of the banking application. The natural types `Account`, `Person`, and `Company` are implemented as regular *Java* classes (Line 1–4). Conversely, the role types `Source` and `Target` are first specified to be played by `Account` objects and to provide the methods `withdraw` respectively `deposit`. Afterwards, both are implemented as `S` and `T` within the `Transaction` institution. Specifically, the implementation of `deposit` utilizes `that` to refer to its player, in Line 10. Moreover, the `execute` method of `Transaction` first binds new role instances of `S` and `T` to the accounts `f` and `t`, respectively. Next, the role cast `(this.S)f` retrieves the newly bound `Source` role type and invokes the `withdraw` method. Likewise, the `Bank` institution is implemented by defining and implementing the role types `Customer`, `Consultant`, `Savings`, and `Checking`. Notably though, the `Customer` role type is played by the interface `Party`.

[9] `http://www.di.unito.it/~guido/powerJava.zip`

Hence, persons, companies, and banks can play the `Customer` role, however, the role implementation `Cus` can only refer to methods defined by the `Party` interface. In sum, *powerJava* presents a simple way to implement the collaboration- and context-dependent dynamics of the banking domain. By extension, it manages to combine the behavioral and context-dependent nature of roles [Baldoni et al., 2006c], as it supports the corresponding features. On the one hand, *powerJava* fulfills Feature 1, 3–5, 8–13, and 15 by introducing roles as objects with properties, behavior, inheritance, and unique identity. Objects can play multiple roles at the same time including roles of the same type, as long as each instance is tight to another institution. However, unrelated objects cannot play the same role type (Feature 7), as at least a common interface is required. Likewise, *powerJava* does not provide additional means to constrain roles (Feature 6). Nonetheless, role constraints could be implemented easily by modifying the constructor of each role implementation. On the other hand, *powerJava's* institutions satisfy the contextual Feature 19–20, 22, 24, and 26. In general, institutions are represented as classes with roles as inner classes. Thus, institutions can have properties, behavior, and their own identity. Furthermore, nothing within *powerJava* prevents the definition of institutions that contain another institution as inner class, which fulfills Feature 22. Regardless, it only partially satisfies Feature 21, as only the interface defined by the `role` keyword can be shared among institutions. Likewise, *powerJava* only lacks support for institution inheritance, because *Java* does not directly support family polymorphism [Ernst, 2001]. Finally, Baldoni et al. [2010] illustrate how relationships can be implemented with *institutions* and thus showing that it partially fulfills Feature 2. Granted, *powerJava* lacks the support of *constituent methods* [Graversen and Østerbye, 2002] or *true delegation* [Herrmann, 2005], yet it successfully introduces both roles and institutions solely relying on features already provided by *Java*.

5.3.5 SCALA ROLES

As a more recent contextual RPL, **Scala Roles** introduces both *roles* and *collaborations*, to enable the dynamic adaptation of objects as well as the description and reuse of object collaborations. Specifically, Pradel and Odersky [2009] introduce *dynamic collaborations* to encapsulate the interrelations of objects in a certain context. Moreover, *collaborations* contain a set of *roles* that specify the context-dependent behavior of objects [Pradel and Odersky, 2009]. In particular, collaborations declare the behavior by means of roles that are later played by objects. As a result, a *collaboration* is an object-level representation of the context-dependent behavior of collaborating objects that can be individually instantiated and reused. Notably, most RPLs discussed so far chose *Java* as their host language. However, *Java's* lack of advanced language features, e.g. traits, dynamic proxies, and implicit conversion, forced many researchers to implement custom compilers or preprocessors hiding the required management code. To avoid this problem, Pradel and Odersky [2009] decided to implement *Scala Roles* as an application library for the *Scala* programming language.[10] In this way, the authors emphasize that *"the underlying programming language need not be changed and existing tools like compilers can be used without modifications"* [Pradel and Odersky, 2009, p.27]. In fact, *Scala Roles* mainly relies on two language features of *Scala*. First, the ability to dynamically generate *proxies* for player objects. Second, the notion of *nested types* to implement collaboration types containing *role types*. Besides that, *Scala Roles* employs *implicit conversions* and *dependent method types* to provide a more convenient syntax [Pradel and Odersky, 2009].

[10] `https://www.scala-lang.org`

Listing 5.12: Bank example implemented with Scala Roles.

```
1  class Account(val id:Int, var balance:Money){/*...*/}
2  class Person(val firstName:String, /*...*/){/*...*/}
3  class Company(val name:String, /*...*/){/*...*/}
4  trait Transaction(val amount: Money) extends Collaboration{
5    val source=new Source(); val target=new Target()
6    trait Source extends Role[Account]{
7      def withdraw=core.decrease(amount)
8    }
9    trait Target extends Role[Account]{def deposit=/*...*/}
10   def execute={source.withdraw(); target.deposit()}
11 /*...*/}
12 trait Bank extends Collaboration{
13   val customers   = new HashMap[Integer,Customer]()
14   val consultants = new HashMap[String, Consultant]()
15   /*...*/
16   trait Consultant(val phone:String) extends Role[Person]{/*...*/}
17   trait Checking extends Role[Account]{var limit:Money /*...*/}
18   trait Savings  extends Role[Account]{var fee:Double /*...*/}
19   trait Customer extends Role[AnyRef] {var name:String /*...*/}
20   def transferal(val s:Account, val a:Money, val t:Account)={
21     val tr=new Transaction(a); (s as tr.source); (t as tr.target);
22     return tr.execute();
23   }
24 /*...*/}
```

Granted, a full-fledged compiler, e.g. *OT/J*, is able to perform type checking and optimizations on a role-based program, yet, a language extension provided as an application library is easier to deploy by practitioners and more amenable to investigation by researcher. This holds true, in particular, as *Scala Roles* is open source, publicly available and still working.[11]

Thus, the implementation of the running example not only demonstrates the use of *Scala Roles*, but can actually be executed. As an illustration, Listing 5.12 shows an excerpt of the implemented banking application. In general, all natural types are implemented as regular classes, such as `Account`, `Person`, and `Company` (Line 1–3), whereas the compartment types `Bank` and `Transaction` are implemented as `trait`. Traits are types defined by a set of methods including their implementation that, in contrast to classes, can be partially defined [Pradel and Odersky, 2009]. In the *Scala Roles* library, traits are used to declare both collaborations and roles. The `Transaction` collaboration, for instance, is defined as a trait extending the `Collaboration` trait provided by the library. Likewise, two role types `Source` and `Target` inherit from the `Role` trait. However, they are declared as inner traits of the `Transaction` collaboration and provide the `withdraw` and `deposit` methods, respectively. In this way, roles have access to the private members of their collaboration and their player (using the `core` reference). Finally, the `execute` method specifies how the money transferal is performed by referring to the role instances `source` and `target` of the `Transaction` collaboration (Line 10). The `Bank` collaboration is defined similarly in Line 12–24, however it manages multiple `Customer` and `Consultant` instances with a `HashMap`. After specifying a collaboration, it can be instantiated like an ordinary object, and objects can be bound to roles contained in this collaboration using the `as` expression. In Line 21, for instance, the `as` operator is used to bind the accounts `s` and `t` to the role instances `source` and `target` of the transaction `tr`. Technically, the `as` operator creates an *implicit proxy* around a player and its roles that is able to dispatch method invocations to the responsible instance [Pradel and Odersky, 2009].[12] Indeed, the created proxy establishes a *com-*

[11] `https://github.com/tupshin/Scala-Roles`

[12] On a closer examination it occurred that the `as` operator only binds roles within one statement. To overcome this limitation, the library provides the `StickyCollaboration` trait to explicitly assign, retain, and remove roles.

pound object that hides the fact that objects and its roles are distinct objects. In addition, Pradel and Odersky [2009] implemented the equality operator of these compound objects to ultimately prevent object schizophrenia [Pradel and Odersky, 2009, Sec. 3.1]. As such, this implementation appropriately captures the context-dependent collaborations of the banking application.

Accordingly, *Scala Roles* supports the behavioral and context-dependent nature of roles. In case of the behavioral nature, Pradel and Odersky [2009] already evaluated their role concept with respect to *Steimann's features of roles* and concluded that it fully satisfies Feature 1, 3–5, and 7–15 [Pradel and Odersky, 2009]. Notably, only Feature 6 is not supported, as *Scala Roles* does not constrain which roles can play which objects. Moreover, *Scala Roles* also incorporates a considerable amount of the *additional features of roles*. Specifically, the notion of dynamic collaborations fulfills Feature 19, 20, 22, and 24–26. Basically, roles depend on collaborations both on the type- and the instance-level. Moreover, collaborations are traits with properties, behavior, and multiple inheritance that can be instantiated to objects. Thus like normal objects, they carry identity and can play roles. Regardless, using *Scala Roles* collaborations cannot play roles that are part of themselves, as this would create recursive types not allowed by the *Scala* type system. Besides all that, one could argue that collaborations could be refined to represent relationships (Feature 2). Unfortunately, this extension has not been investigated further by the authors. Nonetheless, *Scala Roles* is one of the few RPLs able to seamlessly combine the context-dependent and behavioral nature of roles without requiring a custom compiler. In fact, it utilizes advanced language features of *Scala*, i.e. *dynamic proxies*, *type nesting*, *implicit conversions*, and *dependent method types* [Pradel and Odersky, 2009], to establish a seamless language extension. In conclusion, Pradel and Odersky [2009] presented a very small implementation of roles that could be easily adopted and extended by both researchers and practitioners. Designed in this way, one would assume that *Scala Roles* is a successful RPL by now. On the contrary, it was barely recognized by later researchers, except for the fellows at the RoSI research training group.

"All definitions of roles discussed here have their merits and drawbacks."

— Steimann [2000b]

6 COMPARISON OF ROLE-BASED LANGUAGES

Undoubtedly, all the 26 role-based modeling and programming languages investigated during this SLR have their use cases. However, Steimann [2000b] would still be right today contending that all the definitions of roles have their benefits and shortcomings. Arguably, this could be said about almost everything. Nonetheless, it misses the point that all these languages share a common notion, and thus should be treated as a family of languages rather than individual approaches. Consequently, this chapter provides a broader perspective on both the RMLs and RPLs by directly comparing them utilizing the classification scheme established in Chapter 2.6. In fact, this comparison will provide insight into the research fields and practical applicability of role-based modeling and programming languages. In particular, the discussion focuses on the commonalities, differences, and chronological relations of the various approaches, to uncover the structure of the research fields. Furthermore, the summary also compares the tool support available for each language, in order to assess the practicality of the contemporary approaches. In conclusion, this chapter summarizes the results of the conducted SLR by presenting and interpreting the results of the comparison. The chapter is structured, accordingly. First, Section 6.1 compares the contemporary RMLs by means of the supported features of roles, provided graphical notation, and published modeling editor. Conversely, Section 6.2 discusses the contemporary RPLs comparing the established features of roles, the syntactic complexity, and the available tool support. Finally, Section 6.3 concludes the SLR by answering the initial research questions and emphasizing the problems identified in the research field on role-based modeling and programming languages. The majority of the presented comparison and its results have been published in [Kühn et al., 2014, Kühn et al., 2015a, Kühn and Cazzola, 2016, Kühn et al., 2016].

6.1 COMPARISON OF ROLE-BASED MODELING LANGUAGES

Since Chapter 4 individually described each contemporary RML, this section provides an overview on these modeling languages by evaluating the features of roles each of them supports. Granted, these languages differ in their application domain, yet all provide a sufficient definition for roles to apply the classification scheme. This, in turn, allows for directly comparing the contemporary modeling languages regardless of their scope or application domain. This comparison provides some indication whether there exists relations between approaches and whether researchers continuously improved the notion of roles, e.g., by extending existing RMLs.

Table 6.1 aligns the various contemporary RMLs in chronological order, as well as their corresponding classification with respect to the 27 features of roles. In accordance to the evaluations in Chapter 4, each feature can be either fulfilled (■), possible to fulfill (⊞) or not fulfilled (□). In fact, some modeling languages lack operational semantics rendering some features not applicable (∅), i.e., Feature 5, 9, 12 that require dynamic binding, transferring, and accessing role instances. At a first glance, the reviewed research field appears to advance over the years, as the number of features supported by each RML increased. On closer examination, however, the table indicates that the research field suffers from fragmentation and discontinuity. Specifically, *fragmentation* denotes that each modeling approach focuses solely on a specific use case or application domain and does not take previous or related results into consideration [Kühn et al., 2014]. In fact, the conducted SLR indicates that most approaches were unaware of or reluctant to apply Steimann's *features of roles* [Steimann, 2000b] to classify their notion of roles. To take a case in point, only half of the RMLs referenced Steimann [2000b], i.e. [Dahchour et al., 2002, Kim et al., 2003, Zhu and Zhou, 2006, Genovese, 2007, Liu and Hu, 2009a, Hennicker and Klarl, 2014], only few actually utilized his classification scheme, i.e. [Kim et al., 2003, Zhu and Zhou, 2006]. Granted, this might be the result of the diversity in the research community, yet I would argue that negligence is the main reason for the apparent fragmentation in this research field. In contrast, *discontinuity*, emphasizes that each RML defines the role concept reusing and augmenting previous definitions [Kühn et al., 2014]. To put it bluntly, none of the investigated approaches reused either formal models or metamodels as basis for their approach. Similarly, solutions to the representation of roles were not carried one from one approach to subsequent ones, but were just reinvented. As an illustration, consider the feature rich notion of roles established by the TAO framework [Da Silva et al., 2003]. In spite of its early proposal in 2003, it has neither been utilized to define the similar INM nor extended to define subsequent modeling languages. Even though the number of features increased in the past 16 years, however, there has been no continuous improvement or combination of previously proposed role-based modeling languages. Furthermore, because the various RMLs neither share a common underlying model nor a common understanding of roles, researcher are unable to combine the behavioral, relational, and context-dependent modeling languages. Evidently, the research field suffers from fragmentation and discontinuity [Kühn et al., 2014].

Although the research field's condition is important for researchers, it hardly tells anything about the practical applicability of contemporary RMLs. Henceforth, the discussion focuses on those modeling languages that provide a graphical modeling editor and/or introduce a graphical notation. Up to my best knowledge, *ORM 2* [Halpin, 2005] and *OntoUML* [Benevides and Guizzardi, 2009] are the only RMLs that provide a dedicated graphical editor that is publicly available. However, they only support the relational and behavioral nature of roles. Besides that, most contemporary RMLs introduce a graphical notation for roles by either providing their own notation or utilizing UML. On the one hand, both *ORM 2* [Halpin, 2005] and *INM* [Liu and Hu, 2009a] propose new notations based on ER. While *ORM 2* provides a very concise notation for relational roles featuring a vast amount of modeling constraints, *INM's* notation is cluttered mixing entities, context-dependent roles, and multiple kinds of relations [Kühn et al., 2016]. On the other hand, the following modeling languages extend UML and/or employ UML stereotypes to establish their graphical notation. Specifically, the *revised UML* [Steimann, 2000c] and the *Generic Role Model* [Dahchour et al., 2002] both extend UML class diagrams. While Steimann [2000c] adds roles, relationships, and *fills* relations, Dahchour et al. [2002] only introduces the *fills*-relation as a special inheritance relation. Accordingly, *RBML* [Kim et al., 2002], *OntoUML* [Guizzardi and Wagner, 2012], and the *Helena Approach* [Hennicker and Klarl, 2014] define dedicated UML stereotypes to distinguish between the various model elements, e.g. natural types, role types, and ensemble types.

Table 6.1: Comparison of role-based modeling languages, extended from [Kühn et al., 2014]

Features [Kühn et al., 2014]	**Lodwick** [Steimann, 2000b]	**Generic Role Model** [Dahchour et al., 2002]	**TAO** [Da Silva et al., 2003]	**RBML** [Kim et al., 2003]	**Role-Based Patterns** [Kim and Carrington, 2004]	**ORM 2** [Halpin, 2005]	**E-CARGO** [Zhu and Zhou, 2006]	**Metamodel for Roles** [Genovese, 2007]	**INM** [Liu and Hu, 2009a]	**DCI** [Reenskaug and Coplien, 2009]	**OntoUML** [Guizzardi and Wagner, 2012]	**Helena Approach** [Hennicker and Klarl, 2014]
1	■	■	■	■	■	⊞	⊞	■	■	■	■	■
2	■	□	■	■	■	■	□	⊞	■	⊞	■	■
3	■	■	■	■	■	■	■	⊞	■	■	⊞	■
4	■	■	■	■	□	■	■	■	■	□	∅	■
5	■	■	∅	∅	∅	■	■	■	∅	⊞	∅	■
6	⊞	■	∅	∅	■	⊞	⊞	∅	□	□	□	□
7	■	□	■	□	□	□	⊞	■	□	■	⊞	■
8	□	■	□	□	□	□	□	■	■	□	■	□
9	⊞	□	⊞	∅	∅	∅	■	⊞	∅	□	∅	□
10	⊞	■	■	∅	□	∅	⊞	⊞	■	■	■	■
11	⊞	■	■	□	■	□	□	⊞	■	■	■	■
12	∅	⊞	⊞	■	∅	∅	■	∅	∅	■	∅	■
13	⊞	■	■	■	■	□	□	■	■	⊞	⊞	□
14	■	□	□	■	■	■	■	⊞	■	⊞	■	□
15	□	■	■	□	□	□	⊞	⊞	□	⊞	□	■
16	■	□	■	■	■	■	□	□	□	□	■	□
17	□	□	□	□	□	■	□	□	□	□	■	□
18	□	□	□	□	□	□	⊞	□	□	□	□	□
19	⊞	□	■	□	□	⊞	■	■	■	■	⊞	■
20	□	□	■	□	□	⊞	□	■	■	■	■	■
21	⊞	□	□	□	□	□	■	■	⊞	□	■	■
22	□	□	■	□	□	■	□	■	■	■	□	□
23	□	□	□	□	□	□	□	■	□	□	□	□
24	□	□	■	□	□	□	□	□	■	□	□	□
25	□	□	■	□	□	□	□	■	■	⊞	□	□
26	□	□	■	□	□	■	■	⊞	■	■	□	■
27	□	□	□	■	■	□	■	⊞	□	□	⊞	■

■: yes, ⊞: possible, □: no, ∅: not applicable

Nonetheless, each of them needed to extend UML, for instance, to add *occurrence constraints* or additional *formal relations*. While the above languages are usable with any UML editor, they are hard to comprehend as they require reading the stereotype annotations [Moody, 2009]. In conclusion, due to the discontinuity and the fragmentation of the research field, only few RMLs published since 2000 matured to be extensible and applicable by researchers and practitioners, alike.

6.2 COMPARISON OF ROLE-BASED PROGRAMMING LANGUAGES

After comparing the contemporary RMLs, this section provides a similar comparison of the contemporary RPL to assess the structure and progress in this research field. In addition to that, the discussion focuses on the practical applicability and extensibility of the various by taking their syntactic complexity, host language, and compiler availability into account. Ultimately, this comparison indicates whether the research field on RPLs also suffers from fragmentation and discontinuity and whether RPLs can be employed in practice.

Hence, Table 6.2 shows the corresponding comparison of the thirteen contemporary RPLs with respect to the 27 features of roles. In contrast to modeling languages, all features are applicable, as programming languages operate on both the model- and instance-level. Accordingly, most RPLs treat roles as objects featuring their own properties and behavior (Feature 1). The only exception is the data programming language *Relations* [Harkes and Visser, 2014], which lacks operations to dynamically bind or unbind roles rendering Feature 5 and 12 not applicable. Nonetheless, the table generally evinces the presence of fragmentation and discontinuity in the research field on RPLs. In case of *fragmentation*, the diversity of the application domains and features supported by the various RPLs already hints that only few approach take previous modeling or programming languages into account. Specifically, the literature review elucidated that only one third (4 of 13) of the programming languages referred to Steimann's unifying definition of roles [Steimann, 2000b], i.e. *OT/J* [Herrmann, 2005], *powerJava* [Boella and Van Der Torre, 2007], *Scala Roles* [Pradel and Odersky, 2009], and *JavaStage* [Barbosa and Aguiar, 2012]. Nevertheless, three of them (except *JavaStage*) applied his features to delineate their approach. Yet, most approaches published since 2005 at least reference *OT/J* as related work, except for *Rava* [He et al., 2006] and *Relations* [Harkes and Visser, 2014]. In general, however, most approaches considered only a limited number of RPLs as related approaches, usually excluding RMLs. Clearly, this effect cannot only be attributed to the diversity of the application domains.[1] Conversely, *discontinuity* becomes apparent when regarding the improvements in the research field. Generally, programming languages included more features over time. However, if one considers that *OT/J* [Herrmann, 2005] already managed to support most features of roles by the year 2005, it is rather surprising, that none of the subsequent RPL were able exceed its number of features. Even though *Scala Roles* [Pradel and Odersky, 2009] came close to *OT/J*, both languages have not been employed and/or improved by more recent approaches. Moreover, each language provides its own syntax for the definition of roles, compartments, and relationships without using a common terminology or common semantics [Kühn and Cazzola, 2016]. Basically, the various languages reimplemented the required language constructs, such as a *role definition*, both syntactically and semantically. Furthermore, most of the contemporary RPLs were only explored in up to three subsequent publications and then abandoned completely. In fact, only *EpsilonJ* [Ubayashi and Tamai, 2001] and *OT/J* [Herrmann, 2005] have been continuously utilized, extended or improved by the authors. As a result, most of the research field on RPLs suffers from fragmentation and discontinuity.

[1] Consider Barbosa and Aguiar [2012] claim that *"[this] is the first language to tackle the static role use"* [Barbosa and Aguiar, 2012, p.143]

Table 6.2: Comparison of role-based programming languages, extended from [Kühn et al., 2014]

Features [Kühn et al., 2014]	**EpsilonJ** [Ubayashi and Tamai, 2001]	**Chameleon** [Graversen and Østerbye, 2003]	**RICA-J** [Serrano and Ossowski, 2004]	**JAWIRO** [Selçuk and Erdoğan, 2004]	**OT/J** [Herrmann, 2005]	**Rava** [He et al., 2006]	**powerJava** [Baldoni et al., 2006c]	**Rumer** [Balzer et al., 2007]	**First-Class Relationships** [Nelson et al., 2008]	**Scala Roles** [Pradel and Odersky, 2009]	**NextEJ** [Kamina and Tamai, 2009]	**JavaStage** [Barbosa and Aguiar, 2012]	**Relations** [Harkes and Visser, 2014]
1	■	■	■	■	■	■	■	■	■	■	■	■	□
2	⊞	□	□	□	⊞	□	⊞	■	■	⊞	⊞	□	■
3	■	■	■	■	■	■	■	■	■	■	■	■	■
4	■	■	□	■	■	□	■	■	■	■	■	⊞	■
5	■	■	■	■	■	■	■	■	■	■	■	□	∅
6	□	□	⊞	⊞	■	□	⊞	■	□	□	□	■	⊞
7	■	□	■	■	□	■	□	⊞	□	■	■	■	□
8	■	□	■	■	■	□	■	□	□	■	■	■	□
9	■	■	□	■	□	□	■	□	□	■	■	□	□
10	■	■	■	■	■	■	■	■	■	■	■	■	□
11	■	■	■	■	■	■	■	■	■	■	■	■	□
12	■	■	■	■	■	■	■	■	□	■	■	■	∅
13	□	□	■	■	■	■	■	□	□	■	□	■	□
14	⊞	⊞	□	⊞	⊞	□	□	■	■	■	⊞	□	■
15	■	■	■	■	■	■	■	□	□	■	■	■	□
16	□	□	□	□	□	□	□	■	□	□	□	□	□
17	□	□	□	□	□	□	□	□	□	□	□	□	□
18	□	□	□	□	■	□	□	⊞	□	□	□	□	□
19	■	□	■	□	■	□	■	⊞	⊞	■	■	□	■
20	■	□	■	□	■	□	■	■	■	■	■	□	■
21	□	□	□	□	□	□	⊞	□	□	□	□	□	□
22	■	□	□	□	■	□	■	■	□	■	■	□	■
23	□	□	□	□	■	□	□	□	□	⊞	□	□	□
24	⊞	□	□	□	■	□	■	■	□	■	■	□	⊞
25	□	□	■	□	■	□	⊞	□	■	■	□	□	□
26	■	□	■	□	■	□	■	■	■	■	■	□	■
27	⊞	□	□	□	⊞	□	□	□	□	□	⊞	□	■

■: yes, ⊞: possible, □: no, ∅: not applicable

One might argue that Graversen [2006] provides a more suitable classification of RPLs. Though I concede that Graversen's feature model is more suitable to compare the different runtime environments, I still insist that our classification scheme provides useful insights into the constructs, concepts, and relations introduced by RPLs to encode the different natures of roles.

After discussing the research field on RPLs, it is important to evaluate the practical applicability and extensibility of the various approaches by both researchers and practitioners. In accordance, the programming languages are henceforth compared by means of their host language, availability of their grammar, and viability of their implementation. For instance, eight of the contemporary RPLs extend the *Java* programming language by adding multiple syntactic language constructs, whereas both *JAWIRO* [Selçuk and Erdoğan, 2004] and *RICA-J* [Serrano and Ossowski, 2004] introduce the notion of roles to *Java* by providing an application library. Similarly, *Scala Roles* presents a very lightweight application library for *Scala* that introduces roles utilizing *Scala's* flexible syntax. Last but not least, *Rumer* [Balzer, 2011] and *Relations* [Harkes and Visser, 2014] are full-fledged programming languages that define a complete syntax, type system, and semantics. By contrast, only five of the language extensions actually provided a partial grammar, i.e. *OT/J* [Herrmann and Hundt, 2013], *Rava* [He et al., 2006], *powerJava* [Arnaudo et al., 2007], *NextEJ* [Kamina and Tamai, 2010], and *JavaStage* [Barbosa and Aguiar, 2012]. More important than the availability of partial grammars, is the availability of actual implementations. Unfortunately, only five of the thirteen contemporary RPLs made their compilers or libraries publicly available, i.e. *EpsilonJ*,[2] *OT/J*,[3] *powerJava*,[4] *Scala Roles*,[5] *Relations*,[6] even though the other authors were contacted and asked for their implementation. Indeed, its is possible to still run these compilers, when compiled and executed in the corresponding environment. Despite that, only *OT/J* is fully integrated into the *Eclipse* IDE. This not only includes an editor, a compiler, and a debugger; but also the support of a small community of role-based programmers. While this makes it very easy for practitioners to give role-based programming a try, the compiler is to complex to be easily extensible by researchers. Arguably, a lightweight approach like *Scala Roles* would be a better choice for them. In conclusion, there are at least two viable RPLs available to both researchers and practitioners, however, this did not lead to a wide spread use of either of them. This raises the question, why researchers develop new languages from scratch rather than improve upon an existing programming language?

6.3 RESULTS AND FINDINGS

After evaluating and comparing the contemporary role-based languages, this section elucidates the findings and results of the conducted SLR. In general, the survey provides evidence that both the research field on role-based modeling and role-based programming languages suffer from fragmentation and discontinuity [Kühn et al., 2014]. Especially, as most approaches reinvent the notion of roles without taking previous definitions into account. Although this might be the result of negligence, I argue that this is due to a lack of a common understanding of roles among researchers. In fact, researchers attribute different features to roles that fit their particular use case or application domain without being aware of the relations to features of related works. This becomes evident when answering the research questions.

[2] `http://tamai-lab.ws.hosei.ac.jp/pub/epsilon/epsilonj/index.html`
[3] `http://git.eclipse.org/c/objectteams/org.eclipse.objectteams.git`
[4] `http://www.di.unito.it/~guido/powerJava.zip`
[5] `https://github.com/tupshin/Scala-Roles`
[6] `https://github.com/metaborg/relations`

First, is there a common subset of features all contemporary approaches satisfy? Yes, but it only consists of Feature 1 stating that roles have properties and behaviors, as well as Feature 3 declaring that objects can play multiple roles simultaneously. Besides these two, the investigation of the rows of both tables indicate that neither the set of RMLs nor the set of RPLs share a significant amount of common features. This, in turn, is surprising due to the fact that Steimann [2000b] already established such a set by defining *Lodwick* as the lowest common denominator.

Second, how did Steimann's seminal work influenced the research field? In fact, his work had only a limited influence on contemporary RMLs and RPLs. In particular, only few role-based languages actually applied his classification scheme [Kim et al., 2003, Herrmann, 2005, Zhu and Zhou, 2006, Boella and Van Der Torre, 2007, Pradel and Odersky, 2009]. Nonetheless, none of the role-based languages used or extended *Lodwick's* definition of roles. In sum, less than half (11 of 26) of the approaches referenced [Steimann, 2000b] as related work. Evidently, his work did not harmonize and foster the research on role-based languages as he intended [Steimann, 2000b].

Finally, have advances in RMLs been adopted by later RPLs and vice versa? In the same way as *Lodwick* was overlooked, most RMLs only considered other modeling languages as related work. Conversely, most RPLs relate themselves to other programming languages, but, at least *OT/J* [Herrmann, 2005], *powerJava* [Boella and Van Der Torre, 2007], and *Scala Roles* [Pradel and Odersky, 2009] founded their notion of roles on the conceptual framework provided by Steimann [2000b]. In consequence, the SLR uncovered the following problems in the research fields on RMLs and RPLs:

- There is neither a *common understanding* nor *common feature set* shared among the different contemporary role-based modeling and programming languages.
- The research fields on RMLs and RPLs are characterized by an ongoing *discontinuity* and *fragmentation*. Specifically, most approaches reinvent the role concept without taking the definitions of preceding related approaches into account.
- Only four RMLs provide a sufficient *formal foundation* for roles able to incorporate *all natures of roles*, i.e. [Da Silva et al., 2003, Genovese, 2007, Liu and Hu, 2009a, Hennicker and Klarl, 2014]. Regardless, none of them is able to *support all features of roles*.
- Last but not least, most role-based modeling and programming languages are *not readily applicable*, due to their complexity, ambiguous terminology, and/or missing tool support. Even though *OT/J* represents a feature rich, practically usable programming language, there is no corresponding *readily applicable modeling language*.

To approach these problems, the second part of this thesis aims at harmonizing both research fields by providing the formal foundations of combined role-based modeling languages, as well as a family of RMLs supported by a flexible modeling editor.

PART II

FAMILY OF ROLE-BASED MODELING LANGUAGES

"All models are wrong, but some are useful."
— Box [1979]

7 FOUNDATIONS OF ROLE-BASED MODELING LANGUAGES

Following the argument of George E. P. Box, every model used to represent a *system under study* is deemed wrong, however, as history proofs, models can *simplify*, *abstract*, and *focus* the real world [Murer et al., 2008]. These useful models share four important properties [Henderson-Sellers, 2012]. First, they provide *clarity*, about the concepts, relations, and their properties for all users of the model. Second, all users must be *committed* to the model, the representation, and possible consequences. Third, useful models sufficiently represent the system under study and can be used for *communication*. Last but not least, the same model must be used to *control* the specification, design, implementation, and verification throughout the development process. As a result, everyone should focus on modeling languages capable of producing useful models.

In case of role-based modeling languages (RMLs), three blocking factors have to be addressed to enable both researchers and practitioners to produce useful role models. First, besides the intuitive semantics underlying the role concept, its notions must be ontologically founded and formally specified to create a coherent understanding of roles. Moreover, the formalization must combine the behavioral, relational, and context-dependent nature of roles into a comprehensive framework, especially, since most preceding formalizations focus either on the relational or context-dependent nature of roles, such as [Steimann, 2000b, Kim and Carrington, 2004, Balzer and Gross, 2011] and [Zhu and Zhou, 2006, Genovese, 2007], respectively. Furthermore, the formal framework must incorporate most of the modeling constraints of RMLs. In sum, this formal foundation provides *clarity* to both researchers and practitioners, alike. Second, there is no common graphical notation for role models that includes the various kinds of concepts. While most RMLs proposed a graphical notation based on UML, they usually resort to textual differentiation of concepts by means of stereotypes. This, however, is a *"cognitively inefficient way of dealing with excessive graphic complexity, [...] as text processing relies on less efficient cognitive processes"* [Moody, 2009, p.764]. Thus, to tame the graphic complexity of role models, a concise and comprehensive graphical notation must be provided for their specification. This, in turn, permits researchers and software designers to use role models to *communicate* their ideas, domain models, and software designs. Last but not least, there is a lack of tools that support the design, validation, and generation of role-based software systems. Although Halpin [2005] as well as Benevides and Guizzardi [2009] developed dedicated graphical editors, there exists no graphical editor for role models supporting all natures of roles and the various modeling constraints.

Along the same lines, only few approaches provide means to verify the well-formedness of a model or the consistency of its instances, e.g. [Kim and Carrington, 2004, Halpin, 2005, Benevides and Guizzardi, 2009, Hennicker et al., 2015]. In fact, to permit the scalability of an RML, automatic mechanisms to validate the well-formedness and consistency are required. Moreover, in order to design role-based systems in practice, additional tools must be provided to generate corresponding (partial) implementations from a given role model. This permits a fluent transition from a role-based design of an application to its implementation. As a result, additional tool support permits the use of the formal model to *control* the specification, design, verification, and implementation of role-based software systems.[1]

To overcome the first deficiency, this chapter provides both the ontological foundation and a comprehensive formal model for roles, denoted *Compartment Role Object Model (CROM)*, that combines all natures of roles as well as various modeling constraints [Kühn et al., 2015a,b]. To address the second issue accordingly, a corresponding graphical notation is introduced that standardizes the various visual representations of roles [Kühn et al., 2015a,b]. Finally, the third blocking factor is addressed in two ways. On the one hand, a reference implementation of the formal model is provided that is viable for both formal and automatic verification of well-formedness of models at design time and the consistency of their instances at runtime [Kühn et al., 2015a,b]. On the other hand, a corresponding *Full-fledged Role Modeling Editor (FRaMED)* is presented [Kühn et al., 2016]. *FRaMED* is a fully functional modeling editor that includes all natures and proposed modeling constraints. Moreover, it features distinct code generators generating either a formal representation based on the reference implementation, a model of the *context description logic* [Böhme and Lippmann, 2015], an *RSQL schema* [Jäkel et al., 2016], or a partial implementation in the *Scala Roles Language (SCROLL)* [Leuthäuser and Aßmann, 2015]. In conclusion, both *CROM* and *FRaMED* provide all means necessary to allow both researchers and practitioners to model, reason about, and implement role-based software systems. To put it bluntly, *FRaMED* is a graphical editor designed to establish CROM as a useful RML.

This chapter is structured accordingly. Section 7.1 provides the ontological foundation for RMLs, and Section 7.2 illustrates the graphical notation for role models. Afterwards, Section 7.3 highlights the formalization of CROM without inheritance, as published in [Kühn et al., 2015a]. Accordingly, Section 7.4 augments the formalization of CROM to include inheritance, as well. Using the former formalization, Section 7.5 outlines the main aspects of reference implementation. In addition to that, Section 7.6 highlights the tool support provided by the graphical modeling editor *FRaMED*.

7.1 ONTOLOGICAL FOUNDATION

Before providing any formal definition, it is crucial to classify the different kinds of concepts employed by the foundational RML introduced henceforth. Without this distinction, designers of role-based software systems are unable to decide whether a concept should be modeled as either *natural type*, *role type*, *compartment type*, or *relationship type*. Hence, this section provides the ontological foundation for role-based modeling by utilizing established metaproperties to classify the aforementioned kinds of concepts. The presented ontological foundation has been published in [Kühn et al., 2015a,b, Jäkel et al., 2016].

[1] Notably though, only *commitment* must be provided by the users of the RML to facilitate useful role models.

7.1.1 METAPROPERTIES

In order to provide a clear ontological distinction for the various concepts, three well-established ontological metaproperties are employed: *rigidity*, *identity* and *foundedness* (dependence).

First, **rigidity** [Guarino and Welty, 2000, 2009, Guizzardi, 2005] denotes that *"a property is* rigid *if it is essential to all its possible instances; an instance of a rigid property cannot stop being an instance of that property in a different world."* [Guarino and Welty, 2009, p.203]. Accordingly, if a type is a property of an instance, then instances of a *rigid type* belong to that type until they cease to exist [Guizzardi, 2005]. A *person*, for instance, can be considered a rigid type, because you can only stop being a person if you die. In contrast, instances of a *non-rigid type* can start and stop belonging to that type multiple times throughout their lifetime.[2] For example, the instance *Doreen* can start and stop to be of type *customer* depending on the situation, while still being the same person. Simply put, an instance of a non-rigid type can lose this type without losing its identity.

In accordance, the metaproperty **identity** [Guarino and Welty, 2000, 2009] distinguishes *"between properties that* carry an identity criterion *and properties that do not"* [Guarino and Welty, 2009, p.204]. In case of two instances of a certain type, the identity criterion determines whether both are considered equal. In consequence, a distinction is made between instances that have *no*, a *unique* (owned) or a *derived* (supplied) identity [Guarino and Welty, 2009]. A *person*, for instance, has a unique identity throughout its live time, whereas a *consultant* derives its identity from the person in that role. In addition, the notion of *composite identity* denotes the composition of the identities of two instances. An instance (tuple) of the *trans* relationship, for instance, is identified by the combined identities of the *source* and *target* accounts.

The third metaproperty, **foundedness** (dependence) [Guarino and Welty, 2000, 2009, Mizoguchi et al., 2012] defines that a *"property φ is externally dependent on a property ψ if, for all its instances x, necessarily some instance of ψ must exist, which is not a part nor a constituent of x"* [Guarino and Welty, 2000, p.103]. With respect to types, this entails that instances of a *founded type* (dependent type) can only exist when also an instance of another type exists at the same time. Again, the *customer* of our bank application is such a founded type, as instances of *bank customers* depend on the existence of the corresponding *bank*. Notably though, a type might be existentially-dependent on multiple different types.

In conclusion, these three metaproperties are sufficient to distinguish the concepts found in RMLs. However, in contrast to Guarino and Welty [2009], who favor the term *property* to classify sets of instances to avoid confusion, both Steimann [2000b] and Guizzardi [2005] argue that a distinction between *types* and *instances* respectively *universals* and *individuals* is more suitable for the specification of conceptual modeling languages, as it corresponds to the metalevel hierarchy. Following their argument, this thesis strictly separates the model level containing types from the instance level containing instances, whereas each instance belongs to at least one type.

7.1.2 CLASSIFYING MODELING CONCEPTS

By applying the metalevel hierarchy and the three ontological properties, it is possible to discern the following four kinds of concepts on the model level [Kühn et al., 2015a].

Natural Types are rigid, not founded, and their instances carry their own unique identity. Thus, instances of natural types, denoted *naturals*, have an immutable, independent type and identity. The entities *person*, *company*, and *account* are natural types in the banking application.

[2] For simplicity, we do not distinguish between semi-rigid and anti-rigid properties [Guarino and Welty, 2009].

Table 7.1: Ontological classification of concepts

Concept	Rigidity	Foundedness	Identity	Examples
Natural Types	yes	no	unique	*person, account*
Role Types	no	yes	derived	*customer, consultant*
Compartment Types	yes	yes	unique	*rransaction, bank*
Relationship Types	yes	yes	composite	*advises, own_ca*

Role Types, in contrast, are not rigid,[3] founded, and their instances only derive their identity from their players. In fact, instances of role types, simply called *roles*, depend on the identity of their player as well as a foundational relation to their context [Mizoguchi et al., 2012], i.e. the *participate* relation to instances of compartment types. Consequently, instances of a rigid type can dynamically adopt role types by playing one of its instances. In accordance to that, most entities in the banking domain become role types, e.g. *consultants*, *customer*, *source*, and *target*.

Compartment Types are rigid, founded, and their instances have a unique identity. Hence, instances of compartment types, henceforth denoted *compartments*, are founded on the existence of participating roles. For example, both *banks* and *transactions* are considered compartment types, as they are existentially-dependent on *consultant* respectively *source* and *target* role types.

Relationship Types are rigid, founded, and have a composed identity. They represent binary relationships between two distinct role types.[4] Specifically, the identity of *links* (relationship instances) is composed from the identities of the players of the participating roles. Consider, for instance, the *advises* link between the *consultant* Doreen and the *customer* Google whose identity is a composition of the identity of Doreen and Google.

Table 7.1 summarizes the ontological classification of *natural types*, *role types*, *compartment types*, and *relationship types* with respect to the introduced metaproperties.[5] Though I concede that more complex ontological classifications for roles, compartments and relationships exist, e.g. [Guizzardi, 2005, Masolo et al., 2004, Loebe, 2005, Boella and Van Der Torre, 2007], I still insist that the presented classification is sufficient and appropriate to distinguish the various concepts found in typical application domains. In fact, one only needs to ask the following three questions to classify a given domain concept as either a *natural type*, *role type*, *compartment type* or *relationship type*:

- Do instances of the concept belong to this type throughout their lifetime?
- Do instances of the concept depend on the existence of another instance to exist?
- Do instances of the concept carry a unique, derived or composite identity criterion?

Please note, however, that the answers to these question greatly depend on the modeled domain. Specifically, while in one domain *student* might be a role type in a *university*, it might be a natural type when modeling *exam regulations*. In conclusion, the presented ontological foundation provides a tool for both researchers and domain engineers to classify the concepts found in a given *system under study*.

[3] According to Guizzardi [2005] role types are classified as anti-rigid.
[4] Notably though, n-ary relationships can be represented with n binary relationships.
[5] Likewise, *data types*, such as money, temperature, or integer, can be classified as rigid, not founded, and have an identity derived from its state.

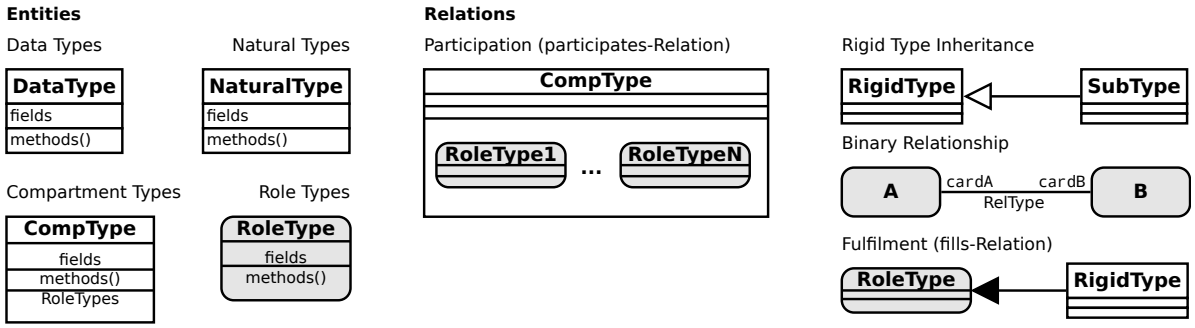

Figure 7.1: Graphical notation for role models.

7.2 GRAPHICAL NOTATION

After discussing the ontological foundation, this section facilitates a common graphical notation for RMLs by outlining its visualization and illustrating its use. In accordance with the distinction between types and instances, this section first illustrates the notation for entities and their relations on the model level, before exemplifying the corresponding notation on the instance level. Last but not least, visual representations for the various modeling constraints found in contemporary RMLs are introduced. In general, the graphical notation was introduced in [Kühn et al., 2015a] and is greatly inspired by UML as well as the notations presented in [Riehle and Gross, 1998, Balzer et al., 2007, Halpin, 2005, Herrmann, 2005]. However, in contrast to most contemporary graphical models, the proposed graphical notation follows the guidelines for visual languages proposed by Moody [2009].

7.2.1 MODEL LEVEL NOTATION

The design of a superior visual notation for RMLs is challenging, due to the increased number of modeling concepts and interrelations to incorporate. In contrast to UML class diagrams that only deals with *classes*, *associations*, and *inheritance* relations, a visual language for role models is more complex. It must provided different graphical symbols to visual distinguish *natural types*, *compartment types* and *role types* as well as their interrelations, e.g. *fills* and *participates* relations [Moody, 2009]. Like in UML, all types are represented as boxes with three segments containing the corresponding name, attributes and operations. However, to make *role types* easily distinguishable, they are drawn as boxes with rounded edges and with a gray background. In turn, all *rigid types*, i.e. natural, data and compartment types, are drawn as white boxes to visually associate their shape to the ability to fill role types. By extension, only *compartment types* have an additional segment that contains its participating role types, and relationship types. Similar to UML, *relationship types* are depicted as labeled lines between two role types, *inheritance* between rigid types is drawn with a white headed arrow from the subtype to the supertype, and the *fills* relation is denoted by a black headed arrow from a rigid type to the filled role type. In contrast to these relations depicted as a line, the *participates* relation (from compartment types to its role types) is directly represented as visually containment of role types within a compartment type. This, in turn, reflects the strong existentially dependency of role types to their compartment types. In sum, Figure 7.1 presents a common visual syntax for RMLs that not only provides a clear distinction between the modeling concepts and relations, but also corroborates their ontological foundation.

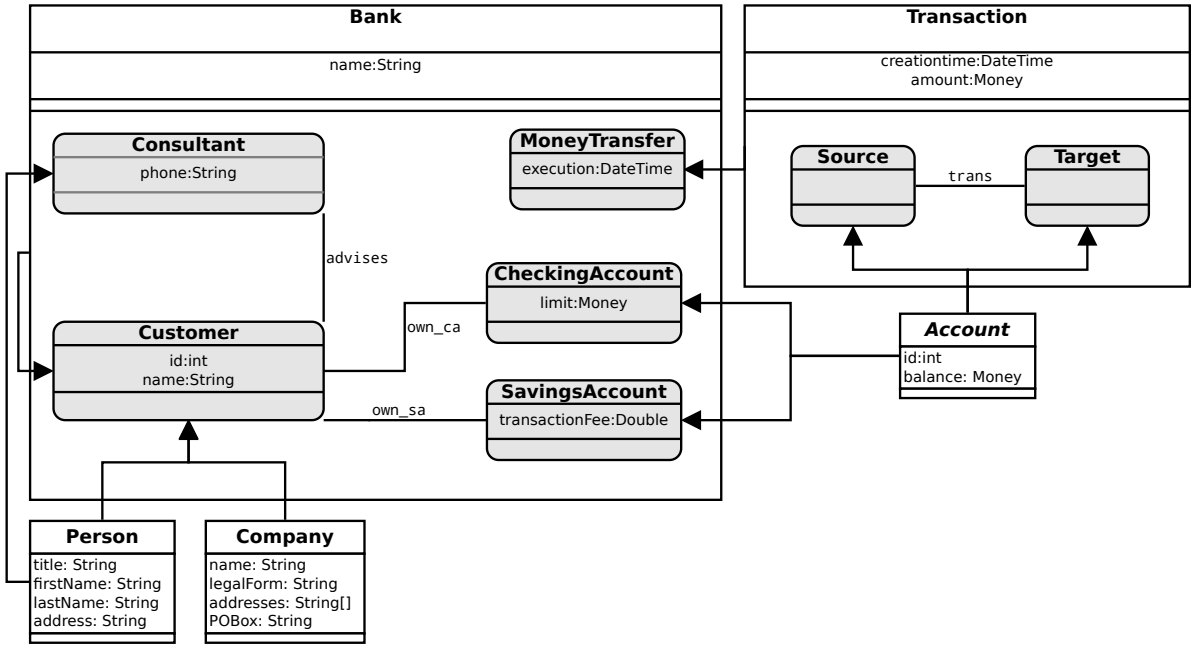

Figure 7.2: Role model of the banking application without constraints.

As an illustration, Figure 7.2 shows the corresponding model of the banking application showcasing the proposed visual language. Arguably, this representation of the banking domain is more concise and comprehensive than the corresponding representations of the contemporary combined modeling languages (cf. Section 4.3). Indeed, it omits the use of stereotypes as visual classifier, like in the *Helena Approach* [Hennicker and Klarl, 2014], or an overload of indistinguishable visual shapes and relations, as in *TAO* [Da Silva and De Lucena, 2007] or *INM* [Hu and Liu, 2009].[6] In consequence, this visual language for RMLs is able to tame the inherent complexity of the role concept by following the *Principles for Designing Effective Visual Notations* [Moody, 2009]. In essence, the graphical notations provide easily distinguishable graphical representations for each modeling concept and relation that reflects their intended semantics.

7.2.2 GRAPHICAL MODELING CONSTRAINTS

Even though the presented graphical notation captures the structure of role models, it lacks visual representations for the various modeling constraints, e.g. *role constraints*, *intra-*, and *inter-relationship constraints*, introduced in the contemporary literature. This is of particular importance, as the specification of constraints are a crucial aspect of domain modeling. Moreover, representing them visually makes them more accessible and recognizable, especially, when compared to their textual counterparts, e.g. *OCL* constraints added to UML class diagrams [Warmer and Kleppe, 1998]. However, to distinguish the modeling constraints from the other model elements, all constraints are drawn using dashed lines. In consequence, the graphical notation is augmented by adding visual representations for various modeling constraints, as shown in Figure 7.3. In general, these constraints range from typical *local role constraints* [Riehle and Gross, 1998] and *relationships constraints* [Balzer et al., 2007, Halpin, 2005] to *global role constraints*.

[6] See Figure 4.7, 4.8, and 4.9 for a direct comparison of their representations.

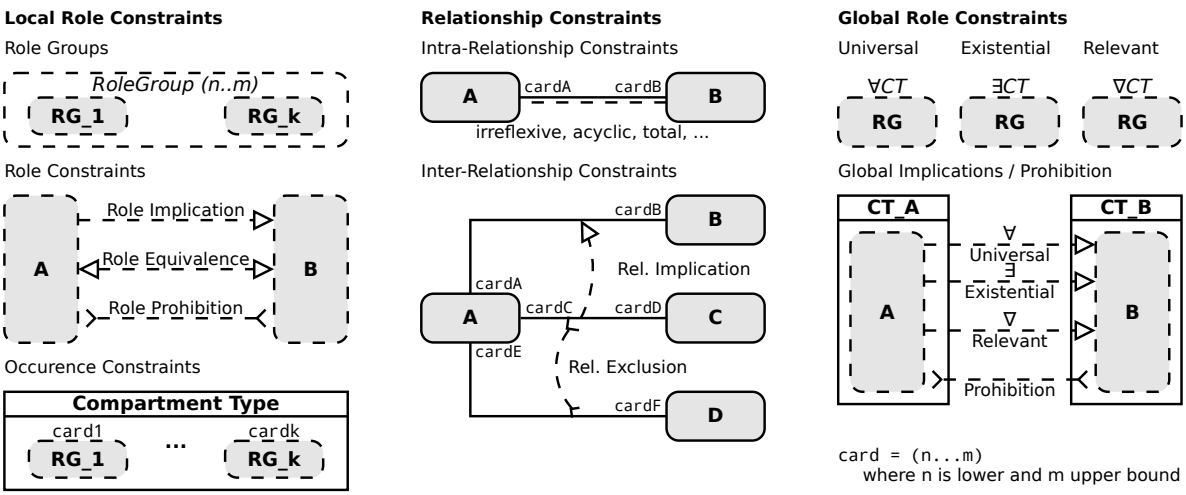

Figure 7.3: Graphical notation for various modeling constraints.

Firstly, *local role constraints* cover all constraints imposed on roles within a particular compartment instance. In detail, *role constraints* are represented by dashed boxes denoting *role groups* and dashed arrows between *role types* (and role groups) denoting role dependency relations. While *role groups* are a novel construct to impose a cardinality constraint on the players of a set of roles inspired by the *cardinality operator* [Van Hentenryck and Deville, 1990], role dependencies have been proposed by Riehle [1997] to requires or prohibits playing a role when another role is already played by an object. Specifically, the graphical notation follows Riehle's notation [Riehle and Gross, 1998] indicating *role implications* and *role equivalence* with a white arrow head and *role prohibition* with a horizontal line at both ends of the arrow. Finally, *occurrence constraints* [Kim et al., 2003, Zhu and Zhou, 2006, Hennicker and Klarl, 2014] are indicated by placing cardinalities above role types. *Relationship constraints*, in turn, includes cardinality constraints as well as intra- and inter-relationship constraints [Halpin, 2005, Balzer et al., 2007]. Cardinality constraints imposed on relationship types are noted at the corresponding relationship ends, whereas the intra-relationship constraints are drawn as a dashed line besides that relationship type and inter-relationship constraints as a curved dashed arrow between two relationship types. Yet, because intra-relationship constraints correspond to the properties of mathematical relations [Balzer et al., 2007], e.g. *reflexive*, *cyclic*, or *total*, they are written alongside the relationship. By contrast, the two inter-relationship constraints *relationship implication* and *relationship exclusion* correspond to the set-comparison constraints *subset* and *disjunction*, respectively [Halpin, 2005]. Last but not least, *global role constraints* are the logical consequence of the introduction of compartment types, as they permit denoting role constraints spanning multiple compartment types. In detail, they permit the quantification of a role group over all compartments of the given type by adding either the universal $\forall$, existential $\exists$, or relevance ∇ quantifier and the compartment type above a role group. In the same way, Riehle's role constraints can be extended to span multiple compartments leading to the notation of *universal*, *existential*, and *relevant global role implication* as well as the *global role prohibition*.[7]

All together, these modeling constraints allows for visually specifying the domain constraints imposed on the banking application. For instance, the constraint that no person playing the customer role is able to advises himself is expressed as an *irreflexive* constraint imposed on the *advises* relationship. Moreover, both the `BankAccounts` and the `Participants` role group with cardinality

[7] The semantics of global role constraints will be clarified in Section 7.3.3.

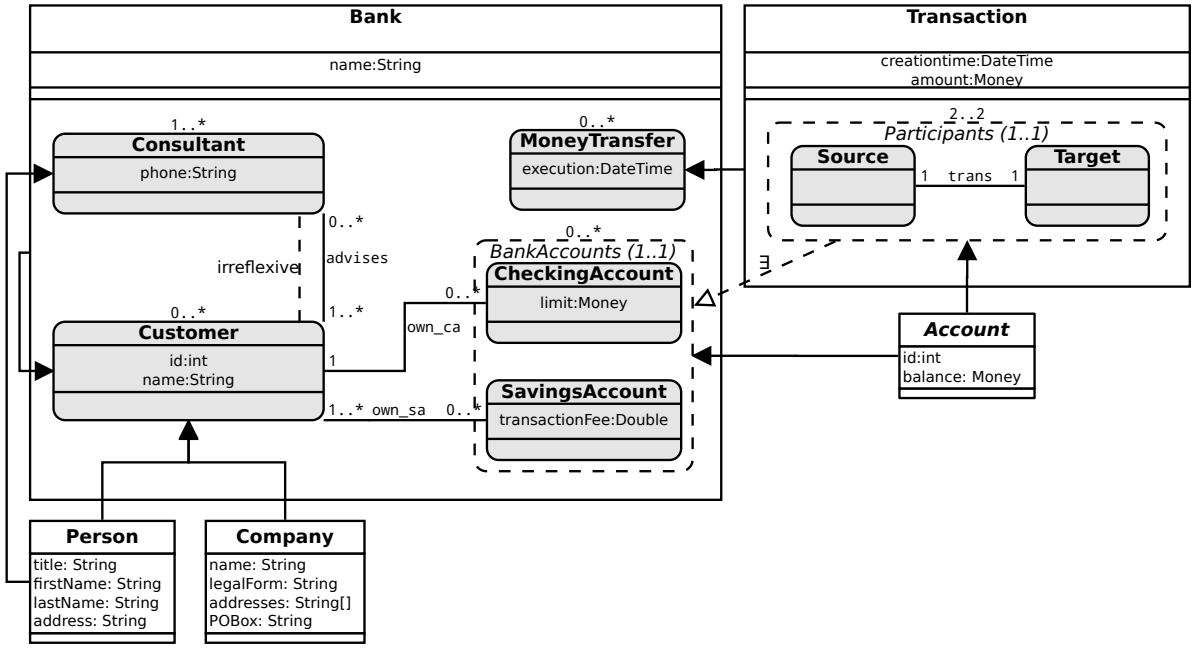

Figure 7.4: Role model of the banking application with additional constraints.

constraint `1..1` denote that no account can be both a *savings* and a *checking account* in the same *bank* compartment respectively both a *source* and *target* in the same *transaction*. Furthermore, the *existential role implication* from the `Participants` to the `BankAccounts` denotes that for each *account* participating in a *transaction* there exists a corresponding *bank* compartment where it is a *bank account*. Besides, the *occurrence constraints* above the `Consultant` role type of `1..*` and the `Participants` role group of `2..2` declares that there exists at least one *consultant* per *bank* compartment and exactly two *participants* per *transaction* compartment, respectively. By adding these constraints this banking model is able to capture all the financial regulations imposed on the banking application. In conclusion, the visual language, presented thus far, encompasses both the three natures of roles and the various modeling constraints creating a suitable graphical notation for role models.

7.2.3 INSTANCE LEVEL NOTATION

After presenting the graphical notation for role models, it is useful to additionally provide a notation for corresponding instances of these models. Hence, this subsection outlines the notation of role instance models in Figure 7.5. In accordance to the model level, objects are depicted as boxes with an underlined label indicating its name and type, whereas roles are drawn as rounded boxes with gray background. Adopting the notation used by Riehle and Gross [1998], *role instances* are drawn at the border of objects to indicate that they are played by the corresponding object. Notably though, a role can only be played by one player object at a time. Similar to relationship types, *relationship instances* are drawn as curved edge between roles of different types. In contrast, to denote that an object plays a role within a certain compartment instance, the objects playing roles in this compartment are drawn within the role instance model segment. Moreover, it is possible that a compartment contains another compartment playing a role in the outer compartment. Arguably, this leads to a clean design where containment denotes participation or fulfillment in case of roles. However, as objects can play multiple roles in multiple compartments, the same object might occur

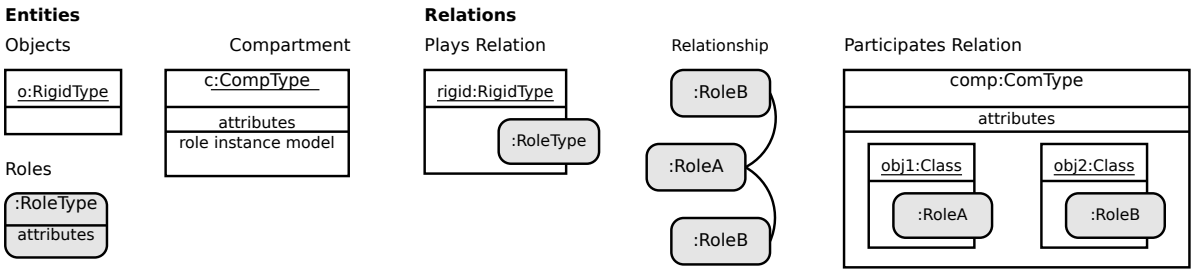

Figure 7.5: Graphical notation for role instance models.

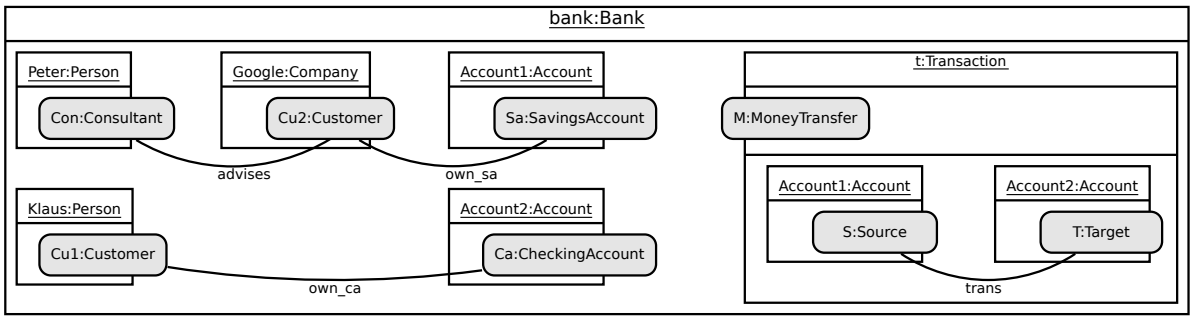

Figure 7.6: One possible role instance model of the modeled banking application.

several times within one role instance model.[8]

Consider, for example, the possible instance of the modeled banking application (cf. Figure 7.4) shown in Figure 7.5, where the accounts `Account1` and `Account2` appear two times, because both play roles in two different compartments. By extension, this role instance model comprises two *persons* `Peter` and `Klaus`, as well as a *company* `Google` that play roles in the `bank` compartment. Both `Klaus` and `Google` play a *customer* role owning a `CheckingAccount` and a `SavingsAccount`, respectively. Besides that, `Google` is advised by `Peter` playing the `Consultant` role. Additionally, the `bank` contains one *transaction* compartment `t` playing the `MoneyTransfer` role, where `Account1` is the *source* and `Account2` the *target* of the transaction. Basically, `t` represents the transaction from *Google's* savings account to *Klaus's* checking account. For brevity, the individual attributes of the compartments, objects, and roles have been omitted. Nonetheless, the presented instance of the modeled banking application clearly reflects its intended structure and, henceforth, serves as our running example of an instance of the bank model.

7.3 FORMALIZATION OF ROLES

After introducing both the ontological foundation and the graphical notation for RMLs, this section describes the initial formalization of the *Compartment Role Object Model (CROM)* that combines the behavioral, relational, and context-dependent nature of roles into a comprehensive and coherent formal framework [Kühn et al., 2015a]. Additionally, this formalization incorporates most of the modeling constraints found in the literature as well as *global role constraints* as additional class of role constraints. By extension, the framework is easy to comprehend and implement as it is only based on *set theory* and *first-order logic*.

[8] To avoid infinite recursion, a compartment playing a role in itself can be depicted as an object instead.

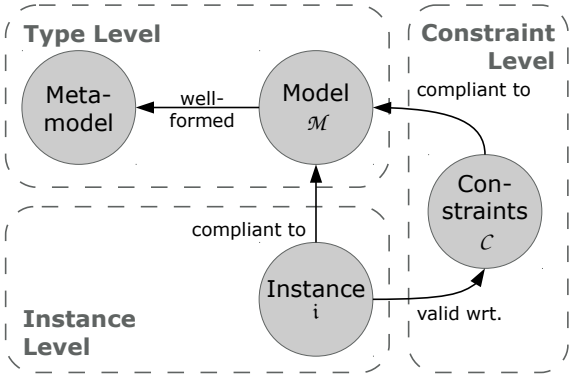

Figure 7.7: Overview of the presented formal model

For the sake of clarity, the formal framework is separated into three parts, as outlined in Figure 7.7. First, the *model level* encompasses the definition of *Compartment Role Object Models (CROMs)* and the notion of *well-formedness*. Second, the *instance level* formalizes *Compartment Role Object Instances (CROIs)* and defines when a CROI is *compliant to* a CROM. Finally, the *constraint level* specifies supported modeling constraints and combines them into a *Constraint Model*. In addition, it defines when a CROI is *valid* with respect to a Constraint Model. In conclusion, the presented formal framework is not only more comprehensive, but also easier to extend and specialize.

Even though, the formalization of roles, presented henceforth, has been published in [Kühn et al., 2015a,b], its formal definitions have been further refined in three major aspects. First and foremost, the definitions have been simplified by removing the need for ε roles, representing empty counter roles [Kühn et al., 2015a]. Secondly, the formalized modeling constraints additionally include definitions for *inter-relationship constraints*. Finally, to resolve the lack of role constraints spanning multiple compartments, the notion of quantified role groups is introduced as a novel modeling constraint. Still, for simplicity, both *fields* and *methods* have been omitted from the following definitions, however, the necessary additions are presented, later on, in Section 7.4.

7.3.1 MODEL LEVEL

The core aspects of a modeling language are captured on the model level. In case of RMLs, this includes the definition of natural types, compartment types, and role types as well as the declaration of the fulfillment relation and relationship types.

Definition 7.1 (Compartment Role Object Model)**.** *Let NT, RT, CT, and RST be mutual disjoint sets of natural types, role types, compartment types, and relationship types, respectively. Then a* Compartment Role Object Model (CROM) *is a tuple $\mathcal{M} = (NT, RT, CT, RST, fills, rel)$ where $fills \subseteq T \times CT \times RT$ is a relation and $rel : RST \times CT \to (RT \times RT)$ is a partial function. Here, $T := NT \cup CT$ denotes the set of all rigid types, i.e. all natural and compartment types.*

A CROM is denoted well-formed *if the following axioms hold:*

$$\forall rt \in RT\ \exists! ct \in CT\ \exists t \in T : (t, ct, rt) \in fills \tag{7.1}$$

$$\forall ct \in CT\ \exists (t, ct, rt) \in fills \tag{7.2}$$

$$\forall rst \in RST\ \exists ct \in CT : (rst, ct) \in \mathbf{domain}(rel) \tag{7.3}$$

$$\forall (rt_1, rt_2) \in \mathbf{codomain}(rel) : rt_1 \neq rt_2 \tag{7.4}$$

$$\forall (rst, ct) \in \mathbf{domain}(rel) : rel(rst, ct) = (rt_1, rt_2) \wedge (\_, ct, rt_1), (\_, ct, rt_2) \in fills \tag{7.5}$$

In this definition, *fills* denotes that rigid types can play roles of a certain role type in a given compartment type and *rel* captures the two role types at the respective ends of a relationship type defined in a compartment type.[9] Accordingly, the well-formedness rules restrict both the *fills* relation and the *rel* function. On the one hand, the first two axioms ensure that each role type participates in exactly one compartment type and is filled by at least one rigid type (7.1) as well as that each compartment type defines at least one participating role type (7.2). On the other hand, the other axioms make sure that each relationship type is defined at least in on compartment type (7.3). Moreover, the *rel* function is restricted to an irreflexive codomain (7.3), such that the two related role types participate in the same compartment type the relationship is defined in (7.5). Notably though, although a relationship type can occur in multiple compartment types with different definitions, each role type belongs to exactly one compartment type. Using this definition, a formal model of our running example can be created, as follows:

Example 7.1 (Compartment Role Object Model)**.** *Let $\mathcal{B} = (NT, RT, CT, RST, fills, rel)$ be the model of the bank (Figure 7.2), where the individual components are defined as follows:*

$$
\begin{aligned}
NT &:= \{Person, Company, Account\} \\
RT &:= \{Customer, Consultant, CA, SA, Source, Target, MoneyTransfer\} \\
CT &:= \{Bank, Transaction\} \\
RST &:= \{own\_ca, own\_sa, advises, trans\} \\
fills &:= \{(Person, Bank, Customer), (Company, Bank, Customer), (Bank, Bank, Customer), \\
&\quad (Person, Bank, Consultant), (Account, Bank, CA), (Account, Bank, SA), \\
&\quad (Transaction, Bank, MoneyTransfer), \\
&\quad (Account, Transaction, Source), (Account, Transaction, Target)\} \\
rel &:= \{(own\_ca, Bank) \rightarrow (Customer, CA), (own\_sa, Bank) \rightarrow (Customer, SA), \\
&\quad (advises, Bank) \rightarrow (Consultant, Customer), (trans, Transaction) \rightarrow (Source, Target)\}
\end{aligned}
$$

The bank model $\mathcal{B}$ is simply created from Figure 7.2 in three steps. First, all the natural types, compartment types, role types, and relationship types are collected into the corresponding set.[10] Second, the set of role types contained in each compartment type and the corresponding player types are collected in the *fills* relation. Finally, the *rel* function is defined for the role types at the ends of each relationship type in each compartment type, accordingly. Thus, a CROM model can be retrieved from its graphical representation.

In addition, the presented bank model $\mathcal{B}$ is well-formed, because each defined role type is filled by at least one natural type or compartment type and participates in exactly one compartment type (7.1), each compartment type contains at least one role type, and each relationship type is established (7.3) between two distinct role types (7.4) in the same compartment type (7.5). Besides well-formedness, the model has no meaning without taking its instances into account.

[9] For a given function $f : A \rightarrow B$, $\mathbf{domain}(f) = A$ returns the domain and $\mathbf{codomain}(f) = B$ the range of f.
[10] Henceforth, *SA* and *CA* are abbreviations for *savings account* and *checking account*, respectively.

7.3.2 INSTANCE LEVEL

The instance level, in turn, is inhabited by naturals, roles, compartments and links, as instances of their respective types. Thus, it encompasses the definition of the instances of a given CROM.

Definition 7.2 (Compartment Role Object Instance)**.** *Let $\mathcal{M} = (NT, RT, CT, RST, fills, rel)$ be a well-formed CROM and N, R, and C be mutual disjoint sets of naturals, roles and compartments, respectively. Then a* Compartment Role Object Instance (CROI) *of $\mathcal{M}$ is a tuple $\mathfrak{i} = (N, R, C, type, plays, links)$, where $type\colon (N \to NT) \cup (R \to RT) \cup (C \to CT)$ is a labeling function, $plays \subseteq (N \cup C) \times C \times R$ a relation, and $links\colon RST \times C \to 2^{R \times R}$ is a total function.*

To be compliant *to the model $\mathcal{M}$ the instance $\mathfrak{i}$ must satisfy the following conditions:*

$$\forall (o, c, r) \in plays\colon (type(o), type(c), type(r)) \in fills \tag{7.6}$$

$$\forall (o, c, r), (o, c, r') \in plays\colon r \neq r' \Rightarrow type(r) \neq type(r') \tag{7.7}$$

$$\forall r \in R\ \exists! o \in O\ \exists! c \in C\colon (o, c, r) \in plays \tag{7.8}$$

$$\forall rst \in RST\ \forall c \in C\ \forall (r_1, r_2) \in links(rst, c)\colon (rst, type(c)) \in \mathbf{domain}(rel) \wedge (\_, c, r_1), (\_, c, r_2) \in plays \wedge rel(rst, type(c)) = (type(r_1), type(r_2)) \tag{7.9}$$

In addition, $O := N \cup C$ denotes the set of all objects in $\mathfrak{i}$, $O^c := \{o \in O \mid \exists r \in R\colon (o, c, r) \in plays\}$ the set of objects played in compartment c, and $O^c_{rt} := \{o \in O \mid \exists r \in R\colon (o, c, r) \in plays \wedge type(r) = rt\}$ only those objects playing a certain type of role in c. Similarly, $R^c_{rt} := \{r \in R \mid (o, c, r) \in plays \wedge type(r) = rt\}$ collects all roles of type rt played in the compartment c.

In detail, the *type* function assigns a distinct type to each instance, whereas *plays* identifies the objects (either natural or compartment) playing a certain role in a specific compartment. In contrast, *links* captures the roles currently linked by a relationship type in a certain compartment. Additionally, a compliant CROI satisfies the given four axioms that guarantee the consistency of both the *plays* relation and the *links* function to the model $\mathcal{M}$. In case of the former, three axioms restrict the *plays* relation, such that the *plays* relation is consistent to the types defined in *fills* relation (7.6), an object is prohibited to play instances of the same role type multiple times in the same compartment (7.7), and each role has one distinct player in one distinct compartment (7.8). In contrast, the last axiom ensures that the *links* function only contains those roles, which participate in the same compartment c as the relationship and whose types are consistent to the relationship's definition in the *rel* function (7.9). In contrast to the published definition of CROM in [Kühn et al., 2015a, Definition 2], Definition 7.2 circumvents the use of *empty counter roles* ε in the definition of the *links* function by calculating the set of roles without a relationship before verifying the cardinality constraints. This, in turn, further simplifys the formal model. Still, one might come to the conclusion that $links(rst, c)$ can contain tuples for compartments c whose type does not define the relationship type rst. Even though this can be argued against, the above definitions allows for proving it wrong formally.

Theorem 7.1 (Completeness of Links)**.** *For a compliant CROI $\mathfrak{i}$ of a wellformed CROM $\mathcal{M}$ it holds for all $rst \in RST$ and $c \in C$ that from $(rst, type(c)) \notin \mathbf{domain}(rel)$ it follows that $links(rst, c) = \emptyset$.*

Proof. Assume that $(r_1, r_2) \in links(rst, c)$. Then, it follows from axiom (7.9) that $(rst, type(c)) \in \mathbf{domain}(rel)$, which is a contradiction. □

Besides all that, it is now possible to specify instances of the modeled banking application.

Example 7.2 (Compartment Role Object Instance)**.** *Let $\mathcal{B} = (NT, RT, CT, RST, fills, rel)$ be the well-formed CROM defined in Example 7.1; then $\mathfrak{b} = (N, R, C, type, plays, links)$ is an instance of that model (Figure 7.6), where the components are defined as follows:*

$$
\begin{aligned}
N &:= \{Peter, Klaus, Google, Account_1, Account_2\} \\
R &:= \{Cu_1, Cu_2, Con, Ca, Sa, S, T, M\} \\
C &:= \{bank, t\} \\
type &:= \{(Cu_1 \rightarrow Customer), (Cu_2 \rightarrow Customer), (Con \rightarrow Consultant), (Ca \rightarrow CA), (Sa \rightarrow SA), \\
&\quad (S \rightarrow Source), (T \rightarrow Target), (M \rightarrow MoneyTransfer), (bank \rightarrow Bank), (t \rightarrow Transaction), \ldots\} \\
plays &:= \{(Klaus, bank, Cu_1), (Google, bank, Cu_2), (Peter, bank, Con), (Account_1, bank, Sa), \\
&\quad (Account_2, bank, Ca), (t, bank, M), (Account_1, t, S), (Account_2, t, T)\} \\
links &:= \{(own\_ca, bank) \rightarrow \{(Cu_1, Ca)\}, (own\_sa, bank) \rightarrow \{((Cu_2, Sa)\}, \\
&\quad (advises, bank) \rightarrow \{(Con, Cu_2)\}, (trans, t) \rightarrow \{(S, T)\}\}
\end{aligned}
$$

The CROI $\mathfrak{b}$ is created, from Figure 7.6, by collecting all the naturals, compartments, and roles accordingly; mapping their respective types; linking the roles to their players; and assigning a tuple for each depicted relationship.[11] Additionally, it can be shown that the CROI $\mathfrak{b}$ is compliant to the CROM $\mathcal{B}$, i.e. it satisfies the four compliance conditions. First, the *plays* relation conforms to the types defined in the *fills* relation of the CROM $\mathcal{B}$ (7.6). Second, no object plays two roles of the same type in the two compartments *bank* and t (7.7). Third, for each role in $\mathfrak{b}$ there is exactly one object playing it in exactly one compartment (7.8). Finally, each of the four tuples in the *codomain* of the *links* function corresponds to the definition of the relationship type in the *rel* function. Moreover, the roles in *own_ca*, *own_sa*, *advises* participate in the *bank* compartment, whereas the roles in *trans* participate in *t*. In conclusion, the CROI $\mathfrak{b}$ is compliant to the CROM $\mathcal{B}$.

In addition to the former definition, three auxiliary functions are defined that are utilized later to validate the various relationship constraints.

Definition 7.3 (Auxiliary Functions)**.** *Let RST be the set of relationship types of a well-formed CROM $\mathcal{M}$, and $\mathfrak{i} = (N, R, C, type, plays, links)$ a CROI compliant to that model $\mathcal{M}$. Then the auxiliary functions* pred *and* succ*, as well as the* inverse *of the plays relation for roles $\overline{\cdot}: R \rightarrow O$ and its extension to the* links *function are defined for $r \in R$, $rst \in RST$, and $c \in C$:*

$$
\begin{aligned}
pred(rst, c, r) &:= \{r' \mid (r', r) \in links(rst, c)\} \\
succ(rst, c, r) &:= \{r' \mid (r, r') \in links(rst, c)\} \\
\overline{r} &:= o \textbf{ with } (o, \_, r) \in plays \\
\overline{links(rst, c)} &:= \{(\overline{r_1}, \overline{r_2}) \mid (r_1, r_2) \in links(rst, c)\}
\end{aligned}
$$

The first two functions collect the predecessors respectively successors of a given role in a relationship within a specific compartment instance. For the CROI $\mathfrak{b}$ (Example 8.2) $pred(own\_ca, bank, Ca)$ would return the set containing Cu_1. The existence of the next two functions, i.e., the inverse *plays* and inverse *links* function, is assured by (7.8) requiring a unique player and compartment for each role instance. In case of the bank instance $\mathfrak{b}$, $\overline{links(trans, t)}$ would return a singleton set with $(Account_1, Account_2)$. In particular, this function is used on the constraint level to evaluate whether a relationship is irreflexive, surjective, acyclic, and so forth [Balzer and Gross, 2011, Halpin, 2005]. However, up to this point the formal framework only asserts whether an instance complies to a given model without taking any modeling constraints into consideration.

[11] For brevity, the types of the naturals were omitted from the *type* function.

7.3.3 CONSTRAINT LEVEL

In accordance, the constraint level augments the formal model to represent the various constraints found in the literature review. In detail, this section presents *Role Groups* [Kühn et al., 2015a] as a novel construct to specify local role constraints and *Quantified Role Groups* as a corresponding global role constraint. Afterwards, *Constraint Models* are defined to encompass the various modeling constraints of CROM. Last but not least, the notion of validity is introduced for *CROIs* with respect to a given *Constraint Model* to specifying when a given instances fulfills the imposed constraints. Nonetheless, the notion of *Cardinality* has to be defined beforehand.

Definition 7.4 (Cardinality)**.** *Let* $\mathbb{N}$ *be the set of natural numbers including 0. Then* $Card \subset \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$ *is the set of cardinalities represented as* $i..j$ *with* $i \leq j$*.*

LOCAL ROLE GROUPS

The notion of *role groups* [Kühn et al., 2015a] have been introduced, after the formalization of *Riehle's role constraints* [Riehle and Gross, 1998] revealed their shortcomings. In fact, his role constraints cannot represent all conceivable combinations of constraints [Kühn et al., 2015b]. In consequence, role groups were introduced specifically to overcome their limitations. However, they ultimately replaced them altogether, due to the fact that all *role constraints* can be represented solely using *role groups*. This is not surprising, when considering that *role groups* are inspired by the *cardinality operator* [Van Hentenryck and Deville, 1990]. In accordance, the syntax and semantics of role groups is defined, as follows:

Definition 7.5 (Syntax of Role Groups)**.** *Let* RT *be the set of role types; then the* set of Role Groups RG *is defined inductively:*

- *If* $rt \in RT$*, then* $rt \in RG$*, and*
- *If* $B \subseteq RG$ *and* $n..m \in Card$*, then* $(B, n..m) \in RG$*.*

Definition 7.6 (Semantics of Role Groups)**.** *Let* RT *be the set of role types of a well-formed CROM* $\mathcal{M}, \mathfrak{i} = (N, R, C, type, plays, links)$ *a CROI compliant to* $\mathcal{M}$*,* $c \in C$ *a compartment, and* $o \in O$ *an object. Then the* semantics of Role Groups *is defined by the evaluation function* $(\cdot)^{\mathcal{I}^c_o} : RG \rightarrow \{0,1\}$*:*

$$a^{\mathcal{I}^c_o} := \begin{cases} 1 & \textbf{\textit{if}}\ a \in RT \wedge \exists (o, c, r) \in plays\colon type(r) = a \\ 1 & \textbf{\textit{if}}\ a \equiv (B, n..m) \wedge n \leq \sum_{b \in B} b^{\mathcal{I}^c_o} \leq m \\ 0 & \textbf{\textit{otherwise}} \end{cases}$$

For simplicity, an object $o \in O$ satisfies *a role group* $a \in RG$ *in a compartment* $c \in C$ *iff* $a^{\mathcal{I}^c_o} = 1$*.*

In general, role groups constrain the set of roles an object o is allowed to play simultaneously in a certain compartment c. In case a is a role type, $rt^{\mathcal{I}^c_o}$ checks whether o plays a role of type rt in c. If a is a role group (B, n, m), it checks whether the sum of the evaluations for all $b \in B$ is between n and m. For simplicity, an object o is said to satisfy a role group $a \in RG$ in a compartment $c \in C$ iff the evaluation function $\mathcal{I}$ returns one, i.e. $a^{\mathcal{I}^c_o} = 1$. Basically, role groups put a lower and upper bound on the types of roles an object can assume in one compartment instance.

Example 7.3 (Local Role Groups)**.** *The following two role groups can be extracted from Figure 7.4:*

$$bankaccounts := (\{CA, SA\}, 1..1)$$
$$participants := (\{Source, Target\}, 1..1)$$

The formal representation of role groups directly correspond to their graphical representation. Indeed, it can be shown that both Riehle's role constraints [Riehle and Gross, 1998] and any propositional formula are representable with role groups. As such, both role groups represent *role-prohibitions*, as they model an *exclusive-or*. Likewise, a *role-implication*, for instance, from consultant to customer could be specified as: $(\{(\{Consultant\}, 0, 0), Customer\}, 1, 2)$. This, in turn, is equivalent to the formula $\neg Consultant \vee Customer$ and thus to the intended semantics of the *role-implication*. Similarly, all *role constraints* can be expressed by role groups, as showcased henceforth.

Definition 7.7 (Shorthand Notations for Role Groups)**.** *Let $a, b \in RG$ be arbitrary role groups; then following shorthand is defined:*

$$\neg a := (\{a\}, 0..0) \qquad a \vee b := (\{a, b\}, 1..2)$$
$$a \wedge b := (\{a, b\}, 2..2) \qquad a \Rightarrow b := (\{(\{a\}, 0..0), b\}, 1..2)$$

Notably, it is easy to show the these definition follow the semantics of the corresponding propositional logic formulae. Besides all that, the following function enumerates the role types occurring in a given (nested) role group.

Definition 7.8 (Atoms of Role Groups)**.** *Let $\mathcal{M} = (NT, RT, CT, RST, fills, rel)$ be a well-formed CROM; then atoms: $RG \to 2^{RT}$ is a function, defined as:*

$$atoms(a) := \begin{cases} \{a\} & \textbf{\textit{if}}\ a \in RT \\ \bigcup_{b \in B} atoms(b) & \textbf{\textit{if}}\ a \equiv (B, n..m) \end{cases}$$

Simply put, the atoms function recursively collects all role types within a given role group. Consider, for instance, the *participants* role group for which atoms(*participants*) yields the set $\{Source, Target\}$.

GLOBAL ROLE GROUPS

Even though role groups can constrain the types of roles an object is able to play in a particular type of compartment, they cannot express constraints ranging over multiple compartment types. After all, it is impossible to declare that *each account participating in a **transaction** compartment must also play the role of either a **savings** or **checking account** in a **bank** compartment.* To integrate global constraints like these, the notion of role groups and their evaluation must be extended by permitting the quantification over multiple compartment instances within one role constraint. Consequently, the syntax and semantics of *quantified role groups* are defined as an extension to the local role groups.

Definition 7.9 (Syntax of Quantified Role Groups)**.** *Let RT be the set of role types, CT compartment types, and RG the set of role groups; then the* set of Quantified Role Groups *QRG is defined inductively:*

- *If $a \in RG$, $ct \in CT$ and $n..m \in Card$, then $\mathbb{Q}\langle ct, n..m\rangle.a \in QRG$, and*
- *If $B \subseteq QRG$ and $n..m \in Card$, then $\langle B, n..m\rangle \in QRG$.*

Definition 7.10 (Semantics of Quantified Role Groups)**.** *Let RT be the set of role types of a well-formed CROM $\mathcal{M}$, $\mathfrak{i} = (N, R, C, type, plays, links)$ a CROI compliant to $\mathcal{M}$ and $o \in O$ an object. Then the* semantics of Quantified Role Groups *is defined as the evaluation function $(\cdot)^{\mathcal{J}_o} : QRG \to \{0, 1\}$:*

$$a^{\mathcal{I}_o} := \begin{cases} 1 & \textbf{\textit{if}}\ a \equiv \mathbb{Q}\langle ct, m..n\rangle.b \wedge ct \in CT \wedge m \leq \sum_{d \in C_{ct}} b^{\mathcal{I}_o^d} \leq n \\ 1 & \textbf{\textit{if}}\ a \equiv \langle B, m..n\rangle \wedge m \leq \sum_{b \in B} b^{\mathcal{J}_o} \leq n \\ 0 & \textbf{\textit{otherwise}} \end{cases}$$

In general, *quantified role groups* constrain the number of compartment instances where an object must satisfy a specific *local role constraint* (i.e. *role group*). In detail, a quantification $\mathbb{Q}\langle ct, n..m\rangle.a^{\mathcal{I}_o}$ over a role group $a \in RG$ checks whether o *satisfies* the role group a in at least n and at most m compartments $c \in C$ of type ct. Furthermore, these quantifications can be combined just like *role groups*, however, using *quantified role groups* instead. Notably though, *local role groups* and *quantified role groups* cannot be combined arbitrarily. In fact, only *quantified role groups* can refer to *role groups* local to a compartment type by quantifying over the number of corresponding compartment instances. Accordingly, *local role groups* can only encompass to role types or other local role groups. Even though *quantified role groups* are a proper subset of *first-order logic*, they are easy to implement and expressive enough to represent the following typical global role constraints. In sum, an object o satisfies a quantified role group $\varphi \in QRG$ iff the evaluation function $\mathcal{J}$ returns one, i.e. $\varphi^{\mathcal{J}_o} = 1$.

Definition 7.11 (Shorthand Notations for Quantified Role Groups)**.** *Let $a \in RG$ be an arbitrary role group and $\varphi, \psi \in QRG$ arbitrary quantified role groups; then we define the following shorthand:*

$$\neg\varphi := \langle\{\varphi\}, 0..0\rangle \qquad \varphi \vee q := \langle\{\varphi, \psi\}, 1..2\rangle$$
$$\varphi \wedge q := \langle\{\varphi, \psi\}, 2..2\rangle \qquad \varphi \Rightarrow q := \langle\{\langle\{\varphi\}, 0..0\rangle, \psi\}, 1..2\rangle$$
$$\forall ct.a := \neg\langle\exists ct.\neg a\rangle \qquad \exists ct.a := \mathbb{Q}\langle ct, 1..\infty\rangle.a$$
$$\triangledown ct.a := \forall ct.((atoms(a), 1..\infty) \Rightarrow a) \qquad \exists! ct.a := \mathbb{Q}\langle ct, 1..1\rangle.a$$

While the first four definitions correspond to typical logical expressions, the latter four definitions correspond to quantifiers in first-order logic. Specifically, *existential* ($\exists ct.a$) and *universal* ($\forall ct.a$) quantification specify that each object $o \in O$ satisfies the role group a in at least one respectively all instances of the compartment type ct. Similarly, the *exactly once* quantifier ($\exists! ct.a$) denotes that each object $o \in O$ satisfies a in exactly one instance of the corresponding compartment type. In contrast to these *global role constraints* affecting all objects in a *CROI*, the *relevance* quantifier ($\triangledown ct.a$) ensures that only objects that play a relevant role in a compartment of type ct must satisfy the role group a. Simply put, an object is relevant for a compartment if it plays a role in that compartment. More precisely, an object is considered relevant in a compartment of type ct, if it plays a role of type rt contained in the atoms of the role group, i.e. $rt \in atoms(a)$. In consequence, the *existential role implication* depict in Figure 7.4 can now be formally specified, as follows:

Example 7.4 (Quantified Role Group)**.** *The following quantified role group is depicted in Figure 7.4:*

$$accountimpl := \langle\triangledown\, Transaction.participants \Rightarrow \exists Bank.bankaccounts\rangle$$

In general, this global role constraint denotes that for each object $o \in O$ relevant in a *Transaction* satisfying the *participants* role group there must also be a compartment of type *Bank* where o satisfies the *bankaccounts* role group. On one hand, the relevance quantifier ensures that the global role constraint is only applied to those objects that currently play either a *Source* or a *Target* role in a *Transaction* compartment, because $atoms(participants) = \{Source, Target\}$. On the other hand, the existential quantifier checks that for all relevant objects there is at least on *Bank* compartment that also satisfy the *bankaccounts* role group. In short, this global role constraint specifies that each object participating in a transaction either plays a savings or a checking account role in at least one bank. In case of the instance $\mathfrak{b}$ of the bank model $\mathcal{B}$, only $account_1$ and $account_2$ are relevant for the constraint, as $account_1$ plays the role T of type *Target* and $account_2$ the role S of type *Source*. Moreover, both accounts satisfy the *participants* role group in the *Transaction t* as well as the *bankaccounts* role group in the compartment *bank*. Consequently, both $account_1$ and $account_2$ satisfy this global role constraint.

CONSTRAINT MODEL

After formalizing both the local and quantified role groups, the following definition combines the various modeling constraints into a dedicated model. In general, a constraint model defines *local role constraint, intra-* and *inter-relationship constraints* declared for a specific compartment type, as well as *global role constraints* declared for a whole *CROM*.

Definition 7.12 (Constraint Model)**.** *Let $\mathcal{M} = (NT, RT, CT, RST, fills, rel)$ be a well-formed CROM and $IRC := \{\trianglelefteq, \otimes\}$ the set of inter-relationship constraints. Then $\mathcal{C} = (rolec, card, intra, inter, grolec)$ is a* Constraint Model over $\mathcal{M}$*, where $rolec\colon CT \to 2^{Card \times RG}$, and $card\colon RST \times CT \to (Card \times Card)$ are partial functions, as well as $intra \subseteq RST \times CT \times \mathbb{E}$ and $inter \subseteq RST \times CT \times IRC \times RST$ are relations with $\mathbb{E}$ as the set of functions $e\colon 2^O \times 2^O \times 2^{O \times O} \to \{0,1\}$ ranging over the set of objects O. Additionally, $grolec \subseteq QRG$ is a finite set of quantified role groups. A Constraint Model is* compliant *to the CROM $\mathcal{M}$ if the following axioms hold:*

$$\forall ct \in \textbf{\textit{domain}}(rolec)\, \forall (\_, a) \in rolec(ct)\colon atoms(a) \subseteq parts(ct) \tag{7.10}$$

$$\textbf{domain}(card) \subseteq \textbf{domain}(rel) \tag{7.11}$$

$$\forall (rst, ct, \_) \in intra\colon (rst, ct) \in \textbf{domain}(rel) \tag{7.12}$$

$$\forall (rst_1, ct, \_, rst_2) \in inter\colon (rst_1, ct), (rst_2, ct) \in \textbf{domain}(rel) \wedge\ rst_1 \neq rst_2 \tag{7.13}$$

Here, $parts(ct) := \{rt \mid (t, ct, rt) \in fills\}$ collects all role types defined within a compartment type.

Specifically, *rolec* collects all local role constraints imposed on specific compartment types. Each local role constraint, in turn, defines both a cardinality and a role group, such that the cardinality specifies the occurrence of objects satisfying the given role group. In accordance to that, *card* assigns a cardinality to a relationship type defined in a compartment type. Additionally, *intra* defines a set of *intra-relationship constraint* imposed on a relationship type in a compartment type, such that each constraint is given as an evaluation function. This function takes the domain A, range B, and the tuple set $\mathbb{R} \subseteq A \times B$ of a relationship and returns either zero or one. For instance, to define that the *advises* relationship type is *irreflexive*, a corresponding evaluation function returns one if $\forall x \in A \cup B\colon (x, x) \notin \mathbb{R}$ and zero otherwise. Of course, these evaluation functions directly correspond to mathematical properties of relations, e.g., reflexive, total, cyclic, and acyclic. In the same way, *inter* denotes a set of *relationship implications* and *exclusions* declared between two relationship types defined in the same compartment type. In particular, a *relationship implication* imposes a subset relation upon the two relationships, whereas a *relationship exclusion* enforces their disjointness. Notably, all these constraints are defined locally to a compartment type, i.e., no constraint crosses the boundary of a compartment type. Finally, *grolec* declares a set of quantified role groups that each object in a given CROI must satisfy. Besides all that, constraint models are denoted compliant to a CROM if they only constrain well-defined role and relationship types. Role groups defined in a compartment type, for instance, can only refer to role types defined in that compartment type (7.10). Similarly, cardinalities, intra-, and inter-relationship constraints must be declared for relationship types defined in the *rel* function (7.11–7.13). Moreover, (7.13) guarantees that inter-relationship constraints are specified between (two) distinct relationship types defined in the same compartment type. In conclusion, the constraint model captures not only most modeling constraints introduced in RMLs, but also *global role constraints* as novel kind of role constraint. Naturally, a *constraint model* can be easily defined for the constraints of the banking application, depicted in Figure 7.4.

Example 7.5 (Constraint Model)**.** *Let $\mathcal{B}$ be the bank model from Example 7.1 and $irreflexive$ an evaluation function for relationships. Then $\mathcal{C}_\mathcal{B} = (rolec, card, intra, inter, grolec)$ is the constraint model, where the components are defined as:*

$$\begin{aligned}
rolec &:= \{Bank \to \{(1..\infty, Consultant), (0..\infty, bankaccounts)\},\\
&\qquad (Transaction \to \{(2..2, participants)\}\}\\
card &:= \{(own_ca, Bank) \to (1..1, 0..\infty), (own_sa, Bank) \to (1..\infty, 0..\infty),\\
&\qquad (advises, Bank) \to (0..\infty, 1..\infty), (trans, Transaction) \to (1..1, 1..1)\}\\
intra &:= \{(advises, Bank, irreflexive)\}\\
inter &:= \{(own_ca, Bank, \otimes, own_sa)\}\\
grolec &:= \{accountimpl\}
\end{aligned}$$

A constraint model can be obtained by basically mapping the graphical constraints to their formal counterparts. Within each compartment type, role groups with cardinalities are added to the *rolec* mapping, relationship cardinality to the *card* function, intra-relationship constraints to the *intra* relation, and relationship implications/exclusions to the *inter* relation. Afterwards, the global role constraints are translated to quantified role groups, e.g., the *accountimpl*, and added to *rolec*. For the sake of argument, the example additionally introduces a relationship exclusion between *own_ca* and *own_sa* to the constraint model. Regardless, because each role group contains only role types of the same compartment type (7.10) and relationship constraints refer to relationship types in the correct compartment types (7.11–7.13), $\mathcal{C}_\mathcal{B}$ is compliant to the CROM $\mathcal{B}$. After separately defining the constraint model, the last step to verify the consistency of a given CROI is to validate whether it fulfill all the defined constraints. Accordingly, the notion of *validity* is formally defined, as:[12]

Definition 7.13 (Validity)**.** *Let $\mathcal{M} = (NT, RT, CT, RST, fills, rel)$ be a well-formed CROM, $\mathcal{C} = (rolec, card, intra, inter, grolec)$ a constraint model compliant to $\mathcal{M}$, and $\mathfrak{i} = (N, R, C, type, plays, links)$ a CROI compliant to $\mathcal{M}$. Then* $\mathfrak{i}$ is valid with respect to $\mathcal{C}$ *iff the following conditions hold:*

$$\forall ct \in CT\ \forall (i..j, a) \in rolec(ct)\ \forall c \in C_{ct}\colon\ i \le \left(\sum\nolimits_{o \in O^c} a^{\mathcal{I}_o^c}\right) \le j \tag{7.14}$$

$$\forall (o, c, r) \in plays\ \forall (crd, a) \in rolec(type(c))\colon\ type(r) \in atoms(a) \Rightarrow a^{\mathcal{I}_o^c} = 1 \tag{7.15}$$

$$\begin{aligned}
&\forall c \in C\ \forall (rst, type(c)) \in \mathbf{domain}(card)\colon\\
&\qquad rel(rst, type(c)) = (rt_1, rt_2) \land card(rst, type(c)) = (i..j, k..l) \land\\
&\qquad \left(\forall r_2 \in R^c_{rt_2}\colon i \le |pred(rst, c, r_2)| \le j\right) \land\\
&\qquad \left(\forall r_1 \in R^c_{rt_1}\colon k \le |succ(rst, c, r_1)| \le l\right)
\end{aligned} \tag{7.16}$$

$$\begin{aligned}
&\forall c \in C\ \forall (rst, type(c), f) \in intra\colon rel(rst, type(c)) = (rt_1, rt_2) \land\\
&\qquad f(O^c_{rt_1}, O^c_{rt_2}, \overline{links(rst, c)}) = 1
\end{aligned} \tag{7.17}$$

$$\forall c \in C\ \forall (rst_1, type(c), \otimes, rst_2) \in inter\colon \overline{links(rst_1, c)} \cap \overline{links(rst_2, c)} = \emptyset \tag{7.18}$$

$$\forall c \in C\ \forall (rst_1, type(c), \trianglelefteq, rst_2) \in inter\colon \overline{links(rst_1, c)} \subseteq \overline{links(rst_2, c)} \tag{7.19}$$

$$\forall o \in O\ \forall \varphi \in grolec\colon \varphi^{\mathcal{J}_o} = 1 \tag{7.20}$$

Here, $O^c := \{o \in O \mid \exists r \in R\colon (o, c, r) \in plays\}$ retrieves the set of objects played in compartment c, $O^c_{rt} := \{o \in O \mid \exists r \in R\colon (o, c, r) \in plays \land type(r) = rt\}$ only those objects playing a certain type of role in c, and $R^c_{rt} := \{r \in R \mid (o, c, r) \in plays \land type(r) = rt\}$ collects all roles of type rt played in the compartment c.

[12] Here, $|A|$ denotes the size of the set A, i.e., the number of elements in A.

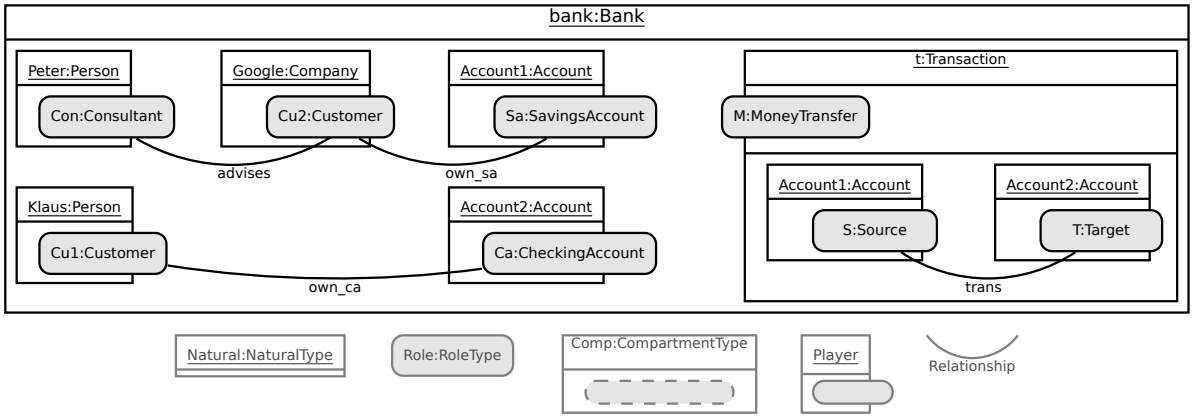

Figure 7.8: Graphical representation of the instance of the bank model.

In short, each axiom verifies a particular kind of constraint. The first two validate the *occurrence constraint* and *local role groups*. While (7.15) guarantees that only objects that play a relevant role in the constrained compartment must satisfy the corresponding role group, (7.14) checks whether the number of objects satisfying that role group is within the boundaries of the occurrence constraint.

In contrast to them, (7.16) verifies whether relationships respect the imposed cardinality constraints. However, due to the removal of the empty counter roles ε (cf. [Kühn et al., 2015a, Jäkel et al., 2015]), the *links* function only captures roles related by the relationship. Thus, to consistently check the cardinality of a relationship, its domain and range must be determined including all roles of the correct type in the given compartment c. In fact, this is done with R^c_{rt} that returns all roles in the compartment c of type rt. In case of the *advises* relation, shown in Figure 7.8, between *Consultant* and *Customer* within the *bank* compartment instance, the domain would be $R^{bank}_{Consultant} = \{Con\}$ and the range $R^{bank}_{Customer} = \{Cu_1, Cu_2\}$. Nonetheless, the cardinality of relationships is evaluated by counting the successors and predecessors of roles in its domain and range, respectively. In addition, (7.7) ensures the semantics of cardinality constraints, as it prevents an object to be related by multiple relationships (i.e., tuples in *links*) in the same compartment instance.

In the same way, (7.17) evaluates the various *intra-relationship constraints* by determining the domain and range of a relationship. Yet, to ascertain the that the player identity is used for the evaluation functions (e.g., *reflexive*, *cyclic*), the domain, range, and set of *links* of a relationship type *rst* must be lifted to objects, i.e., the domain is determined with O^c_{rt}, the range with O^c_{rt}, and *links* with $\overline{links}(rst, ct)$.

Accordingly, (7.18) and (7.19) ensure that each *inter-relationship constraint* is satisfied, i.e., *relationship implications* and *relationship prohibitions*. Conversely, both lift *links* to the corresponding objects with $\overline{links}(rst, ct)$ before evaluating the subset relation respectively the disjointness of both tuple sets.

Last but not least, (7.20) verifies that all objects (naturals or compartments) in a given CROI satisfy all *global role constraint* defined in *grolec*. As an illustration of the above definition, the bank instance $\mathfrak{b}$ (Example 8.2) is validated with respect to the constraints defined in $\mathcal{C}_{\mathcal{B}}$ (Example 8.3). For the sake comprehensibility, Figure 7.8 shows the role instance model corresponding to the CROI $\mathfrak{b}$.

Example 7.6 (Validity)**.** *To prove that the instance* $\mathfrak{b}$ *(Example 8.2) of the bank model* $\mathcal{B}$ *is* ***valid*** *with respect to the constraint model* $\mathcal{C}_\mathcal{B}$*, the axioms (7.14–7.20) must be fulfilled.*

Proof. To fulfill (7.14), at least one person must play a consultant role in the compartment $bank$, as well as exactly two distinct accounts must fulfill the $participants$ role group in the transaction t. Indeed, $Peter$ plays the consultant role Con in the $bank$. Moreover, $Account_1$ and $Account_2$ play the respective source role S and target role T in the transaction t, such that each account object individual fulfills the $participants$ role group. Accordingly, (7.14) holds for the CROI $\mathfrak{b}$.

For (7.15), each object playing a role in a compartment must fulfill all role groups that contain a relevant role type. In $\mathfrak{b}$, $Peter$ fulfills the $Consultant$ role group and both accounts, i.e., $Account_1$ and $Account_2$, individually satisfy both the $participants$ and the $bankaccounts$ role group. Thus, (7.15) also holds.

In case of (7.16), the number of successors for the domain and predecessors for the range of each link (relationship instance) is computed and checked against the limits imposed by the cardinality constraints. In case of the CROI $\mathfrak{b}$, the number of successors and predecessors ranges from zero, i.e., for Cu_1 in the own_sa and the $advises$ relationship and Cu_2 in the own_ca relationship, to one, i.e. for all other roles and relationships. As it turns out, the former cases all correspond to zero-to-many cardinality. As a result, (7.16) is satisfied, as well.

In contrast, (7.17) validates the other intra-relationship constraints, specifically, that the $advises$ relationship is irreflexive in $\mathfrak{b}$, i.e.:

$$irreflexive(O^{bank}_{Consultant}, O^{bank}_{Customer}, \overline{links(advises, bank)}) = 1$$

Trivially, $\overline{links(advises, bank)} = \{(Peter, Google)\}$ is irreflexive. Hence, the CROI $\mathfrak{b}$ fulfills (7.17).

Likewise, (7.18) and (7.19) check the relationship exclusion and relationship implication, respectively. However, because the constraint model $\mathcal{C}_\mathcal{B}$ only features a relationship exclusion between own_ca and own_sa, (7.18) only checks that $\overline{links(own\_ca, bank)} \cap \overline{links(own\_sa, bank)} = \emptyset$. Obviously, this holds in $\mathfrak{b}$, as each account is either a savings or a checking account, as ensured by the $bankaccounts$ role group.

Finally, the global role constraints are evaluated by (7.19). In the CROI $\mathfrak{b}$, for example, every object must fulfill the $accountimpl$ quantified role group. In fact, only the objects $Account_1$ and $Account_2$ play relevant roles in $Transaction$ compartment types. Moreover, because these accounts already satisfy the $bankaccounts$ role group in the $bank$ compartment, they also satisfy the global role group $\langle \exists Bank.bankaccounts \rangle$.

It follows, then that all axioms hold and, thus, $\mathfrak{b}$ is a **valid** instance of the model $\mathcal{B}$ with respect to the constraints specified in $\mathcal{C}_\mathcal{B}$. □

In essence, this example shows that a given *CROI* is consistent with a modeled domain if it is *compliant to* the corresponding *CROM* and *valid* with respect to the given *constraint model*. In spite of this informal validation, the idea of the formal framework is to support both formal and automated validation of well-formedness, compliance, and validity [Kühn et al., 2015a]. In fact, the presented formalization is easy to implemented and, thus, enables automatic validation of the formal models, as discussed in Section 7.5.

7.4 REINTRODUCING INHERITANCE

Up to this point, the presented formalization excluded the notion of *inheritance* for all types. Even though this made the initial formalization easier and more comprehensible, it presents a serious limitation to the Compartment Role Object Model (CROM). An anonymous reviewer puts this into perspective, when she argued that *"[it] seems by not supporting inheritance the baby is thrown out with the bath water"*. While inheritance can limit the extensibility of an object-oriented application, especially when considering the *fragile base class problem* [Mikhajlov and Sekerinski, 1998, Aßmann, 2003], it is still regarded the most suitable feature of object-oriented languages to foster code reuse. Thus, to save the *"baby"*, this section reintroduces the notion of inheritance to the formal Compartment Role Object Model (CROM). To be precise, the goal of this section is to reestablish subtyping of *natural types* and to formalize subtyping of both *compartment types* and *role types*.[13] Although *natural inheritance* corresponds to classical single inheritance of object-oriented languages, e.g., *Java*, *Scala*, both *role inheritance* and *compartment inheritance* closely resembles *family polymorphism*, i.e., *"a feature that allows us to express and manage multi-object relations"* [Ernst, 2001, p.303]. By extension, Dahchour et al. [2002] formally described the interaction of both natural and role inheritance with the *plays* relations. He concluded that all *subtypes* of a role type *rt* fulfill the same natural types as *rt*, whereas all *supertypes* of a natural type *nt* can play the same role types type as *nt* [Dahchour et al., 2002, Sec. 4.3]. By extension, Herrmann et al. [2004] outlined a methodology to reconcile role inheritance and compartment inheritance by introducing *family polymorphism* for compartment types and limiting role inheritance. In accordance with his approach, the presented extension to CROM facilitates compartment inheritance by means of family polymorphism. However, to avoid the introduction of dependent types [Ernst, 2001, Herrmann et al., 2004], the presented formalization employs *lightweight family polymorphism* [Igarashi et al., 2005] for compartment types and further restricts role inheritance to coincide with compartment inheritance. To put it succinctly, CROM_I incorporates both natural and compartment inheritance ensuring the subtyping relation as well as enabling the extension of both role types and compartment types. Furthermore, to illustrate the effects of inheritance, the notion of attributes and fields of the various entity types is added.

In accordance with the structure of the previous sections, the following discussion first introduces an extension to the banking application and then extends the various definitions on the model, instance, and constraint level. However, as the general aspects of the formal framework have already been described, the discussion focuses on the influence of subtyping on the definitions of well-formedness, compliance, and validity.

7.4.1 EXTENDING THE BANKING APPLICATION

Typically, the need for extension of a software system arises when the application domain is changed or extended. In case of our running example (cf. Chapter 2.1), the banking scenario additionally introduces two distinct kinds of financial institutions: retail banks and business banks. On the one hand, *retail banks* are banks that specialize on individual persons as customers. They have a several local affiliates and additionally provide *credit cards* to customers after assessing their liquidity. In short, customers of a retail bank can own at most one *credit card*, such that each one belongs to exactly one customer. Moreover, credit cards are bank account that store the interest charge and current debt, in addition to its id and balance.

[13] Subtyping ensures that if *s* is a subtype of *t* then *s* can safely occur wherever *t* would be expected.

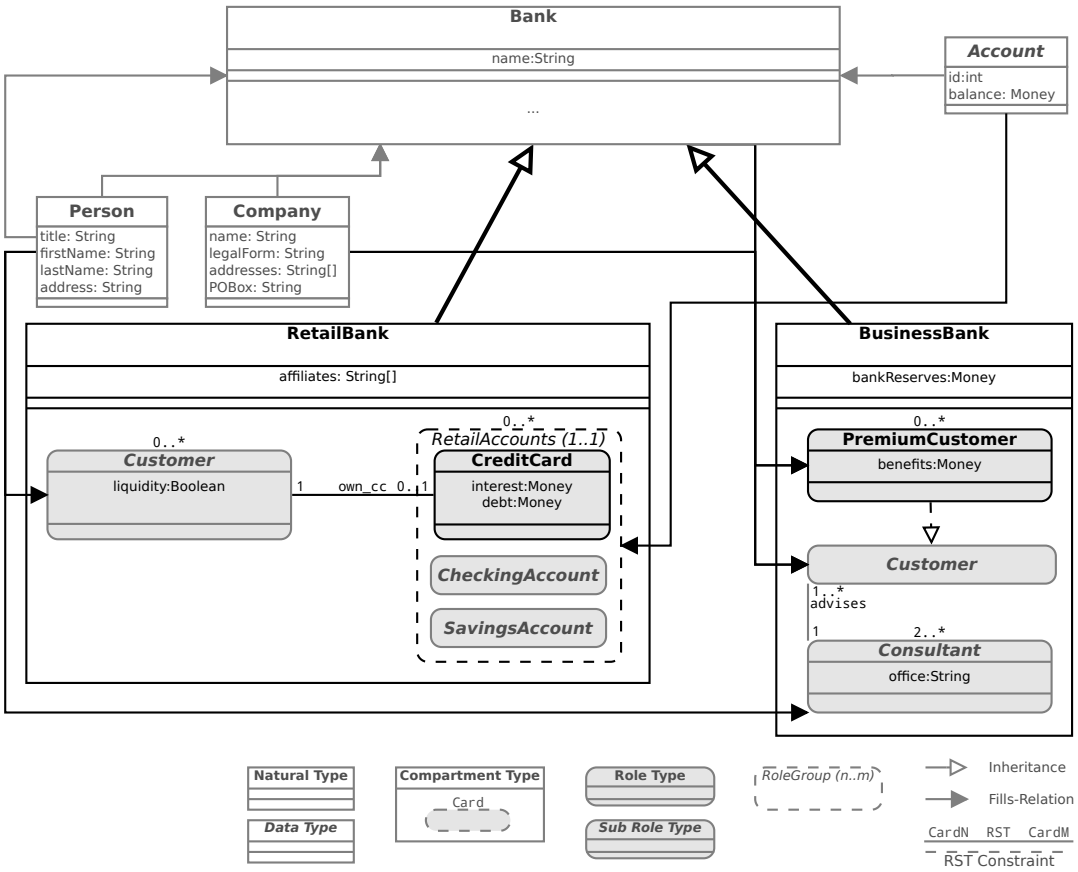

Figure 7.9: Extension of the banking application using compartment inheritance.

On the other hand, *Business banks* are financial institutions that solely focus on companies as customers. Specifically, they introduce *premium customers* as special customers, who will receive benefits worth a particular amount. Furthermore, business banks have a certain amount of money on reserve, must employ at least 2 *consultants* to be operational and require that each customer is advised by their own dedicated consultant.[14] To model these extensions of the banking application, accordingly, the `Bank` compartment (Figure 7.4) is specialized into two subcompartment types, as depicted in Figure 7.9. In particular, the `RetailBank` adds a `CreditCard` role type related to `Customer` by a `own_cc` relationship type. Moreover, the role group `RetailAccount` with `1..1` ensures that the credit card, savings account, and checking account are mutual exclusive. In contrast, the `BusinessBank` only introduces the `PremiumCustomer` as additional role type with a *role implication* to `Customer`, such that only companies who are already customers can become premium customers. Furthermore, the occurrence constraint above the `Counsultant` role type is increased to `2..*`, such that business banks employ at least two consultants. Similarly, the cardinality of the `advises` relationship type is updated to `1` on the consultant side indicating the each customer has exactly one consultant. Finally, the definition of the formal CROM can be augmented to appropriately model this extended banking application.

[14] Please note, that these constraints are for illustration purpose only and do not reflect actual financial regulations.

7.4.2 MODEL LEVEL EXTENSIONS

Unsurprisingly, the introduction of subtyping mostly influences the model level, as it entails the introduction of individual subtyping relations for both natural and compartment types as well as fields and methods for natural, compartment, and role types. In detail, the following definition introduces $\preceq$ as subtyping relation, such that $s \preceq t$ denotes that s is a subtype of t and, conversely, that t is a supertype of s.

Definition 7.14 (Compartment Role Object Model with Inheritance)**.** *Let NT, RT, CT, and RST be mutual disjoint sets of natural types, role types, compartment types, and relationship types, respectively, as well as F and M the set of field and method names. Then a* Compartment Role Object Model with Inheritance (CROM_I) *is a tuple $\mathcal{N} = (NT, RT, CT, RST, F, M, \mathit{fills}, \mathit{rel}, \mathit{fields}, \mathit{methods}, \prec_{NT}, \prec_{CT})$ denotes the CROM_I, where $\mathit{fills} \subseteq T \times CT \times RT$ is a relation, $\mathit{rel} : RST \times CT \to (RT \times RT)$ a partial functions, as well as $\mathit{fields} : T \cup (CT \times RT) \to MM(F \times T)$ and $\mathit{methods} : T \cup (CT \times RT) \to MM(M \times (\overline{T} \to T))$ total functions assigning a finite multiset to rigid types or role types (noted as $ct.rt$). Moreover, $\prec_{NT} \subset NT \times NT$ and $\prec_{CT} \subset CT \times CT$ represent irreflexive, asymmetric and functional relations over natural types and compartment types, respectively. For brevity, $\preceq_{NT}$, $\preceq_{CT}$ and $\preceq_T$ denotes the reflexive, transitive closure of $\prec_{NT}$, $\prec_{CT}$ and $\prec_T := \prec_{NT} \cup \prec_{CT}$, respectively.*

Additionally, a CROM_I is denoted well-formed *if (7.2–7.5) hold as well as the following axioms:*

$$\forall rt \in RT\ \exists ct \in CT\ \exists t \in T\colon\ (t, ct, rt) \in \mathit{fills} \tag{7.21}$$

$$\forall (t_1, t_2) \in \preceq_T\colon \mathit{fields}(t_2) \subseteq \mathit{fields}(t_1) \wedge \mathit{methods}(t_2) \subseteq \mathit{methods}(t_1) \tag{7.22}$$

$$\forall (ct_1, ct_2) \in \preceq_{CT}\ \forall rt \in \mathit{parts}(ct_2)\colon\ \mathit{fields}(ct_2.rt) \subseteq \mathit{fields}(ct_1.rt) \wedge \mathit{methods}(ct_2.rt) \subseteq \mathit{methods}(ct_1.rt) \tag{7.23}$$

$$\forall (ct_1, ct_2) \in \preceq_{CT}\ \mathit{parts}(ct_2) \subseteq \mathit{parts}(ct_1) \tag{7.24}$$

$$\forall (ct_1, ct_2) \in \preceq_{CT}\ \forall (rst, ct_2) \in \mathbf{domain}(\mathit{rel})\colon (rst, ct_1) \in \mathbf{domain}(\mathit{rel}) \wedge \mathit{rel}(rst, ct_2) = \mathit{rel}(rst, ct_1) \tag{7.25}$$

$$\forall (ct_1, ct_2) \in \preceq_{CT}\ \forall rt \in \mathit{parts}(ct_2)\ \forall (s, ct_1, rt) \in \mathit{fills}\ \exists (t, ct_2, rt) \in \mathit{fills}\colon s \preceq_T t \tag{7.26}$$

Here, $T := NT \cup CT$ denotes the set of all rigid types (i.e. natural and compartment types), $MM(S)$ the set of all finite multisets over S, and $\mathit{parts}(ct) := \{rt \mid (t, ct, rt) \in \mathit{fills}\}$ the set of role types defined within a compartment type.

In general, CROM_I introduces distinct subtyping relations for natural and compartment types, the notion of fields and methods, and additional well-formedness rules. In particular, $\prec_{NT}$ and $\prec_{CT}$ represent two disjoint inheritance hierarchies with *single inheritance*. Nonetheless, they induce the corresponding subtyping relations $\preceq_{NT}$ and $\preceq_{CT}$ for natural and compartment types, respectively. Besides that, the model includes the functions *fields* and *methods* to assign a set of fields ($f : t$) and method definitions ($m : \overline{T} \to T$) to each type. However, due to the fact that role types rt depend on a specific compartment type ct, their *fields* and *methods* functions are assigned to a *path type* from compartment type to role type, denoted as $ct.rt$. Nonetheless, it is assumed that $\text{fields}(t) = \emptyset$ and $\text{methods}(t) = \emptyset$, if there is no field respectively method defined for type $t \in T \cup (CT \times RT)$. In accordance with these additions, a well-formed CROM_I must fulfill six axioms in addition to the four axioms (7.2), (7.3), (7.4) and (7.5) of Definition 7.1. Notably though, it must also fulfill a weaker form of (7.1) that permits role types to participate in multiple compartments (7.21), such that a role type can be redefined in different compartment types. This, for instance, allows for defining distinct *customer* role types for a bank and a shop compartment type. In fact, these types are considered different, as they have different properties and behavior.

Besides that, the rest of the axioms validate that the type definitions adhere to the subtyping relation. For instance, (7.22) guarantees that all subtypes of a rigid type inherit all fields and methods from its supertype. Similarly, (7.23) ensures that all fields and methods of a role type rt defined in a compartment type ct_2 are passed down to all corresponding definitions in subcompartment types of ct_1. Basically, this axiom establishes that role inheritance coincides with compartment inheritance, such that the definition of a role type can be extended in a sub compartment type of its original definition. In the same way, (7.24) and (7.25) establishes that a sub compartment type inherits all role types and relationship types from its supercompartment type. Even though the above axioms handle subtyping of individual types, only (7.26) captures the interaction of natural inheritance, compartment inheritance, and *plays* relations. In particular, this axiom ensures that a sub compartment type can only restrict the types able to play an inherited role type by either limiting the players or requiring a sub type of a player type instead. Conversely, an inherited role type cannot be filled by more player types then its definition in the supercompartment type would permit. Consider, for instance, the retail bank as specialization of a bank compartment type. While the bank permits both persons, companies, and banks to be customers, the retail bank can restrict this set of players to persons only, however, cannot introduce a car as potential player of the customer role type. Notably though, there must be at least one compatible type filling an inherited role type, because (7.24) would be violated otherwise. In summary, compartment inheritance not only allows for adding new fields, methods, and role types but also enables the specialization of both the inherited role types and the types of their players.[15] Hence, it permits us to extend the modeled banking application by means of compartment inheritance.

Example 7.7 (Compartment Role Object Model with Inheritance)**.** *Let $\mathcal{B} = (NT, RT, CT, RST, fills, rel)$ be the CROM of the bank (Example 7.1). Then a CROM with Inheritance of the extended bank (Figure 7.9) is defined as $\mathcal{A} = (NT', RT', CT', RST', F', M', fills', rel', fields', methods', \prec_{NT}, \prec_{CT})$, where the individual components are defined as follows:*

$$
\begin{aligned}
NT' &:= NT \cup \{Money, Boolean, \ldots\} \\
RT' &:= RT \cup \{CC, PremiumCustomer\} \\
CT' &:= CT \cup \{RetailBank, BusinessBank\} \\
RST' &:= RST \cup \{own\_cc\} \\
F' &:= \{liquidity, interest, debt, benefits, \ldots\} \\
M' &:= \{increase, decrease, \ldots\} \\
fills' &:= fills \cup \{(Person, RetailBank, Customer), (Account, RetailBank, CC), \ldots \\
&\qquad (Company, BusinessBank, Customer), (Company, BusinessBank, PremiumCustomer), \\
&\qquad (Bank, BusinessBank, Customer), (Bank, BusinessBank, PremiumCustomer), \ldots\} \\
rel' &:= rel \cup \{(own\_cc, RetailBank) \rightarrow (Customer, CreditCard), \ldots\} \\
fields' &:= \{RetailBank.Customer \rightarrow \{(liquidity, Boolean), \ldots\}, \\
&\qquad RetailBank.CC \rightarrow \{(interest, Money), (debt, Money), \ldots\}, \\
&\qquad BusinessBank.PremiumCustomer \rightarrow \{(benefits, Money)\}, \ldots\} \\
methods' &:= \{RetailBank.CC \rightarrow \{decreas : Money \rightarrow Boolean, \ldots\}\} \\
\prec_{NT} &:= \emptyset \\
\prec_{CT} &:= \{(RetailBank, Bank), (BusinessBank, Bank)\}
\end{aligned}
$$

[15] Only single inheritance is introduced, as all cases of multiple classifications of objects can and should be modeled using roles and compartments instead.

Even though this model can be derived from Figure 7.9 as easily as Example 7.2, it only presents an incomplete CROM_I. Due to the fact that compartment inheritance relies on redefining fields, methods, and role types for each subtype, a complete model CROM_I would be significantly larger. For the sake of brevity and comprehensibility, however, only the definition of the additional role types introduced within the subcompartment types *RetailBank* and the *BusinessBank* is highlighted. For instance, the *RetailBank* includes a *credit card* role types, named as *CC*, such that it is filled by *Account* types and owned by exactly one customer of the retail bank. Likewise, the *BusinessBank* adds the role type *PremiumCustomer*, such that only *Company* natural type and *Bank* compartment types can play the *Customer* role type and, by extension, the *PremiumCustomer* role type. Moreover, *fields'* shows the definition of the fields of *Customer* and *CreditCard* within the *RetailBank*, e.g., the *liquidity* added to the *Customer* role type, as well as the *benefits* field of the *PremiumCustomer* within the *BusinessBank*. Furthermore, both *Money* and *Boolean* actually represent data types, for simplicity, the example defines them as natural types, instead. Finally, even though the above example cannot be shown to be a well-formed CROM_I, it is trivial to complete the extended bank model $\mathcal{A}$, such that the well-formedness rules hold. Consequently, $\mathcal{A}$ is assumed to be well-formed, henceforth.

7.4.3 INSTANCE LEVEL EXTENSIONS

So far, CROM_I ensures the conformance of natural and compartment inheritance only on the model level, however, subtyping is usually utilized on the instance level, to permit the save substitution of an object of a supertype with an object of a subtype. Consequently, the definition of *Compartment Role Object Instance with inheritance* is extended to take the subtyping relation of natural and compartment types into account. In short, the following definition guarantees the substitutability of objects of a type t with objects of a subtype of t for both attributes and *plays* relations.

Definition 7.15 (Compartment Role Object Instance with Inheritance)**.** *Let $\mathcal{N}$ be a well-formed CROM_I with $\mathcal{N} = (NT, RT, CT, RST, F, M, \mathit{fills}, \mathit{rel}, \mathit{fields}, \mathit{methods}, \prec_{NT}, \prec_{CT})$, as well as N, R, and C be mutual disjoint sets of Naturals, Roles and Compartments, respectively. Henceforth, $O := N \cup C$ denotes the set of all objects. Then a* Compartment Role Object Instance with Inheritance (CROI_I) *of $\mathcal{N}$ is a tuple $\mathfrak{n} = (N, R, C, \mathit{type}, \mathit{plays}, \mathit{links}, \mathit{attr})$, where $\mathit{type} : (N \to NT) \cup (R \to RT) \cup (C \to CT)$ is a labeling function, $\mathit{plays} \subseteq O \times C \times R$ a relation, $\mathit{links} : RST \times C \to 2^{R \times R}$ a partial function, and $\mathit{attr} : (N \cup R \cup C) \to 2^{F \times O}$ a total function.*

Moreover, $\mathfrak{n}$ is compliant *to $\mathcal{N}$ if it satisfies (7.7–7.9) and the following axioms:*

$$\forall (o, c, r) \in \mathit{plays}\ \exists (t, \mathit{type}(c), \mathit{type}(r)) \in \mathit{fills}\colon\ \mathit{type}(o) \preceq_T t \tag{7.27}$$

$$\forall o \in O\ \forall (f, v) \in \mathit{attr}(o)\ \exists (f, t) \in \mathit{fields}(\mathit{type}(o))\colon \mathit{type}(v) \preceq_T t \tag{7.28}$$

$$\forall (o, c, r) \in \mathit{plays}\ \forall (f, v) \in \mathit{attr}(r)\ \exists (f, t) \in \mathit{fields}(\mathit{type}(c).\mathit{type}(r))\colon \mathit{type}(v) \preceq_T t \tag{7.29}$$

In general, this extension adds the *attr* function, which maps each natural, compartment, and role instance to its state, i.e. a set of assignments of objects to field names. Beyond that, a compliant CROI_I satisfies the compliance rules (7.7), (7.8), (7.9) of Definition 7.2. In addition to these, the above definition extends axiom (7.6) to permit objects of a subtype $s \preceq_T t$ as players of a role, if *fills* declares t as the corresponding player type (7.27). Simply put, this axiom facilitates the substitutability of the role players in accordance with the subtyping relation. Similarly, (7.28) and (7.29) guarantee that the objects assigned to an attribute have the same or a subtype of the type defined in *fields*. In short, CROI_I introduces attributes and compliance rules, such that substitutability for both attributes and role players is permitted.

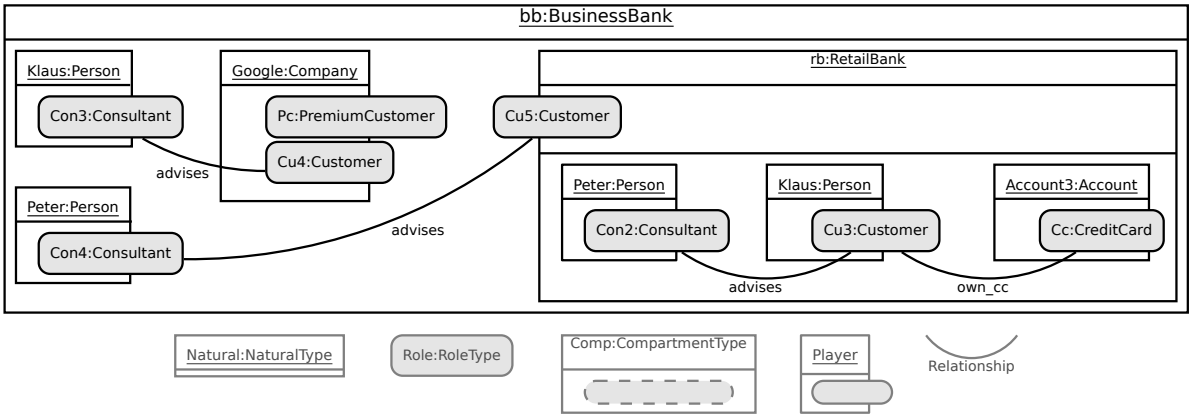

Figure 7.10: A possible role instance model of the extended banking application.

Applying this definition, in turn, allows for easily extending the instance $\mathfrak{b}$ of the bank model $\mathcal{B}$ to include the newly defined *retail* and *business banks*, in accordance to the role instance model depicted in Figure 7.10.

Example 7.8 (Compartment Role Object Instance with Inheritance)**.** *Let* $\mathfrak{b} = (N, R, C, type, plays, links)$ *be the CROI from Example 7.2 and* $\mathcal{A} = (NT', RT', CT', RST', F', M', fills', rel', fields', methods', \prec_{NT}, \prec_{CT})$ *the CROM*$_I$ *defined previously (Example 7.8); then* $\mathfrak{a} = (N', R', C', type', plays', links', attr')$ *is an instance of the extended bank model, where the components are derived as follows:*

$$
\begin{aligned}
N' &:= N \cup \{Account_3\} \\
R' &:= R \cup \{Cu_3, Cu_4, Cu_5, Con_2, Con_3, Con_4, Cc, Pc\} \\
C' &:= C \cup \{rb, bb\} \\
type' &:= type \cup \{(Cc \rightarrow CC), (Pc \rightarrow PremiumCustomer), \\
&\qquad (rb \rightarrow RetailBank), (bb \rightarrow BusinessBank), \ldots\} \\
plays' &:= plays \cup \{(Klaus, rb, Cu_3), (Peter, rb, Con_2), (Account_3, rb, Cc), \\
&\qquad (Klaus, bb, Con_3), (Peter, bb, Con_4), (Google, bb, Cu_4), (rb, bb, Cu_5)\} \\
links' &:= links \cup \{(own\_cc, rb) \rightarrow \{(Cu_3, Cc)\}, \\
&\qquad (advises, rb) \rightarrow \{(Con_2, Cu_3), (Con_4, Cu_5)\}, \\
&\qquad (advises, bb) \rightarrow \{(Con_3, Cu_4), (Con_4, Cu_5)\}\} \\
attr' &:= \{Cu_3 \rightarrow \{(liquidity, true), \ldots\}, \\
&\qquad Cc \rightarrow \{(interest, 100\,\$), (debt, 1000\,\$)\}, \\
&\qquad Pc \rightarrow \{(benefits, 1\,mio.\$)\}, \ldots\}
\end{aligned}
$$

In addition to the instance $\mathfrak{b}$, the extended CROI$_I$ $\mathfrak{a}$ additionally features a retail bank rb and a business bank bb. Specifically, the person *Klaus* is a customer Cu_3 in the retail bank rb who owns the credit card Cc played by $Account_3$. Moreover, *Peter* also plays the role of a *Consultant* in the retail bank *rb*, which, in turn, advises *Klaus*. Likewise, the business bank bb is specified to contain two customers Cu_4 and Cu_5 played by the company *Google* and the *retail bank* rb, respectively. However, only *Google* plays the role of a premium customer Pc. Besides that, both persons *Klaus* and *Peter* also play the consultant role Con_3 and Con_4 in the business bank bb, such that *Klaus advises Google* and *Peter advises* the *retail bank* rb. Last but not least, $\mathfrak{a}$ captures the state of the

various objects and roles, as well. Regardless, the example only specifies the attributes $attr'$ of some of the roles, i.e. the customer Cu_3, the credit card Cc, and the premium customer Pc, and the $type'$ function for roles and compartments whose type would be ambiguous otherwise. In conclusion, the instance $\mathfrak{a}$ of the extended bank model $\mathcal{A}$ represents simple example of a CROI_I. Nonetheless, the compliance of $\mathfrak{a}$ can only be argued, due to its partial definition. Even though, because the $plays'$ relation is completely defined, it is at least possible to validate Axiom (7.26). Yet, most roles are played by objects whose type equals to the player type in the *fills* relation. In fact, only the customer role Cu_5 is played by an object with the divergent player type *RetailBank*. However, because *Bank* fills the *Customer* role type in the *BusinessBank* compartment type and *RetailBank* is a subtype of the *Bank* compartment type, (7.26) still holds. Similar to the extended bank model $\mathcal{A}$, its extended instance $\dashv$ can be completed to be compliant to $\mathcal{A}$.

7.4.4 CONSTRAINT LEVEL EXTENSIONS

After introducing both natural and compartment inheritance on the model level and the instance level, the last step is to modify the semantics of the modeling constraints and their evaluation. However, while compartment inheritance affects the evaluation of the validity of a CROI_I, the semantics of most modeling constraints is unaffected. Indeed, both *local role constraints* and *relationship constraints* are evaluated for each individual compartment instance, even though their validation must adhere to subtyping, as well. To be precise, a constraint defined for a compartment type must also hold in all its subtypes. Thus, only the definition of validity must be extended without changing the semantics of the local constraints. In contrast, however, *global role constraints* must be refined, in order to take the subtypes of the quantified compartment types into account.

Definition 7.16 (Semantics of Quantified Role Groups with Inheritance)**.** *Let RT be the set of role types of a well-formed $\text{CROM}_I\,\mathcal{N}$, $\mathfrak{n} = (N, R, C, type, plays, links, attr)$ a CROI_I compliant to $\mathcal{N}$ and $o \in O$ an object. Then the* semantics of Quantified Role Groups *that handle compartment inheritance is defined by the evaluation function $(\cdot)^{\mathcal{H}_o} : QRG \to \{0,1\}$:*

$$a^{\mathcal{H}_o} := \begin{cases} 1 & \textbf{\textit{if}}\ a \equiv (B, m..n) \wedge m \leq \sum_{b \in B} b^{\mathcal{H}_o} \leq n \\ 1 & \textbf{\textit{if}}\ a \equiv \mathbb{Q}(ct, m..n).b \wedge : m \leq \sum_{d \in \widehat{C_{ct}}} b^{\mathcal{I}_o^d} \leq n \\ 0 & \textbf{\textit{otherwise}} \end{cases}$$

Here, C_{ct} is extended to $\widehat{C_{ct}} := \{c \in C \mid type(c) \preceq_{CT} ct\}$ to select all compartment instances of a type $t \preceq_{CT} ct$.

Admittedly, this definition only adapts the evaluation of quantifications $\mathbb{Q}\langle ct, n..m\rangle.a^{\mathcal{H}_o}$, in order to range over all compartment instances of the given type ct and all its subtypes. Defined in this way, the $accountimpl := \langle \triangledown Transaction.participants \Rightarrow \exists Bank.bankaccounts\rangle$ (Example 7.4), for instance, now entails that an account *participating* in a transaction must also be a *bank account* in at least one instance of *Bank* including instances of *RetailBank* and *BusinessBank*. Without this extension, the *quantified role group* would be violated for all *transactions* from or to accounts of *retail* and *business bank*. In fact, although $accountimpl$ ranges over all subtypes of *Bank*, it does not incorporate the added *CreditCard* role type. However, because the $bankaccounts$ role group does not include the *CreditCard* role type, the *quantified role group* is violated whenever a credit card (i.e., an account playing a *CreditCard* role) participates in a transaction. It follows, then that this *global role constraint* must be refined, as follows.

Example 7.9 (Extended Quantified Role Group)**.** *The following quantified role group is an extension of the* $accountimpl$ *of Example 7.4:*

$$existentialimpl := \langle \exists Transaction.(Source \vee Target) \Rightarrow \exists Bank.(CC \vee SA \vee CA) \rangle$$

After all, the definition of the constraint model is still appropriate to specify the modeling constraints. Hence, the example constraint model can be simply extended, as follows:

Example 7.10 (Extended Constraint Model)**.** *Let* $\mathcal{A}$ *be the extended bank model from Example 7.7 and* $\mathcal{C}_\mathcal{B} = (rolec, card, intra, inter, grolec)$ *the constraint model from Example 7.5; then* $\mathcal{C}_\mathcal{A} = (rolec', card', intra', inter', grolec')$ *is the extended constraint model derived from Figure 7.9, where the components are defined as:*

$$\begin{aligned}
rolec' &:= rolec \cup \{RetailBank \rightarrow \{(0..\infty, (\{CC, SA, CA\}, 1..1)\}, BusinessBank \rightarrow \{(2..\infty, Consultant)\}\} \\
card' &:= card \cup \{(own_cc, RetailBank) \rightarrow (1..1, 0..1)\} \\
intra' &:= intra' \\
inter' &:= inter' \\
grolec' &:= \{existentialimpl\}
\end{aligned}$$

Basically, the constraint model includes the additional modeling constraints depicted in Figure 7.9. Specifically, it adds the occurrence constraint of $2..\infty$ for consultants of the business bank, the `RetailAccount` role group, as well as the cardinality of the *own_cc* relationship type. In addition to that, the model includes the previously refined *global role constraint* to guarantee that transactions can only be performed between bank accounts, i.e., accounts that play a role in either a *Bank*, *RetailBank* or *BusinessBank* compartment. In short, the extended constraint model $\mathcal{C}_\mathcal{A}$ is a straightforward extension of the constraint model $\mathcal{C}_\mathcal{B}$. For the sake of completeness, the *compliance* of constraint models is broadened towards CROM_I, as well.

Definition 7.17 (Extended Compliance)**.** *Let* $\mathcal{N} = (NT, RT, CT, RST, F, M, fills, rel, fields, methods, \prec_{NT}, \prec_{CT})$ *be a well-formed* $CROM_I$ *and* $\mathcal{C} = (rolec, card, intra, inter, grolec)$ *be a* Constraint Model*. Then* $\mathcal{C}$ *is denoted* compliant *to* $\mathcal{N}$ *iff (7.10–7.13) hold for* $\mathcal{N}$*.*

In general, a compliant constraint model $\mathcal{C}$ of a CROM_I $\mathcal{N}$ satisfies the same four axioms as the original definition (cf. Definition 7.12). Especially, since these axioms only depend on the *fills* relation (via $parts(ct)$ and the *rel* function, and compartment inheritance entails the redefinition of role types and relationship types in all subcompartment types, they can be applied immediately. Furthermore, it easy to show that the constraint model $\mathcal{C}_\mathcal{A}$ is compliant to the CROM_I $\mathcal{A}$. In fact, the additional role groups and cardinality constraints only refer to role types and relationship types defined in the respective compartment type. Finally, the last definition extends the notion of validity to incorporate compartment inheritance and facilitate substitutability of compartment types, i.e., that each subtype of a compartment type can be safely used instead of its supertype.

Definition 7.18 (Extended Validity)**.** *Let $\mathcal{N} = (NT, RT, CT, RST, F, M, fills, rel, fields, methods, \prec_{NT}, \prec_{CT})$ be a well-formed CROM$_I$, $\mathcal{C} = (rolec, card, intra, inter, grolec)$ a constraint model compliant to $\mathcal{N}$, and $\mathfrak{n} = (N, R, C, type, plays, links, attr)$ a CROI$_I$ compliant to $\mathcal{N}$.*

Then the CROI$_I$ $\mathfrak{n}$ is *valid* with respect to $\mathcal{D}$ *iff the following conditions hold:*

$$\forall c \in C\ \forall (type(c), ct) \in \preceq_{CT}\ \forall (i..j, a) \in rolec(ct):\ i \leq \left(\sum_{o \in O^c} a^{\mathcal{I}_o^c}\right) \leq j \tag{7.30}$$

$$\forall (o, c, r) \in plays\ \forall (type(c), ct) \in \preceq_{CT}\ \forall (crd, a) \in rolec(ct):\ type(r) \in atoms(a) \Rightarrow a^{\mathcal{I}_o^c} = 1 \tag{7.31}$$

$$\begin{aligned} \forall c \in C\ \forall (type(c), ct) \in \preceq_{CT}\ \forall (rst, ct) \in \mathbf{domain}(card):\ & \\ & rel(rst, ct) = (rt_1, rt_2) \wedge card(rst, ct) = (i..j, k..l) \wedge \\ & \left(\forall r_2 \in R^c_{rt_2}: i \leq |pred(rst, c, r_2)| \leq j\right) \wedge \\ & \left(\forall r_1 \in R^c_{rt_1}: k \leq |succ(rst, c, r_1)| \leq l\right) \end{aligned} \tag{7.32}$$

$$\begin{aligned} \forall c \in C\ \forall (type(c), ct) \in \preceq_{CT}\ \forall (rst, ct, f) \in intra:\ & rel(rst, ct) = (rt_1, rt_2) \wedge \\ & f(O^c_{rt_1}, O^c_{rt_2}, \overline{links(rst, c)}) = 1 \end{aligned} \tag{7.33}$$

$$\begin{aligned} \forall c \in C\ \forall (type(c), ct) \in \preceq_{CT}\ \forall (rst_1, ct, \otimes, rst_2) \in inter:\ & \\ & \overline{links(rst_1, c)} \cap \overline{links(rst_2, c)} = \emptyset \end{aligned} \tag{7.34}$$

$$\forall c \in C\ \forall (type(c), ct) \in \preceq_{CT}\ \forall (rst_1, ct, \trianglelefteq, rst_2) \in inter: \overline{links(rst_1, c)} \subseteq \overline{links(rst_2, c)} \tag{7.35}$$

$$\forall o \in O\ \forall \varphi \in grolec:\ \varphi^{\mathcal{H}_o} = 1 \tag{7.36}$$

Here, $O^c_{rt} := \{o \in O \mid \exists r \in R: (o, c, r) \in plays \wedge type(r) = rt\}$ retrieves all objects playing a certain type of role in c, and $R^c_{rt} := \{r \in R \mid (o, c, r) \in plays \wedge type(r) = rt\}$ collects all roles of type rt played in the compartment c.

Indeed, the *extended validity* of CROI$_\text{I}$ refined all axioms, to account for the introduction of compartment inheritance. That is, all local role constraints and local relationship constraints imposed on a compartment type are also imposed on all its subtypes. Conversely, the axioms (7.30–7.35) not only validate the constraints defined for the exact type of compartment instance, but also validates the constraints defined by its supertypes. Accordingly, (7.36) is adapted to utilize the previously extended evaluation function $(\cdot)^{\mathcal{H}}$. In sum, these definition guarantees that all constraints defined for a compartment type are passed on to its subtypes. Moreover, it ensures that a sub compartment type can only further restrict the previously defined constraints, however, never weaken a constraint of a supertype. In essence, this reflects the notion of subtyping, i.e., that any instance of a subtype of a given compartment type ct can safely substitute an instance of ct. In conclusion, this definition defines the validity of CROI$_\text{I}$ in the presence of both natural and compartment inheritance. Consequently, Definition 7.18 is appropriate to verify the validity of the extended CROI$_\text{I}$ of our extended banking application, as follows:

Example 7.11 (Extended Validity)**.** *To prove that the instance $\mathfrak{a}$ (Example 7.8) of the bank model $\mathcal{A}$ is **valid** with respect to the constraint model $\mathcal{C}_\mathcal{A}$, the axioms (7.30–7.36) must be fulfilled.*

Proof. Due to the fact that both the instance $\mathfrak{a}$ and the constraint model $\mathcal{C}_\mathcal{A}$ extend the valid instance $\mathfrak{b}$ and the constraint model $\mathcal{C}_\mathcal{B}$, respectively, it suffices to validate the additional compartment instances rb and bb, as well as the additional and global role constraint.

In case of the compartment rb of type *RetailBank*, each local role constraint and relationship constraint defined for the *RetailBank* and the *Bank* compartment type must be verified. It is easy to see, that the participants of rb satisfies all constraint specified for *Bank* compartment type. Notably, while rb contains the $Account_3$ that would not fulfill the $bankaccounts$ role group, (7.31) still holds, because $CreditCard \notin atoms(bankaccounts)$. Similarly, the role group ({CC, SA, CA}, 1..1) and the cardinality of the *own_cc* relationship are satisfied for the compartment rb. Thus, the retail bank rb fulfills all local constraints, i.e., satisfies (3.30–3.36).

In the same way, each local constraint imposed on the *BusinessBank* or the *Bank* must be evaluated for the compartment bb. Indeed, bb fulfills the role and relationship constraints specified for the *Bank* compartment type, i.e. there is at least on consultant (7.30), each consultant advises a customer (7.32), and advises is irreflexive (7.33). Nonetheless, it also satisfies the much stronger occurrence constraint requiring two consultants within *RetailBank* compartments. Accordingly, business bank bb satisfies the imposed local constraints, as well.

Last but not least, the modified global role constraint $existentialimpl$ must be evaluated for all object in $\mathfrak{a}$ (7.36). Similar to $accountimpl$ quantified role group, it states that every object participating in a transaction compartment (playing either a *Source* or a *Target*) must also participate in a compartment of type *Bank*, *RetailBank* or *BusinessBank* (as either a *CC*, *CA*, or *SA*). Still, only the objects $Account_1$ and $Account_2$ play relevant roles in the *Transaction* t. Conversely, these accounts must also participate in a (subtype of) *Bank* as either a credit card, savings account, or checking account, i.e., playing a role of type *CC*, *CA*, or *SA*. Unsurprisingly, both accounts satisfy this role constraint within the compartment instance $bank$. From all this follows, that all axioms hold and, thus, $\mathfrak{a}$ is a **valid** instance of the model $\mathcal{A}$ with respect to the constraints specified in $\mathcal{C}_\mathcal{A}$. □

In conclusion, the presented formal framework CROM_I not only captures the behavioral, relational, and context-dependent nature of roles as well as various modeling constraints imposed on these models, but also supports the use of both natural and compartment inheritance. Furthermore, the formal framework is designed to support both formal and automated validation of the well-formedness, compliance, and validity. However, in order to permit automated validation of *CROMs*, *CROIs*, and *Constraint Models*; the development of a reference implementation is indispensable.

7.5 REFERENCE IMPLEMENTATION

Due to the fact, that the presented formal framework solely relies on set semantics and first-order logic, it is readily applicable for implementation and, by extension, for automated verification. To prove this, two reference implementations have been developed, whereas *formalCROM*[16] is based on *Python*[17] and *ScalaFormalCROM*[18] based on *Scala*[19]. In general, these implementations can be used to create *CROMs*, CROIs, *and* Constraint Models, *and automatically check their well-formedness, compliance, and validity [Kühn et al., 2015a]. In particular, the goal was to provide implementations that directly corresponds to the formal definitions. Henceforth, this section outlines the* Python* based implementation of the formal Compartment Role Object Model (CROM) without inheritance, as presented in Chapter 7.3 by first discussing the representation of logical formulae in *Python*, then presenting the architecture of the reference implementation, and finally illustrating its application for automatic verification.

[16] https://github.com/Eden-06/formalCROM
[17] https://www.python.org
[18] https://github.com/max-leuthaeuser/ScalaFormalCROM
[19] http://www.scala-lang.org

7.5.1 TRANSLATION OF LOGICAL FORMULAE

Though I concede that a *logic programming language*, e.g. *Prolog* [Clocksin and Mellish, 2003], seams to be a better choice, it is rather difficult to implement the additional set semantics with *Prolog*, for instance, to ensure that the $\prec_{NT}$ relation is asymmetric or that *links* is a left total function. *Python*, by contrast, supports most of the set operations directly or via the *itertools* module. More importantly, however, the combination of generator expressions and the build-in functions `all` and `any` permit the representation of most quantified first-order logic formulae. In fact, `all` and `any` directly correspond to *universal* and *existential* quantification in first-order logic. Similarly, all relations are represented as a python `set` containing tuples and functions as hash maps (python `dict`). Consider, for instance, the following axiom, extracted from Definition 7.1:

$$\forall rst \in RST\ \exists ct \in CT\colon\ (rst, ct) \in \textbf{domain}(\text{rel}) \tag{7.3}$$

This axiom can be implemented, as follows:

```
1 all(any( (rst,ct) in M.rel.keys() for ct in M.ct) for rst in M.rst)
```

To put it succinctly, the *universal quantification* $\forall rst \in RST$ is written as `all( ... for rst in M.rst )` and the *existential quantification* $\exists ct \in CT$ as `any( ... for ct in M.ct )`, where `crom.rst` and `M.ct` refer to the set of relationship types and set of compartment types, respectively. The final test `(rst,ct)\in M.rel.keys()` is then embedded into these *generator* expressions and checks that the given tuple `(rst,ct)` is a key of the hash map `rel`, i.e. that (rst, ct) is in the domain of the *rel* function. Similarly, each of the 20 axioms presented in the initial formalization (Section 7.3) have been implemented. In conclusion, *Python* provides sufficient language constructs to implement the specified axioms, such that each axiom closely resemble its formal definition. This, in turn, makes this reference implementation more comprehensible, then a corresponding *Prolog* implementation could.

7.5.2 STRUCTURE OF THE REFERENCE IMPLEMENTATION

While the underlying formalization should directly correspond to its formal specification, the architecture of the reference implementation should reflect the partitioning of the formal framework (cf. Figure 7.7) into a model, instance, and constraint level. Additionally, to permit later refinement of the various definitions, the definition of *CROM*, *CROI*, and the *constraint model* have been implemented as classes. Listing 7.1 shows an excerpt of the reference implementation. In detail, these provide a constructor declaring the corresponding, sets, relations, and functions as immutable attributes, as well as implement the relevant axioms as methods. Moreover, the class `CROM` implements the well-formedness criterion with the method `wellformed` that returns true if the axioms (7.1–7.5) return true, as well. Similarly, the class `CROI` and `ConstraintModel` each encode the compliance property as a method taking a `CROM` object as additional parameter. Finally, the notion of *validity* is defined as a method within the `ConstraintModel` class, which takes both a `CROI` object and a `CROM` object to check whether the instance is valid with respect to the given constraints. Notably though, the `valid` method first verifies that both the constraint model and the instance model are compliant to the given CROM. In sum, each class implements a corresponding definition. Thus, each definition can be specialized or extended by simply using *inheritance* and *method overriding*.[20] The reference implementation not only reflects the structure of the formal framework, but also permits researchers to easily apply and extend the presented formalization.

[20] Indeed, it is a mere technicality to implement the *CROM with Inheritance* as extension of the reference implementation.

Listing 7.1: Extract of the reference implementation.

```
class CROM:
  def __init__(self,nt,rt,ct,rst,fills,rel):
    # initialization
  def wellformed(self):
    return self.axiom1() and self.axiom2() and self.axiom3()
      and  self.axiom4() and self.axiom5()
    # axioms ...

class CROI:
    def __init__(self,n,r,c,type1,plays,links):
    # initialization
    def compliant(self,crom):
        return crom.wellformed() and self.axiom6(crom) and
            self.axiom7(crom)
           and self.axiom8(crom) and self.axiom9(crom)
  # axioms...

class ConstraintModel:
  def __init__(self,rolec,card,intra,inter,grolec):
    # initialization
  def compliant(self,crom):
    return crom.wellformed()
      and self.axiom10(crom) and self.axiom11(crom)
      and self.axiom12(crom) and self.axiom13(crom)
  def validity(self,crom,croi):
    return self.compliant(crom)   and croi.compliant(crom)
      and self.axiom14(crom,croi) and self.axiom15(crom,croi)
      and self.axiom16(crom,croi) and self.axiom17(crom,croi)
      and self.axiom18(crom,croi) and self.axiom19(crom,croi)
      and self.axiom20(crom,croi)
  # axioms...
```

7.5.3 SPECIFYING AND VERIFYING ROLE MODELS

Having a complete implementation of a formal framework grants the ability to automatically compute the properties of a specification. In case of the *formal CROM framework*, the reference implementation allows for specifying *CROMs*, *constraint models*, and *CROIs*, as well as the programmatic verification of their properties, e.g. *well-formedness*, *compliance*, and *validity*. To illustrate the use of the reference implementation, Listing 7.2 showcases the specification and automatic evaluation of the banking application, as modeled in Figure 7.4. In detail, it creates a `CROM` object `m` on the model level by providing the set of natural types, compartment types, role types, as well as set of tuples and a dictionary representing the *fills* relation and the *rel* function, respectively. Afterwards, the `CROI i` is constructed on the instance level by passing the set of naturals, compartments, and roles, as well as the *type* function, the *plays* relation, and the *links* function. This CROI, in turn, corresponds to the role instance model of the banking application depicted in Figure 7.6. Finally, the corresponding `ConstraintModel cm` is instantiated with the dictionaries `rolec` and `card` denoting the local role constraints and the cardinality constraints, as well as the sets `intra`, `inter`, `grolec` of *intra-relationship constraints*, *inter-relationship constraints*, and *global role constraints*. Once these three objects have been created, their methods can be used to automatically validate the specification. Line 27, for instance, verifies whether the specified model `m` simply by calling the `wellformed` method. Ultimately, to check if the specified instance `i` of the modeled banking application `m` is valid with respect to the constraint model `cm`, one simply needs to invoke the `valid` method of the constraint model with the model `m` and instance `i` as parameter. Similarly, each of the implemented axioms can be individually evaluated by calling the corresponding method.

Listing 7.2: Specification of banking application using the reference implementation.

```
#Model Level
NT   =["Person","Company","Account"]
RT   =["Source","Target","Consultant","Customer","CA","SA", ...]
CT   =["Transaction","Bank"]
RST  =["trans","own_ca","advises","own_sa"]
fills=[("Company","Bank","Customer"),("Person","Bank","Customer"),...]
rel  ={("trans","Transaction"): ("Source","Target"), ...}
m=CROM(NT,RT,CT,RST,fills,rel)
#Instance Level
n    =["Peter","Klaus","Google","Account_1","Account_2"]
r    =["Con","Cu_1","Cu_2","Ca","Sa","S","T","M"]
c    =["bank","transaction"]
type1={"Peter": "Person", "Klaus":"Person", "Google":"Company", ...}
plays=[("Klaus","bank","Cu_1"), ("Google","bank","Cu_2"), ...]
links={("own_ca","bank"): [ ("Cu_1","Ca") ], ...}
i=CROI(n,r,c,type1,plays,links)
#Constraint Level
rolec={"Transaction":[((2,2),RoleGroup(["Source","Target"],1,1))],
       "Bank":[((1,inf),"Consultant"),
               ((0,inf),RoleGroup(["CA","SA"],1,1))]}
card  ={("trans","Transaction"): ((1,1),(1,1)), ... }
intra =[("advises","Bank",irreflexive)]
inter =[("own_sa","Bank",exclusion,"own_ca")]
grolec=[...]
cm=ConstraintModel(rolec,card,intra,inter,grolec)

if (m.wellformed):   print "The bank model M is a wellformed CROM"
if (i.compliant(m)): print "The bank instance i is compliant to M"
if (cm.compliant(m)):print "The constraint model C is compliant to M"
if (cm.valid(m.i)):  print "The bank instance i is valid wrt. to C"
```

Notably though, these specifications directly correspond to the formal specifications presented in Example 7.1, Example 7.2, and Example 7.5. This as well as the previously described translation of the axioms, gives a strong indication of the soundness and completeness of the reference implementation. In conclusion, this reference implementation can be utilized by researchers to develop and test other implementations of the presented framework, as well as to investigate specializations and extensions of the Compartment Role Object Model.

7.6 FULL-FLEDGED ROLE MODELING EDITOR

Up to this point, the discussion mainly focused on formal foundation and representation of role models. While their definition copes with major blocking factors for researchers to investigate and improve RML, formal specifications are generally not regarded useful by practitioners. For them the major blocking factor is the lack of a graphical modeling editor for the creation, manipulation, and provisioning of role models. Moreover, to enable them to design role-based software systems, additional means for validation and code generation are mandatory. To address these issues, this section highlights the first *Full-fledged Role Modeling Editor (FRaMED)*, a graphical modeling editor for the design of *Compartment Role Object Models*. Introduced in [Kühn et al., 2016], the editor embraces all natures of roles as well as most of the presented modeling constraints. Besides providing a full-fledged modeling editor, FRaMED also features distinction code generators generating formal representations of CROM, data definitions for a role-based database or source code of role-based programming languages. Consequently, FRaMED is able to provide all means necessary to allow practitioners to model, reason about, and implement role-based software systems.

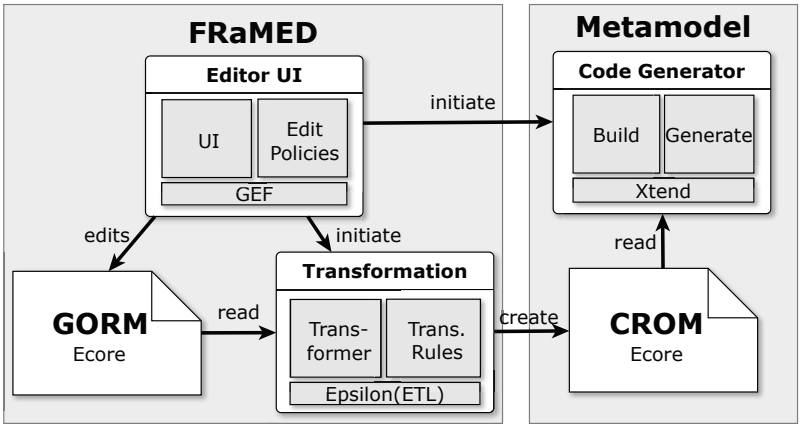

Figure 7.11: Architecture of FRaMED, extracted from tep@kuehn2016framed.

7.6.1 SOFTWARE ARCHITECTURE

Generally, *FRaMED* is built on the *Eclipse* platform[21] and is available on *GitHub*.[22] The editor was implemented using a model-driven approach and employs the *Eclipse Modeling Framework (EMF)*[23] and most of the following discussion has been published in [Kühn et al., 2016]. Figure 7.11 provides an overview of its software architecture.

Prior to its development, both the formalization of CROMs and constraint models have been encoded into a single corresponding *Ecore* metamodel. The resulting *CROM metamodel* is maintained in a separate plugin[24] and represents the central artifact for the editor and adjunct tools. However, this metamodel only represents the structure of CROMs and is void of any layout information, e.g. positions, rectangles, and bend points. To further decouple FRaMED from this metamodel, the editor represents a separate plugin, which only emits instances of the CROM metamodel. Within the FRaMED plugin, the *Editor UI* handles all user interactions and is implemented employing the Graphical Editing Framework (GEF).[25] Internally, the plugin uses a custom *Ecore* metamodel for the graphical representation of CROM, denoted *Graphical Object Relation Model (GORM)*. This metamodel is tailored towards the graphical aspects of role models, such as *shapes*, *relations*, *segments*, and *bend points*. As a result, FRaMED's implementation only depends on the GORM, simplifying the development and extension of the editor. Nonetheless, whenever a GORM instance is saved (as a `*.crom_dia` file), another plugin is tasked with its transformation to the corresponding CROM (saved as `*.crom` file). The *Transformation* plugin, in turn, utilizes the *Epsilon* framework,[26] a rule-based model-to-model transformation engine, to declaratively specify the translation of GORM files to CROM files. In addition to that, *JUnit 4*[27] and *EMF Compare*[28] are employed to perform a series of transformation test cases, in order to at least partially guarantee that the transformation is sound, i.e., that each valid graphical model is transformed properly to a valid CROM.

[21] `https://eclipse.org`
[22] `https://github.com/leondart/FRaMED/releases/tag/v2.0.3`
[23] `https://eclipse.org/modeling/emf`
[24] `https://github.com/Eden-06/CROM`
[25] `https://eclipse.org/gef`
[26] `http://www.eclipse.org/epsilon`
[27] `http://junit.org/junit4`
[28] `https://www.eclipse.org/emf/compare`

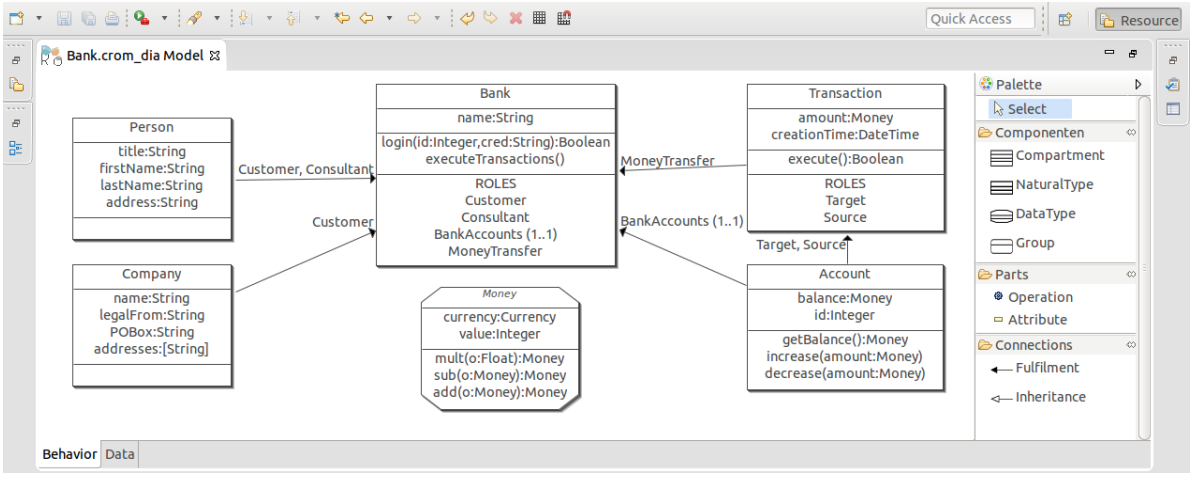

Figure 7.12: Banking application modeled in *FRaMED*, from [Kühn et al., 2016].

Finally, once a CROM file is created, a user can trigger the *Code Generator* plugin to either generate a formal representation of the role model or generate corresponding source code. While the former can be used to validate the designed CROM, the latter can be completed to a working role-based application or a role-based database. Granted, this architecture is rather complex, however, it facilitates separation of concerns by establishing the CROM metamodel as central representation of the foundation for RMLs. This permits the separate evolution of the metamodel, editor, and code generators, as well as the development of additional tools, while assuring that all adhere to the structure and terminology defined in the CROM metamodel and, by extension, the presented formalization.

7.6.2 ILLUSTRATIVE EXAMPLE

In general, FRaMED supports the graphical notation for RMLs, as introduced in Section 7.2. However, the visual representation of CROMs is separated into two distinct levels: the *top-level view* and *focus view*. In the *top-level view* of FRaMED, shown in Figure 7.12, the user can create natural, data, and compartment types; specify their inheritance relation; as well as create and refine the *fills* relation [Kühn et al., 2016]. Model elements are added by selecting them in the palette, dragging them to the canvas, and naming them accordingly. In contrast to the common graphical notation, the *fills* relation is drawn from the player type to the compartment type, first. Afterwards, the role types filled by a *fills* relation can be selected using the *"Fulfill Roles"* dialog (accessible via its context menu) and are then listed adjacent to the *fills* relation. Finally, the user is able to *step into* a selected compartment type by clicking on the *"Step In"* context menu item. As a result, the *focus view* is opened showing the internals of the selected compartment type, as depicted in Figure 7.13. Here, one can create role types, role groups, and relationship types between two role types, as well as specify various role constraints, intra-relationship constraints, and inter-relationship constraints. While most of these model elements are selected and drawn from the palette, intra-relationship constraints are added by selecting the relationship type to constrain and opening the *"Relationship Constraints ..."* dialog via the context menu. Last but not least, to exit the focus view the user simply clicks on the *"step out"* context menu item. Besides all that, whenever the current role model is saved, the corresponding `crom` file is generated. In sum, FRaMED provides all means necessary to create, manipulate and provision role models.

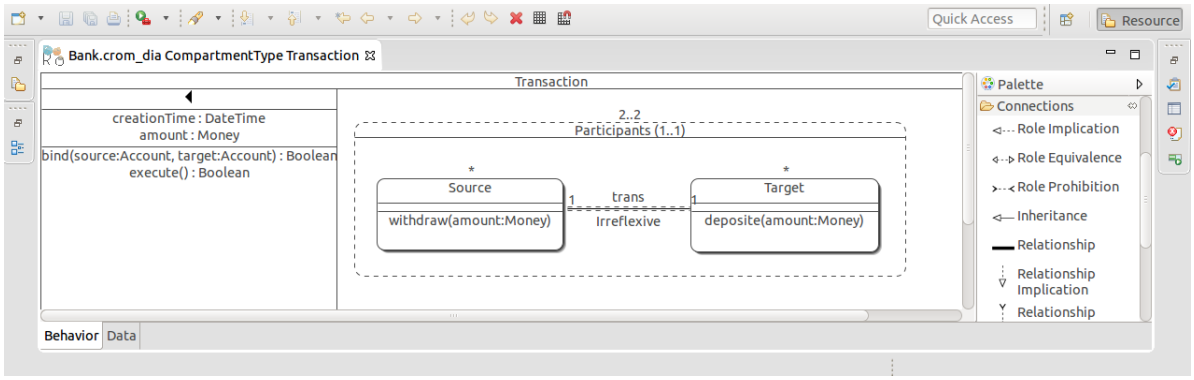

Figure 7.13: Focus view of the *Transaction* compartment type, from [Kühn et al., 2016].

To further illustrate its use, the banking application is modeled (cf. Figure 7.4) again, but now using FRaMED. In the top-level, the natural types `Person`, `Company` and `Account` are defined, as well as the compartment types `Transaction` and `Bank`. Afterwards, each compartment type is further specified by stepping into the corresponding focus view. Within the `Bank` compartment type, the role types `Consultant`, `Customer`, `SavingsAccount`, and `CheckingAccount` are added as well as their relationship types `advises`, `own_sa` and `own_ca`. Similarly, the `Transaction` is refined by adding the role types `Source` and `Target`, the `trans` relationship type, as well as the local constraints. Moreover, the various constraints, e.g. the `Participants` role group and the cardinality constraints, are added in accordance to the modeled banking application (Figure 7.4). The resulting role model of the `Transaction` compartment type is shown in Figure 7.13. Last but not least, after returning to the top-level view the rigid types `Person`, `Company`, `Account` and `Transaction` are declared to fill role types in the `Bank` compartment type. Likewise, the `Account` fills the role group `Participants` of the `Transaction` compartment type. Figure 7.12 depicts the resulting model from the top-level view.[29]

In conclusion, FRaMED represents by far the most convenient way to specify Compartment Role Object Models. Moreover, the enforced separation of *top-level* and *focus view* further tame the visual complexity of role models. Following the argument of Moody [2009], both perspectives successfully modularizes the graphical model by providing a structural overview in the *top-level*, as well as a detailed view on a particular compartment type in the *focus view*. To put it succinctly, FRaMED reduces the complex of role models by modularizing its specification.

7.6.3 ADDITIONAL TOOL SUPPORT

While it is true that a modeling editor alone is useful for domain analysts and researchers, software practitioners strive for more tool supported integrated into the development environment. In particular, practitioners require code generators for various target languages to quickly validate or implement their domain model. To gratify their needs, FRaMED comes equipped with a set of code generators, as illustrated in Figure 7.14. These are available upon right-clicking on a CROM file (`*.crom`) and expanding the *"Generate"* context menu item to *"RSQL Data Definition"*, *"SCROLL Code"*, *"OWL Ontology"*, and *"Formal CROM"*, which triggers the corresponding code generator. Henceforth, each of these code generators is briefly introduced.

The first, code generator translates the given CROM to *RSQL* data definition statements for the *Role-based Contextual Database* [Jäkel et al., 2016]. In general, *RSQL* is role-based extension to *SQL*

[29] The presented bank model can be found at: `https://github.com/Eden-06/FRaMED-Example`

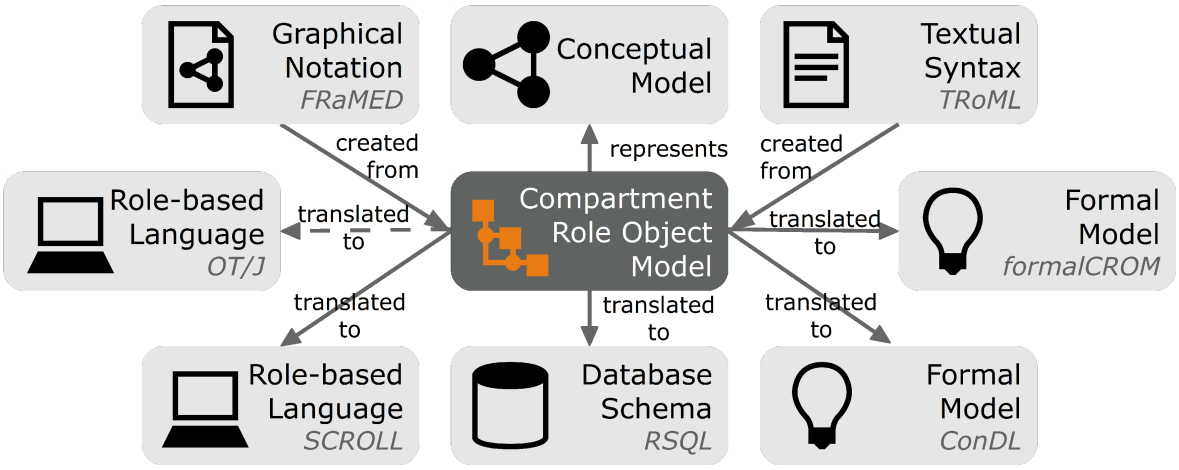

Figure 7.14: Overview of tool support for CROM.

Listing 7.3: Excerpt of the RSQL data definition of the bank model.

```
1 CREATE NATURALTYPE Account(oid BIGINT IDENTITY,id INT,/*...*/);
2 CREATE COMPARTMENTTYPE Transaction (cid BIGINT IDENTITY,/*...*/);
3   CREATE ROLETYPE Source () PLAYED BY (Account) PART OF Transaction;
4   CREATE ROLETYPE Target () PLAYED BY (Account) PART OF Transaction;
5   CREATE RELATIONSHIPTYPE trans CONSISTING OF Source AND Target;
```

that fully incorporates the behavioral, relational, and context-dependent nature of roles, however, lacks direct support for the specification of most constraints, except cardinalities [Jäkel et al., 2015]. Nonetheless, Jäkel et al. [2016] provides a functional role-based database that can be initialized by providing the generated *RSQL* data definitions. For instance, Listing 7.3 showcases the definition of the `Transaction` compartment type including its role types and relationship type, as it would be generated for the previously modeled banking application. Similarly, the second code generator emits class and method definitions for the role-based programming language *SCROLL* [Leuthäuser, 2015]. *SCROLL* is a library for *Scala* that introduces compartments, roles, and relationships to its host language. Moreover, the language supports the definition of local role constraints as well as cardinalities.[30] Accordingly, Listing 7.4 outlines a small portion of the SCROLL code generated for the bank model. It shows the definition of the `Transaction` compartment as *Scala* `case class`. While the generator creates complete classes with both fields and methods, the method implementations must still be provided by a software engineering.[31] In contrast to the previous generators, the latter two translate a specified CROM to a formal representation, to validate its consistency, well-formedness, and compliance. The third generator, for instance, generates a corresponding ontology based on a *two-dimensional contextualized description language (ConDL)* [Böhme and Lippmann, 2015]. This family of description logics are designed to model and reason about contextual knowledge and context-dependent properties. In fact, this description logic allows for verifying the consistency of a given CROM, i.e., whether the specified model has at least one valid instance. For brevity, Listing 7.5 only shows a small portion of the generated ontology. Specifically, it outlines the definition of the `Transaction` including its participating role types and relationship type.

[30] `https://github.com/max-leuthaeuser/SCROLL`
[31] The generator for *Object Teams/Java* code had been implemented by a student, however, does not work reliably.

Listing 7.4: Slice of the generated SCROLL source code.

```
case class Account(balance: Money, id: Integer) { /*...*/ }
case class Transaction(amount: Money, creationTime: DateTime)
     extends Compartment {
  /*...*/
  @Role case class Source(){def withdraw(amount:Money):Unit={/*...*/}}
  @Role case class Target(){def deposite(amount:Money):Unit={/*...*/}}
  RoleGroup("Participants").containing[Source, Target](1, 1)(2, 2)
  val trans = Relationship("trans").from[Source](1).to[Target](1)
  RoleRestriction[Account, Source]
  RoleRestriction[Account, Target]
}
```

Listing 7.5: Extract from the generated context description logic.

```
Class: rosi:CompartmentTypes
   SubClassOf: Annotations: rdfs:label "objectGlobal" owl:Nothing
    DisjointUnionOf: rosi:Transaction, rosi:Bank
Class: rosi:Transaction
    SubClassOf: rosi:PlaysSourceInTransaction
    SubClassOf: rosi:PlaysTargetInTransaction
    #...
    SubClassOf: rosi:transDomainInTransaction
    SubClassOf: rosi:transRangeInTransaction
    #...
```

The last generator emits the formal representation of the given CROM based on the reference implementation, described in Section 7.5. This representation, in turn, can be utilized to automatically evaluate the well-formedness and compliance of the generated `CROM` object and `ConstraintModel` object, respectively, by simply executing the generated formal specification with a suitable *Python* interpreter. Indeed, the *Python* code, generated for the banking application, resembles the example outlined in Listing 7.2, excluding the definition of the CROI. In sum, these generators should satisfy both researchers and practitioners and permit both to investigate, model, and implement role-based software systems. However, because programmers always want to code rather than model, *Textual Role Modeling Language (TRoML)* provides a textual representation and editor for the specification of CROM files.[32] To summarize the various tools, formal frameworks, and languages supported by the CROM modeling language, Table 7.2 compares and evaluates each of them by applying the 27 classifying features of roles (Chapter 2.6). Arguably, The only feature the CROM metamodel disregards, is Feature 8, denoting the *roles can play roles*. In fact, this feature only affects the instance level, as it imposes a partial order on the played roles, to resolve the ambiguities in method dispatch. This partial order, however, can be specified with the *dispatch description language* in *SCROLL* [Leuthäuser, 2015].

In conclusion, *FRaMED* is not just a graphical modeling editor for the CROM modeling language, but also the first role modeling editor that supports all natures of roles and the various constraints presented in the literature. Moreover, it enables the design of role-based software systems by providing additional means for validation and code generation. Furthermore, FRaMED is open source and freely available, in order to let both researchers and practitioners harness the power of role-based modeling. Additionally, FRaMED (v2.0.3) has been packaged as a virtual machine,[33] to guarantee simple installation for practitioners and reproducibility for researchers.

[32] `https://github.com/Eden-06/TRoML`
[33] `http://st.inf.tu-dresden.de/intern/framed/framed-ubuntu.ova`

Table 7.2: Comparison of the languages supported by the CROM modeling language.

Features [Kühn et al., 2014]	**OT/J** [Herrmann, 2005]	**SCROLL** [Leuthäuser and Aßmann, 2015]	**Context DL** [Böhme and Lippmann, 2015]	**RSQL** [Jäkel et al., 2016]	**formal CROM** [Kühn et al., 2015a]	**formal CROM$_I$** *(with inheritance)*	**TRoML**	**FRaMED** [Kühn et al., 2016]
1	■	■	■	■	■	■	■	■
2	⊞	□	■	■	■	■	■	■
3	■	■	■	■	■	■	■	■
4	■	■	■	■	■	■	■	■
5	■	■	■	∅	∅	∅	∅	∅
6	■	□	⊞	■	■	■	■	■
7	□	■	■	■	■	■	■	■
8	■	■	□	■	□	□	□	□
9	□	■	■	∅	∅	∅	∅	∅
10	■	■	■	■	■	■	■	■
11	■	■	■	■	■	■	■	■
12	■	⊞	∅	■	∅	∅	∅	∅
13	■	■	□	■	□	■	■	■
14	⊞	■	■	□	■	■	■	■
15	■	■	■	■	■	■	■	■
16	□	■	■	■	■	■	■	■
17	□	□	□	■	■	■	■	■
18	■	■	□	■	■	■	■	■
19	■	■	■	■	■	■	■	■
20	■	■	■	■	■	■	■	■
21	□	⊞	□	■	□	■	■	■
22	■	■	■	□	■	■	■	■
23	■	■	■	□	■	■	■	■
24	■	■	□	□	⊞	⊞	⊞	⊞
25	■	■	□	■	□	■	■	■
26	■	■	■	■	■	■	■	■
27	⊞	□	□	□	■	■	■	■

■: yes, ⊞: possible, □: no, ∅: not applicable

"As one might expect there is not one ideal way of defining such a concept, but a number of competing approaches."

— Steimann [2000b]

8 FAMILY OF ROLE-BASED MODELING LANGUAGES

Thus far, one would assume that the introduction of a foundational role-based modeling language able to fulfill most of the features of roles is sufficient (cf. Table 7.2), to convince other researchers to adopt and utilize the modeling language. However, when considering the history of RMLs and RPLs, it becomes evident that this is, generally, not the case. In fact, both *TAO* [Da Silva et al., 2003] and *Scala Roles* [Pradel and Odersky, 2009] serve as counter example, i.e., role-based languages that have not been adopted, in spite of their superior feature set. Indeed, Steimann's quote above, reminds us that providing one role definition is not enough to accommodate the various needs and divergent definitions of different researchers. To put it bluntly, after introducing yet another RML, there is still **no** common RML able to capture these divergent definitions of roles. In fact, this appears to be one of the main reasons for the apparent *fragmentation* and *discontinuity* within the research fields on RMLs and RPLs. Furthermore, these issues become a major blocking factor, when different researchers working on role-based approaches are supposed to collaborate with one another. In fact, the research training group on *Role-based Software Infrastructures for continuous-context-sensitive Systems (RoSI)* faced the same problem in the beginning, as each fellow researcher had a different understanding of roles. Thus, instead of requiring all researcher to agree on a common definition of roles, the solution is to introduce a *family of role-based modeling languages* that harmonizes and reconciles their divergent views. In particular, the 27 classifying features of roles were utilized to identify the commonalities and differences of the contemporary role-based languages. Moreover, to directly address the *discontinuity* among contemporary role-based approaches, the *family of metamodels for role-based languages* was introduced in [Kühn et al., 2014]. Specifically, it is able to generate compatible *metamodel variants* for arbitrary role-based languages. Thus, researchers can simply generate a metamodel for their specific approach and, more importantly, for previous approaches they intend to combine or reuse [Kühn et al., 2014]. Furthermore, to tackle the *fragmentation* of the various contemporary RMLs, FRaMED is upgraded to a fully dynamic *Software Product Line (SPL)* for the *family of RMLs*. As a result, researchers can tailor FRaMED to support their particular *language variant*. In conclusion, the *family of RMLs* addresses both the *fragmentation* and *discontinuity*, present in the research on roles.

This chapter is structured, accordingly. Section 8.1 describes the *family of metamodels for RMLs* highlighting its use and implementation. Afterwards, Section 8.2 facilitates the *family of RMLs* by illustrating the extension of FRaMED to a *dynamic SPL*. Finally, Section 8.2.3 discusses the applicability of both the metamodel family and the language family within RoSI.

8.1 FAMILY OF METAMODELS FOR ROLE-BASED MODELING LANGUAGES

Arguably, the *discontinuity* in the research field stems from the incompatibility of the various approaches. However, to be able to freely combine the various contemporary RMLs and RPLs, they require compatible metamodels. Unfortunately, only few approaches actually defined and published their underlying metamodel, e.g., [Da Silva and De Lucena [2004];Kim and Carrington [2004];Herrmann [2007];[Genovese, 2007]. As a result, it becomes infeasible to create or combine the metamodels of two role-based languages. To approach this, the *family of metamodels for role-based languages* has been proposed in [Kühn et al., 2014], such that each member of the family corresponds to the 27 classifying features. Accordingly, a *feature modeling approach* [Kang et al., 1990] has been employed to implement a feature-oriented metamodel generator. In this way, it becomes feasible to generate metamodels for two different approaches, which can then be combined by mapping their sibling metamodels to a merged metamodel. This allows for combining and improving the various contemporary RMLs and, furthermore, paves the way to reconcile the research fields on RPLs and RPLs [Kühn et al., 2014].

8.1.1 FEATURE MODEL FOR ROLE-BASED LANGUAGES

The first step of feature modeling is to generate a feature model as a hierarchical representation of the 27 identified features of roles (cf. Chapter 2.6). This, in turn, elucidates the various implicit dependencies of the classifying features of roles.

Figure 8.1 depicts the resulting feature model for RMLs, published in [Kühn et al., 2014]. To better trace, how the classifying features of roles have been mapped to the feature model, the feature nodes have been annotated with the corresponding number in the feature list (cf. Figure 2.1 and Figure 2.2). In detail, the feature model specifies three main feature arcs, e.g., `Role Types`, `Relationships`, and `Compartment Types`, to group all features dependent on the existence of these modeling concepts. Notably though, those features that are essential for the existence of a concept, are marked as mandatory. The mandatory feature `Naturals` of `Player`, for instance, denotes that role types can always be played by naturals. In conclusion, the feature model encompasses all 27 classifying features of roles. Besides that, the model additionally includes features for Riehle's *role constraints* [Riehle and Gross, 1998], i.e., `Role Implication`, `Role Exclusion`, and `Role Equivalence`, and two options for compartment identity, i.e., `Composite Identitiy`, and `Own Compartment Identity`. In conclusion, the feature model manages to arrange all 27 features of roles with respect to their dependencies to roles, relationships, and compartments. In addition to the dependencies of features visible in the feature model, four additional *cross-tree constraints* [Thüm et al., 2014] have been defined, as shown in Figure 8.2. Conversely, these constraints ensure that a configuration contains all entities on which the `Role Type` depends (8.1 and 8.2), that `Role Equivalence` is included, whenever the `Role Implication` is selected (8.3), and that `Compartment types` are supported, if `Compartments` can be players (8.4).

It follows, then that the presented feature model faithfully captures and elucidates the dependencies of the 27 classifying features of roles. Henceforth, the feature model is used to define a configuration by selecting the various features. For simplicity, two particular configurations (of over 7200 possible configurations) serve as examples, namely the *feature minimal configuration* and the *feature complete configuration.*

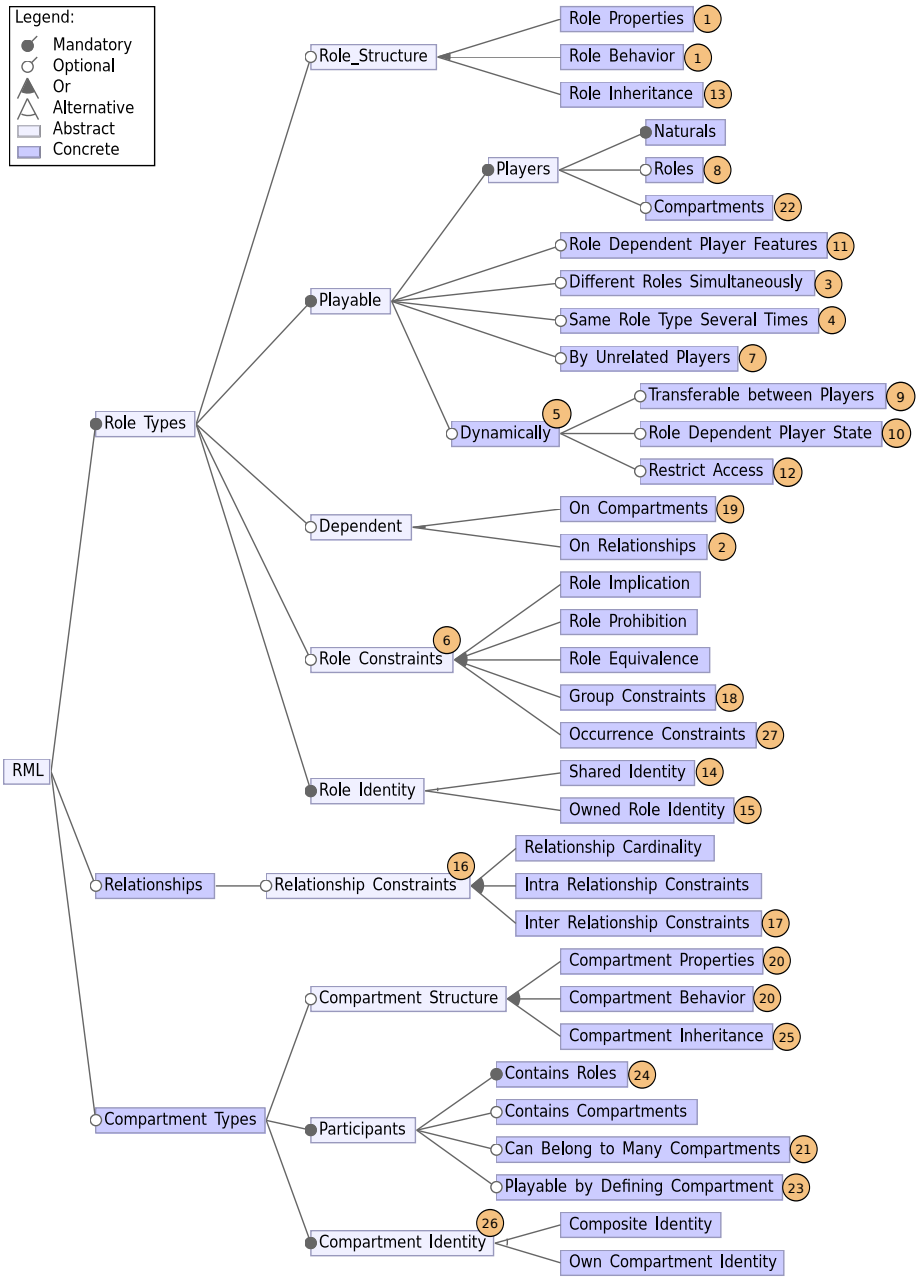

Figure 8.1: Feature model for role-based modeling languages, extended from [Kühn et al., 2014].

$$RoleTypes.Dependent.OnRelationships \Leftrightarrow Relationships \tag{8.1}$$

$$RoleTypes.Dependent.OnCompartments \Leftrightarrow CompartmentTypes \tag{8.2}$$

$$RoleImplication \Rightarrow RoleEquivalence \tag{8.3}$$

$$RoleTypes.Playable.Players.Compartments \Rightarrow CompartmentTypes \tag{8.4}$$

Figure 8.2: Cross-tree constraints of the feature model for RMLs.

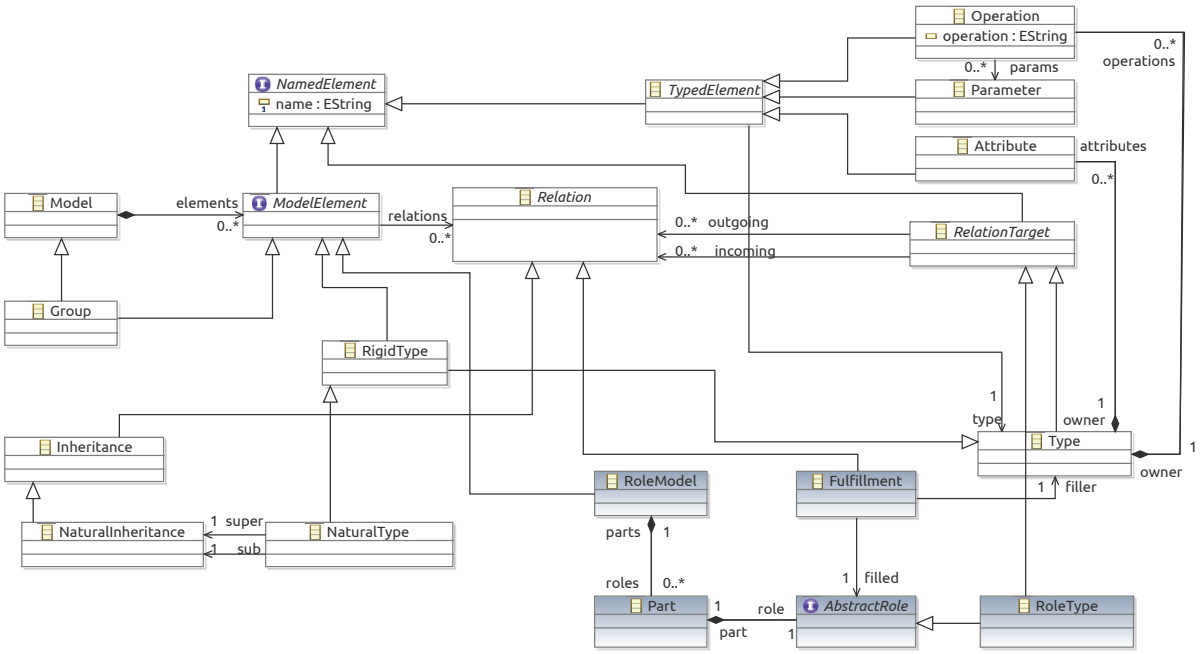

Figure 8.3: Ecore metamodel of a feature minimal metamodel.

8.1.2 FEATURE MINIMAL METAMODEL

In a feature minimal configuration, only mandatory features are selected. Thus, only natural types (with structures and inheritance), which can play role types, exist. Role types, however, are merely annotations, because they only have a name and lack structure, inheritance, and relationships.

In accordance with the minimal configuration of the feature model, Figure 8.3 depicts the metamodel of a minimal RML. Besides the general definition of types, attributes, and operations, this metamodel features a specific `RoleModel` class that represents the default container for all role types (and possibly relationships and constraints). In fact, it is included if the configuration does not include compartment types. Moreover, the metamodel denotes the `RoleType` as `NamedElement` implementing the `AbstractRole` interface. Furthermore, the model specifies the `Fulfillment` class represents *fills* relation from `Type` class to `AbstractRoles`. Finally, the `Part` class links the role model to its participants and holds the lower and upper bound if `OccurrenceConstraint` have been selected. Of course, this model resembles a standard object-oriented metamodel, however, with the additional ability to mark classes with role types.

8.1.3 FEATURE COMPLETE METAMODEL

In contrast to the minimal configuration, a feature complete configuration selects as many features as possible without violating the feature model. However, due to the fact that a metamodel can only reflect features of the model level (M1), all features solely affecting the instance level (M0), have been omitted. As a result, the feature complete metamodel incorporates natural types, role types, relationships, and compartment types as classes. Furthermore, roles can be played by naturals, other roles, and compartments.

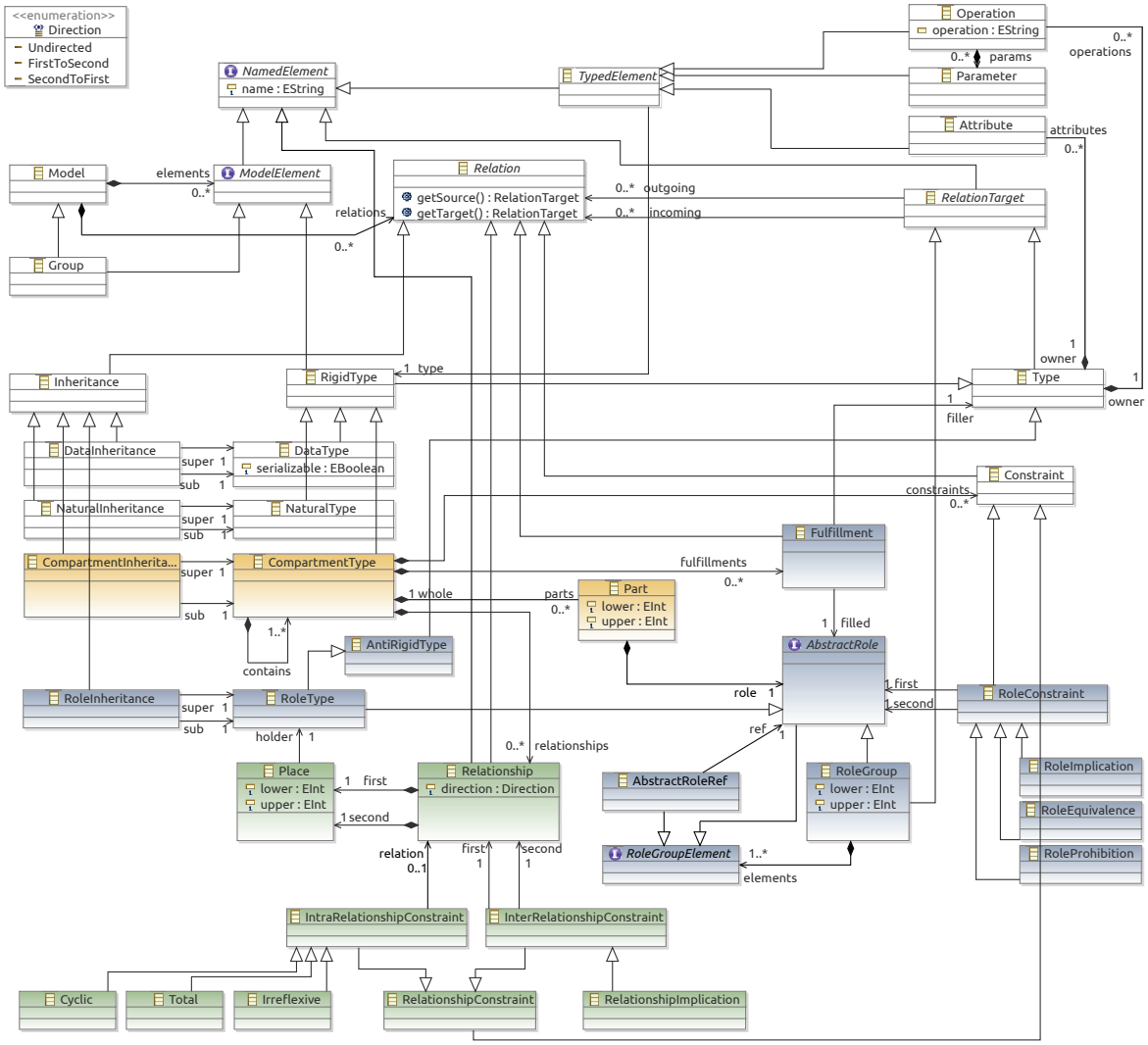

Figure 8.4: Ecore metamodel of a feature complete metamodel.

Figure 8.4 shows the corresponding feature complete metamodel. To illustrate how the three natures of roles are reflected in the metamodel, all classes that contribute to a particular nature of roles are colored accordingly. In detail, blue classes correspond to the behavioral nature, green classes to the relational nature, and orange classes to the context-dependent nature. The classes `RoleType` and `RoleInheritance`, for example, correspond to the behavioral nature, whereas the classes `CompartmentType` and `CompartmentInheritance` contribute to the context-dependent nature of roles. Furthermore, the metamodel encompasses the various relations between the different types, e.g., the *fills* relation, and the various *inheritance* relations, as well as the several local role and relationship constraints. Additionally, this metamodel includes a typical list of intra-relationship constraints and the two inter-relationship constraints, as well as the `RoleGroup` class for the specification of local role groups. In conclusion, the feature complete metamodel represents the unification of the 27 features of roles proposed in the contemporary literature.

8.1.4 MAPPING FEATURES TO VARIATION POINTS

After showcasing both the feature minimal (Figure 8.3) and the feature complete metamodel (Figure 8.4), this section describes how variants can be derived by adding and modifying either classes or references from the feature minimal metamodel. Henceforth, this section describes the mapping from features to the corresponding variation points.

In general, there are five kinds of *variation points* in the metamodel family. The first kind of variation point directly corresponds to classes (highlighted in Figure 8.4), i.e., their existence in the metamodel is directly linked to the selection of a specific feature. To put it bluntly, the following classes directly correspond to a selected feature:

- `On Relationships` (Feature 2),
- `RoleConstraint` (Feature 6),
- `RoleInheritance` (Feature 13),
- `IntraRelationshipConstraint` (Feature 16),
- `InterRelationshipConstraint` (Feature 17),
- `RoleGroup` (Feature 18), and
- `CompartmentInheritance` (Feature 25).

Conversely, the selection of one of these features leads to the addition of the corresponding class together with the respective incoming and outgoing references. The only exception to this rule is the `CompartmentType` class corresponding to Feature 19, which is replaced by a `RoleModel` class, if the feature is deselected.

The second kind of variation point correlates with creation or modification of particular references in the metamodel. For instance, the `filler` reference of `Fulfillment` class either points to `NaturalType`, `RigidTypes`, `Type` or to a generic `Player` interface depending on the selected combination of the Features 8 and 22, which declare whether compartments and/or roles can play roles, as well. Similarly, the references `contains` and `fulfillments` are only included in the metamodel if the configuration includes Feature 24 and Feature 8, respectively.

By contrast, the third kind of variation point changes the inheritance relation of specific classes to change their properties or implemented interfaces. Consider, for instance, the classes `RoleType` and `CompartmentType`, which only inherit (indirectly) from `Type`, if Feature 1 respectively 20 is selected. In other words, they only inherit `attributes` and `operations` from `Type`, if the corresponding feature has been selected. Otherwise, they would inherit from `RelationTarget` and, in case of the `CompartmentType`, also from `ModelElement`. The fourth kind of variation point corresponds to the presence of attributes with a specific class. In fact, the attributes `lower` and `upper` are added to the `Part` class, if *occurrence constraints* (Feature 27) is selected.

The last kind of variation points cannot be captured by standard *Ecore* models, because they correspond to invariants that must be satisfied by instances of that particular metamodel. This holds true for Feature 3, 7, 11, and 23, which broaden the number of valid models. Arguably, an axiom could be added to the *well-formedness rules*, whenever one of these features is deselected. For instance, the axiom $\forall o \in O \exists! r \in R : (o, c, r) \in \mathit{plays}$ could be added whenever Feature 3 is not present in the configuration. Altogether, these variation points are sufficient to generate each member of the metamodel family for a given configuration by iteratively transforming the feature minimal metamodel.

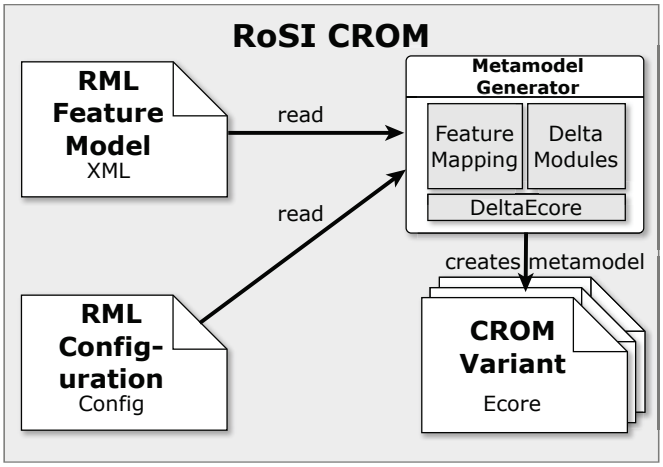

Figure 8.5: Overview of the generator for the metamodel family.

Listing 8.1: Excerpt of the feature mapping for the family of metamodels.

```
1  /*...*/
2  !On_Compartments:
3    <deltas/NotOnCompartments.decore>
4  On_Compartments:
5    <deltas/OnCompartments.decore>
6
7  On_Relationships && On_Compartments :
8      <deltas/OnCompartmentsAndOnRelationships.decore>
9  On_Relationships && !On_Compartments :
10     <deltas/NotOnCompartmentsAndOnRelationships.decore>
11 /*...*/
```

8.1.5 IMPLEMENTATION OF THE METAMODEL GENERATOR

To facilitate the generation of metamodel variants, a corresponding feature-oriented metamodel generator, denoted *RoSI CROM*, has been implemented and published in [Kühn et al., 2014]. In general, the generator was developed utilizing two *Eclipse* plugins designed to support feature-oriented software design: *FeatureIDE* and *DeltaEcore*. In particular, *FeatureIDE*[1] [Thüm et al., 2014] provided the foundation for the metamodel generator by offering dedicated editors for the specification of feature models and configurations. In fact, the feature model, depicted in Figure 8.1, was designed within the *FeatureIDE*. However, the *RoSI CROM* was implemented following a *delta modeling* approach using *DeltaEcore*[2] [Seidl et al., 2014]. Specifically, *DeltaEcore* allows for declaring the changes associated with selecting a feature within *delta modules*. These modules, in turn, manipulate a given base model by adding, modifying, or removing model elements. Additionally, the *feature mapping* connects features to delta modules by specifying their application conditions.[3] Consequently, *RoSI CROM* employs the feature minimal metamodel (cf. Figure 8.3) as its base model; features 34 distinct *delta modules*; and implements the feature mapping, accordingly. Figure 8.5 establishes the general architecture of the metamodel generator *RoSI CROM*.

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide
[2] http://deltaecore.org
[3] https://github.com/Eden-06/RoSI_CROM

Listing 8.2: Delta module for the introduction of comparment types.

```
1  configuration delta "OnCompartments"
2
3  dialect <http://www.eclipse.org/emf/2002/Ecore>
4      requires <../model/crom_l1.ecore>, <Parts.decore> {
5    EClass compartmentTypeEClass = new EClass(name:"CompartmentType");
6    addEClass(compartmentTypeEClass,<crom_l1>);
7    EReference partsEReference = new EReference(
8      eType:<Part>, name:"parts", lowerBound:0, upperBound:-1,
9      containment:true, ordered:true
10   );
11   addEReference(partsEReference, compartmentTypeEClass);
12   EReference wholeEReference= new EReference(
13     eType: compartmentTypeEClass, name: "whole",
14     lowerBound:1, upperBound:1, containment:false
15   );
16   addEReference(wholeEReference, <Part>);
17   makeEReferencesOpposite(partsEReference, wholeEReference);
18 }
```

To further illustrate the implementation of the metamodel generator, Listing 8.1 shows an extract of the specified feature mapping. Here, each mapping consists of a Boolean expression ranging over the set of features and a delta module that is applied, when a given configuration satisfies the Boolean expression. Consider, for instance, a configuration that includes the features *dependent on compartments* and *on relationships*. When providing this configuration to *RoSI CROM*, *DeltaEcore* would apply the delta modules `OnCompartments` (Line 5) and `OnCompartmentsAndOnRelationships` (Line 8) to the base model, as the configuration satisfies the corresponding Boolean expressions. The application of the delta module `OnCompartments`, outlined in Listing 8.2, adds compartment type to the metamodel. Specifically, it creates the `CompartmentType` class (Line 5) as well as the references `parts` (Line 7–10) and `whole` (Line 12–15) between the classes `CompartmentType` and `Part`, such that `whole` is the opposite relation of `parts`. Notably though, most of the other delta modules specify only minor modifications, as most of them only add one model element to the base model.

In conclusion, *RoSI CROM* is a feature-oriented generator able to generate all variants of the family of metamodels. Moreover, due to the good integration of *DeltaEcore* into *FeatureIDE*, the metamodel generator is incredibly easy to use, once the *RoSI CROM* project is imported. Furthermore, the employed delta modeling approach ensures the scalability and evolvability of the metamodel family, as it permits researchers to easily add new features to the metamodel family by providing corresponding *delta modules* and modifying the *feature mapping*. Finally, *RoSI CROM* makes it viable for researchers to generate metamodels for arbitrary role-based modeling and programming languages simply by providing the corresponding configuration to the metamodel generator. Besides all that, *RoSI CROM* is open source and available on *GitHub*,[4] as well.

[4] `https://github.com/Eden-06/RoSI_CROM`

8.2 FIRST FAMILY OF ROLE MODELING EDITORS

While the *family of metamodels* mostly addressed the *discontinuity* among the contemporary role-based languages, this section addresses the *fragmentation* within the research community. In particular, this section facilitates a *family of role-based modeling languages* based on the *feature model for RMLs* to harmonizes and reconcile the divergent definitions of roles found in contemporary RMLs. Indeed, this family of modeling languages not only embodies the various RML using a common graphical notation, but also permits researchers to choose and pick new language variants with respect to the desired features of roles. Yet, what constitutes a *family of modeling languages* and what makes it useful? Similar to modeling languages [Harel and Rumpe, 2004], a family of modeling languages provides a *common syntax* defining the graphical notation and multiple *variations of its semantics* each defining the structure and meaning of individual language variants. Conversely, the *family of RMLs* employs the *graphical notation for RMLs* (Chapter 7.2) to denote its underlying syntax, as well as the *family of metamodels for RMLs* to represent the variants of its static semantics (i.e., its abstract syntax). While this would suffice to establish a language family for CROM models, this family of languages could not be considered useful when reconsidering the properties of useful models, such as *clarity*, *commitment*, *communication*, *control* [Henderson-Sellers, 2012]. Granted, both ontological foundation for roles and the common graphical notation ensure *clarity* and foster *communication* among researcher using the language family. Nonetheless, without proper tool support it is impossible to use the models of a particular RML variant to *control* the design, verification, and implementation of a corresponding role-based software systems.

Up to my best knowledge, there exists no graphical modeling editor able to support the *family of role-based modeling languages*. Thus, this section proposes the development of the *first family of role modeling editors*. That is a feature-oriented, dynamic SPL of graphical modeling editors that enables the flexible configuration of RML variants, as well as the creation, manipulation, validation, and code generation of the corresponding CROM models. To achieve these goals, the Full-fledged Role Modeling Editor (FRaMED) is upgraded to a dynamic, feature-oriented SPL, accordingly. As a result of this extension, researchers will be able to tailor the *FRaMED SPL* to support their particular RML variant that corresponds to their understanding of roles.

8.2.1 DYNAMIC FEATURE CONFIGURATION

Before discussing the underlying architecture of the *FRaMED SPL*, it is important to conceptualize the user's interaction with the modeling editor. In general, it should behave just like the original role modeling editor, *FRaMED*, however, it must permit its users to change the configuration of the underlying RML at any point in time. Additionally, it is conceivable that a user wants to edit multiple CROM models simultaneously, wheres each can belong to a different RMLs, i.e., with different feature configurations. It follows, then that each GORM model must additionally carry the configuration of the corresponding RML. By extensions, the modeling editor must change its behavior dynamically in accordance with the configuration of the currently opened GORM instance. As a result, the *FRaMED SPL* includes a *"Configuration"* tab for each opened editor canvas that allows for modifying the configuration of the underlying RML. As an example, Figure 8.6 showcases the integrated configuration editor highlighting the current configuration of the `BankFamily` model. The implementation of the *FRaMED SPL*, presented henceforth, follows these use cases.

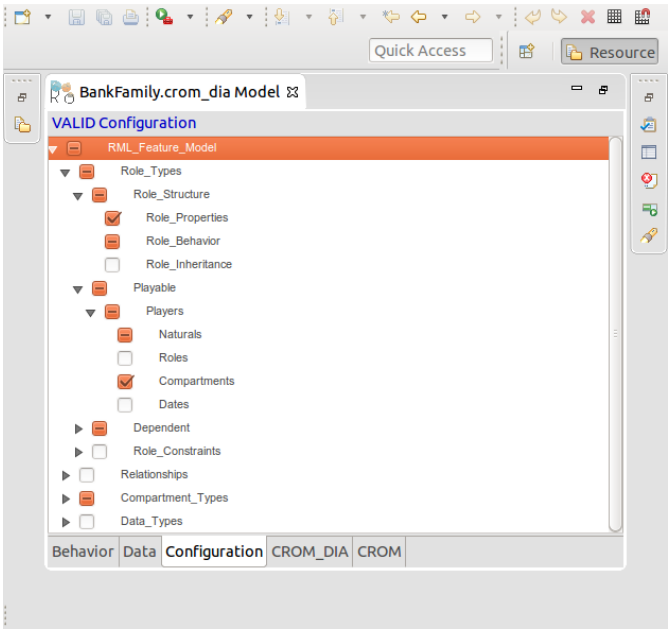

Figure 8.6: Runtime reconfiguration of the FRaMED software product line.

Although the complete *feature model for RMLs* (Section 8.1) could be included, this would also include useless configuration options. This is the case, because not every feature of roles affects the model level and, by extension, the modeling language. Therefore, all features that solely affect the instance level, i.e., Feature 3, 4, 5, 10, 12, 14, 15, and 26, can be omitted from the feature model, as they do not influence the behavior of the modeling editor.[5] The resulting feature model for the *family of role modeling editors* is depicted in Figure 8.7. In addition to that, Figure 8.8 outlines the cross-tree constraints additionally imposed on this feature model. Notably, the constraint (8.9) declares an additional dependency between the feature `Roles` as `Players` and the `ContainsCompartments` feature. In other words, this dependency entails that *roles can only play roles* (Feature 8) if *compartments can contain compartments* (Feature 24) and vice versa. The underlying rational is that a role type defined in a compartment type can only play role types of a contained compartment type. Though I concede that this may be a viable combination of features, I still insist that this would violate the ontological foundation of roles, as, otherwise, the outer role type would be both rigid and anti-rigid at the same time.

8.2.2 ARCHITECTURE OF THE DYNAMIC SOFTWARE PRODUCT LINE

After describing the intended use of the *family of role modeling editors*, this section describes the necessary extensions to upgrade *FRaMED* to a dynamic SPL. As it turns out, the previously established architecture (cf. Figure 7.11) makes it a viable target for the creation of an SPL. In fact, most of the original implementation can be reused as is, whereas only key modules had to be replaced or extended. Accordingly, Figure 8.9 provides an overview on the modified architecture of *FRaMED*. Basically, the editor has been modified in four ways. First and foremost, the editor now loads the *effective feature model for RMLs* and the *edit policy mapping* upon startup. Secondly, the GORM model is extended to incorporate the current configuration of the language variant. Thus, when loading a graphical model (`*.crom_dia`), the editor also loads its feature configuration. Internally,

[5] Table 2.1 and 2.2 indicates the affected meta level for each feature on the right-hand side.

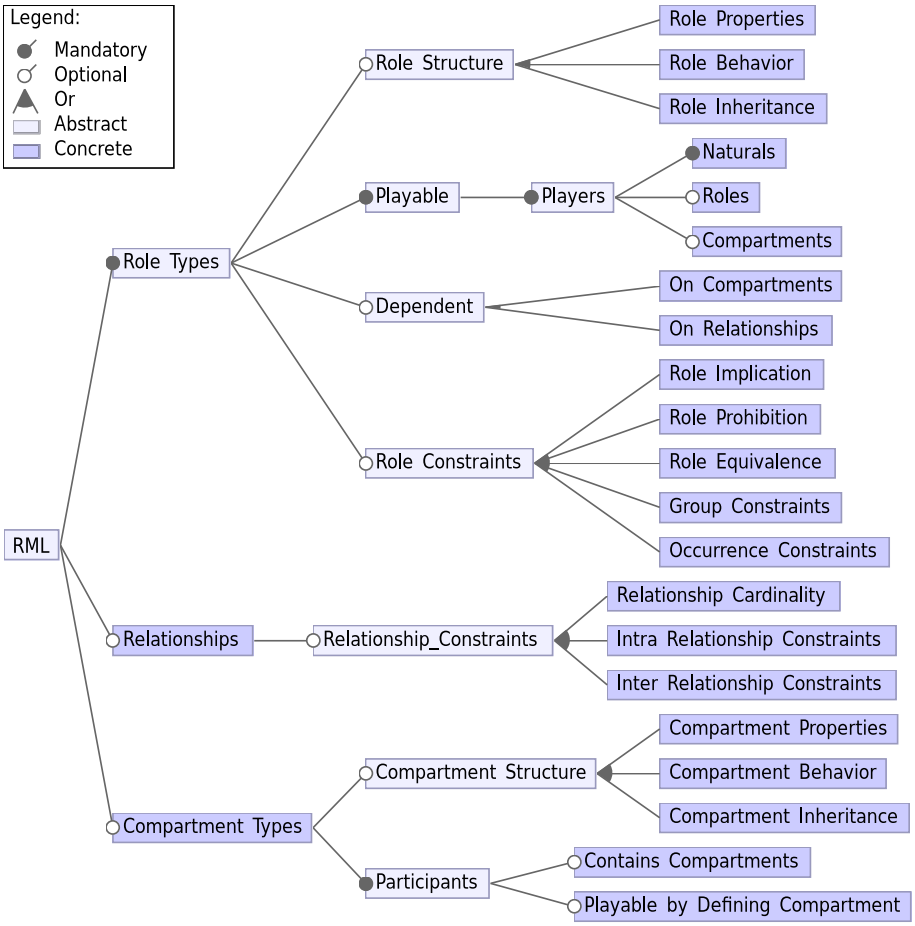

Figure 8.7: Effective feature model for the family of role modeling editors.

$$RoleTypes.Dependent.OnRelationships \Leftrightarrow Relationships \tag{8.5}$$
$$RoleTypes.Dependent.OnCompartments \Leftrightarrow CompartmentTypes \tag{8.6}$$
$$RoleImplication \Rightarrow RoleEquivalence \tag{8.7}$$
$$RoleTypes.Playable.Players.Compartments \Rightarrow CompartmentTypes \tag{8.8}$$
$$RoleTypes.Playable.Players.Roles \Leftrightarrow ContainsCompartments \tag{8.9}$$

Figure 8.8: Cross-tree constraints of the effective feature model for RMLs.

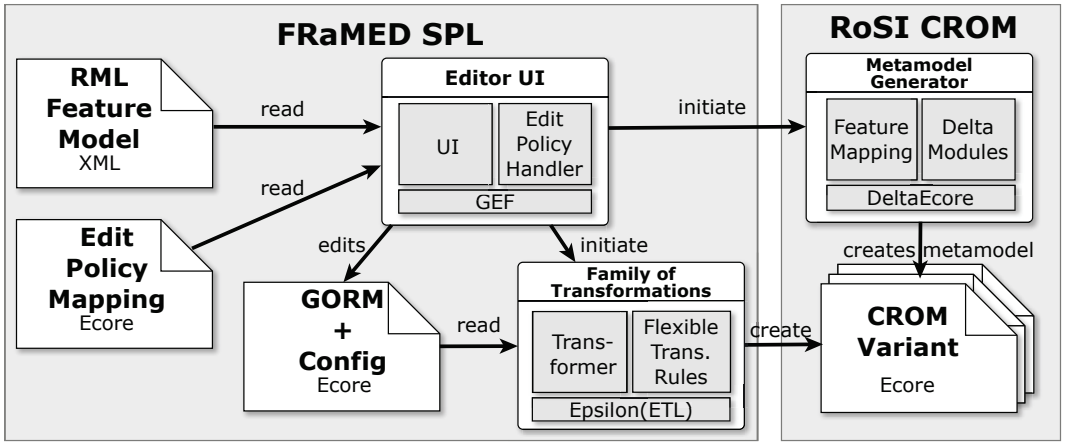

Figure 8.9: Architecture of the family of modeling editors.

each feature configuration is managed by a `Configuration` object (provided by the *FeatureIDE*) that verifies and ensures its validity. Third, the *transformation rules* have been adapted to take the feature configuration of the given GORM instance into account. Finally, *FRaMED SPL* includes the *RoSI CROM* generator as additional plugin to create the corresponding metamodel for a given feature configuration. Accordingly, whenever a GORM instance is saved, the metamodel generator is triggered first to create the corresponding metamodel variant, if not already present. Afterwards, the *transformation* plugin uses this metamodel variant as target to generate the corresponding model.

In general, these modifications suffice to upgrade *FRaMED* to an SPL. However, in order to establish the *FRaMED SPL* as a dynamic product line, the editor must able to dynamically adapt its palette and its edit policies in accordance with the current feature configuration. While it is feasible to implement dynamically changing *palette entries*, it is impractical to modify each and every previously implemented edit policy manually. Therefore, an *edit policy handler* is introduced that loads the *edit policy mapping* and adapts the static edit policies, accordingly. The *edit policy mapping*, in turn, maps features to specific *edit policies*, for instance, to prohibit *inheritance* relations between compartment types if compartment inheritance is not selected. In conclusion, *FRaMED* could be easily extended to a dynamic SPL that, in turn, establishes and supports the *family of RML*.

8.2.3 APPLICABILITY OF THE LANGUAGE FAMILY WITHIN ROSI

According to its proposal, the main goal of the research training school *RoSI* is *"to prove the feasibility of consistent role modeling and its applicability,"*[6] whereas consistency entails that role-based models are utilized throughout the software development process. Consequently, while each *RoSI student* has a different understanding of roles, the *family of RML* and, especially, the *family of role modeling editors* will allow them to easily select the most suitable RML for their particular domain. Arguably, the *family of RMLs* can enable consistent role modeling among the RoSI students. However, to assess its applicability, it basically important to compare their published role-based approaches, i.e., [Kühn et al., 2015a, Leuthäuser, 2015, Böhme and Lippmann, 2015, Jäkel et al., 2016, Chrszon et al., 2016, Taing et al., 2016], with respect to the classifying features of roles. Table 8.1 depicts the results of the comparison and highlights those features that are covered by the effective feature model (cf. Figure 8.7).

[6] `https://wwwdb.inf.tu-dresden.de/rosiproject/rosi-vision`

Table 8.1: Comparison of role-based approaches within RoSI.

Features [Kühn et al., 2014]	**formal CROM** [Kühn et al., 2015a]	**SCROLL** [Leuthäuser and Aßmann, 2015]	**ConDL** [Böhme and Lippmann, 2015]	**RSQL** [Jäkel et al., 2016]	**RRN** [Chrszon et al., 2016]	**LyRT** [Taing et al., 2016]	**formal CROM$_I$** *(with inheritance)*	**FRaMED** [Kühn et al., 2016]	**FRaMED SPL** *(On GitHub)*
1	■	■	■	■	⊞	■	■	■	■
2	■	□	■	■	■	□	■	■	■
3	■	■	■	■	□	■	■	■	■
4	■	■	■	■	■	■	■	■	■
5	∅	■	∅	■	■	■	∅	∅	∅
6	■	□	■	⊞	■	□	■	■	■
7	■	■	■	■	■	■	■	■	■
8	□	■	■	□	□	■	□	□	■
9	∅	■	∅	■	⊞	■	∅	∅	∅
10	■	■	■	■	■	■	■	■	■
11	■	■	■	■	■	■	■	■	■
12	∅	⊞	■	∅	■	⊞	∅	∅	∅
13	□	■	■	□	□	■	■	■	■
14	■	■	□	■	■	■	■	■	■
15	■	■	■	■	■	■	■	■	■
16	■	■	■	■	□	□	■	■	■
17	□	□	■	□	□	□	■	■	■
18	■	■	■	□	⊞	□	■	■	■
19	■	■	■	■	■	⊞	■	■	■
20	■	■	■	■	■	■	■	■	■
21	□	⊞	■	□	⊞	■	■	■	■
22	■	■	□	■	■	■	■	■	■
23	■	■	□	■	■	■	■	■	■
24	⊞	■	□	□	■	⊞	⊞	⊞	■
25	□	■	■	□	□	■	■	■	■
26	■	■	■	■	■	■	■	■	■
27	■	□	■	□	⊞	□	■	■	■

■: yes, ⊞: possible, □: no, ∅: not applicable

Evidently, each of the RoSI students has a slightly different perspective and understanding of roles. Consequently, it is impractical to impose one common RML on all of them, just for the sake of consistent role modeling. On the contrary, if each of them used the *FRaMED SPL* to configure his individual RML, the resulting models could still be shared, reused, and extended.

Naturally, this is true for the approaches *formalCROM* [Kühn et al., 2014], *ConDL* [Böhme and Lippmann, 2015], and *RSQL* [Jäkel et al., 2016], as they mostly operate on the model level and share most of the features. Hence, the models of their individual RMLs are generally compatible. Likewise, the RML variants for both *SCROLL* [Leuthäuser, 2015] and *LyRT* [Taing et al., 2016] would have compatible models. In contrast, only *Roles and Relationships in Reo (RRN)* [Chrszon et al., 2016] would have a rather minimal RML variant. Even though all others could easily reuse his models, it would be difficult to translate a model of one of the other *RML variants* to RRN. This is not surprising, as Chrszon et al. [2016] investigates verification techniques for role-based systems, whereas Leuthäuser and Aßmann [2015] develops a role-based programming language. Nonetheless, it would be feasible to employ the *FRaMED SPL* to create individual *RMLs* for all *RoSI students*. This would not only elucidate the different understanding of roles among the students, but also foster their collaboration.

In conclusion, the *family of RMLs* and its implementation within the *FRaMED SPL* permits researchers to design role models tailor to their use case. In fact, the *FRaMED SPL* is not only the *first family of role modeling editors*, but also the first role modeling editor able to embody all contemporary role-based modeling languages. Moreover, it tames *fragmentation* by enabling researchers to design individual role-based approaches with different features of roles, while maintaining that the models can be shared, combined, reused, and extended by others. Furthermore, the *FRaMED SPL* still provides the necessary tool support for validation and code generation. Although, the *FRaMED SPL* is a research prototype, it is made open source and freely available on *GitHub*.[7]

[7] `https://github.com/leondart/FRaMED/tree/develop_branch`

"After all, role is one of the most elementary terms not only in modelling [...], and it is difficult, if not impossible, to get along without it."
— Steimann [2000c]

9 CONCLUSION

Of course, Steimann is right, when he acknowledges that the notion of roles is so fundamental, so ubiquitous that most people cannot properly define what a role actually is. Throughout the course of this thesis, it became evident that although roles have been used for conceptual modeling for almost 40 years, their underlying nature and formal foundations have not been fully uncovered. Besides Steimann's seminal paper *on the representation of roles* [Steimann, 2000b], this thesis finally establishes the underlying natures of roles, as well as their formal foundation. However, instead of providing yet another role-based modeling language (RML) for conceptual modeling, this thesis established a complete and coherent *family of role-based modeling language*, as well as the corresponding *family of modeling editors*.

9.1 SUMMARY

In particular, this thesis surveyed the contemporary literature on roles in the first part and presented the *foundations for RML*, as well as the *Family of RMLs* in the second part. In detail, Chapter 2 introduced the behavioral, relational, and context-dependent natures of roles; as well as extended Steimann's list of classifying features of roles by including 12 new features that have been retrieved from contemporary role-based languages. Based on these 27 features, a Systematic Literature Review (SLR) was designed and conducted to survey contemporary role-based languages (Chapter 3). In fact, this literature review identified 12 distinct RMLs and 14 RPL published between the year 2000 and 2016. Afterwards, Chapter 4 and Chapter 5 discussed and evaluated each of the contemporary RMLs and RPLs. Finally, Chapter 6 presented the results of the conducted SLR and the corresponding evaluation of contemporary role-based languages. This evaluation, in particular, identified four problems in the research fields on RMLs and RPLs.

1. There is neither a *common understanding* nor *common feature set* shared among the different contemporary role-based modeling and programming languages.
2. The research fields on RMLs and RPLs are characterized by an ongoing *discontinuity* and *fragmentation*. Specifically, most approaches reinvent the role concept without taking the definitions of preceding related approaches into account.
3. Only four RMLs provide a sufficient *formal foundation* for roles able to incorporate *all natures of roles*, i.e. [Da Silva et al., 2003, Genovese, 2007, Liu and Hu, 2009a, Hennicker and Klarl, 2014]. Regardless, none of them is able to *support all classifying features of roles*.

4. Finally, most RMLs and RPLs are *not readily applicable*, due to their complexity, ambiguous terminology, and/or missing tool support. Especially, there exists no feature rich, practically usable graphical modeling editor for an RML.

Consequently, the second part addressed these issues by first providing the foundations for RMLs and then introducing a the *family of RMLs*. Specifically, Chapter 7 established the foundations of RMLs by providing both a comprehensive ontological foundation for roles (Chapter 7.1) and a common graphical notation for RMLs. Above all, this chapter introduced and extended the Compartment Role Object Model (CROM) (Chapter 7.3 and 7.4), a formal framework for conceptual modeling that incorporated the three natures of roles and the modeling constraints. Besides that, Section 7.6 additionally presented *Full-fledged Role Modeling Editor (FRaMED)* as a readily applicable graphical modeling editor for CROM. In contrast, Chapter 8 addressed both the apparent *discontinuity* and *fragmentation* in the research fields on RMLs and RPLs. On the one hand, Chapter 8.1 established the *family of metamodels for role-based languages* that permits researchers to easily generate metamodels for arbitrary role-based languages they intend to combine and/or reuse. On the other hand, Chapter 8.2 finally introduced the *family of RMLs* and upgraded *FRaMED* to a fully dynamic *SPL* to support the language family. Ultimately, both the *metamodel family* and *family of RMLs* have been introduced to tackle the apparent *fragmentation* and *discontinuity* within the research community.

9.2 CONTRIBUTIONS

This thesis presented the following contributions to the field of conceptual modeling, in general, and the field of role-based modeling and programming languages, in particular:

- A thorough literature survey on contemporary RMLs and RPLs published since the year 2000 (Chapter 3).
- The extension of the list of classifying features of roles [Steimann, 2000b] introducing 12 additional features of roles (Chapter 2) [Kühn et al., 2014].
- The introduction of concise ontological foundation for RMLs (Chapter 7.1) [Kühn et al., 2015a].
- The design of a common graphical notation for RMLs (Chapter 7.2) [Kühn et al., 2015a].
- The formalization of both *CROM* [Kühn et al., 2015a,b] and $CROM_I$ as a comprehensive *formal foundation* for RMLs (Chapter 7.3 and 7.3).
- The development of an *award winning*[1] Full-fledged Role Modeling Editor (FRaMED) (Chapter 7.6) [Kühn et al., 2016].
- Introduction of the *family of RMLs* based on the *feature model for RMLs* (Chapter 8.1.1)
- The implementation of the *RoSI CROM* metamodel generator to facilitate the *family of metamodels for role-based languages* (Chapter 8.1) [Kühn et al., 2014].
- The extension of *FRaMED* to a dynamic SPL supporting the creation of language variants of the *family of RML*, and thus introducing the *first family of role modeling editors* (Chapter 8.2).

[1] *Distinguished Artefact Award* of the 9th ACM SIGPLAN Conference on Software Language Engineering.

Table 9.1: Comparison with contemporary role-based modeling languages, extended from [Kühn et al., 2014, 2016].

Features [Kühn et al., 2014]	**Lodwick** [Steimann, 2000b]	**Generic Role Model** [Dahchour et al., 2002]	**TAO** [Da Silva et al., 2003]	**RBML** [Kim et al., 2003]	**Role-Based Patterns** [Kim and Carrington, 2004]	**ORM 2** [Halpin, 2005]	**E-CARGO** [Zhu and Zhou, 2006]	**Metamodel for Roles** [Genovese, 2007]	**INM** [Liu and Hu, 2009a]	**DCI** [Reenskaug and Coplien, 2009]	**OntoUML** [Guizzardi and Wagner, 2012]	**Helena Approach** [Hennicker and Klarl, 2014]	**formal CROM** [Kühn et al., 2015a]	**formal CROM$_I$** *(with inheritance)*	**FRaMED** [Kühn et al., 2016]	**FRaMED SPL** *(On GitHub)*
1	■	■	■	■	■	⊞	⊞	■	■	■	■	■	■	■	■	■
2	■	□	■	■	■	■	□	⊞	■	⊞	■	■	■	■	■	■
3	■	■	■	■	■	■	■	⊞	■	■	⊞	■	■	■	■	■
4	■	■	■	■	□	■	■	■	■	□	∅	■	■	■	■	■
5	■	■	∅	∅	∅	■	■	■	∅	⊞	∅	■	∅	∅	∅	∅
6	⊞	■	∅	∅	■	⊞	⊞	∅	□	□	□	□	■	■	■	■
7	■	□	■	□	□	□	⊞	■	□	■	⊞	■	■	■	■	■
8	□	■	□	□	□	□	□	■	■	□	■	□	□	□	□	■
9	⊞	□	⊞	∅	∅	∅	■	⊞	∅	□	∅	□	∅	∅	∅	∅
10	⊞	■	■	∅	□	∅	⊞	⊞	■	■	■	■	■	■	■	■
11	⊞	■	■	□	■	□	□	⊞	■	■	■	■	■	■	■	■
12	∅	⊞	⊞	■	∅	∅	■	∅	∅	■	∅	■	∅	∅	∅	∅
13	⊞	■	■	■	■	□	□	■	■	⊞	⊞	□	□	■	■	■
14	■	□	□	■	■	■	■	⊞	■	⊞	■	□	■	■	■	■
15	□	■	■	□	□	□	⊞	⊞	□	⊞	□	■	■	■	■	■
16	■	□	■	■	■	■	□	□	□	□	■	□	■	■	■	■
17	□	□	□	□	□	■	□	□	□	□	■	□	□	■	■	■
18	□	□	□	□	□	□	⊞	□	□	□	□	□	■	■	■	■
19	⊞	□	■	□	□	⊞	■	■	■	■	⊞	■	■	■	■	■
20	□	□	■	□	□	⊞	□	■	■	■	■	■	■	■	■	■
21	⊞	□	□	□	□	□	■	■	⊞	□	■	■	□	■	■	■
22	□	□	■	□	□	■	□	■	■	■	□	□	■	■	■	■
23	□	□	□	□	□	□	□	■	□	□	□	□	■	■	■	■
24	□	□	■	□	□	□	□	□	■	□	□	□	⊞	⊞	⊞	■
25	□	□	■	□	□	□	□	■	■	⊞	□	□	□	■	■	■
26	□	□	■	□	□	■	■	⊞	■	■	□	■	■	■	■	■
27	□	□	□	■	■	□	■	⊞	□	□	⊞	■	■	■	■	■

■: yes, ⊞: possible, □: no, ∅: not applicable

9.3 COMPARISON WITH CONTEMPORARY ROLE-BASED MODELING LANGUAGES

This section, finally, compares the RML introduced throughout this thesis by applying the 27 features of roles (cf. Chapter 2.6).

Table 9.1 summarizes the classification of the introduced RMLs approach and compares it to the contemporary RML. In fact, only 23 features apply to modeling languages without operational semantics [Kühn et al., 2014]. As a result, the *formal CROM* supports 18 features of roles, whereas both the *formal CROM$_I$* with inheritance and *FRaMED* support 20 features of roles. Moreover, only one feature is considered *possible to represent*, namely Feature 24, stating that compartments can contain compartments. However, this can be simulated by letting the contained compartment play a role in the container compartment. For instance, the *transaction* compartment is contained inside the *bank* compartment, because it is playing the `MoneyTransfer` role in the *bank* (Figure 7.6). In turn, only Feature 8 is not supported by both *formal CROM, CROM$_I$* and *FRaMED* stating that roles can play roles. Although, this could have been modeled within our formalization, I argue that there is no difference between a football player playing the role of a striker or a person playing both roles at the same time. As it turns out, the underlying rationale that only football players can be strikers would simply be modeled as a role constraint. Consequently, only the *FRaMED SPL* fulfills all 23 applicable features of roles.

At large, the introduced RMLs managed to fulfill significantly more features than any other of the contemporary RMLs.

9.4 FUTURE RESEARCH

Before concluding this thesis, the last section highlights three directions and prospects of future research. First, while this thesis established a family of RMLs, it will be important in the future to introduce a family of RPLs, as well. In this regard, Walter Cazzola and myself proposed a bottom-up approach for the development of Language Product Lines (LPLs) [Kühn et al., 2015c]. In detail, we found that a typical top-down approach, like it has been employed in this thesis, is insufficient for an LPL for RPLs [Kühn and Cazzola, 2016]. Notably though, these were only preliminary results and a more thorough investigation of the family of RPLs is advised. Secondly, although this thesis introduced a novel formal modeling language, its suitability and applicability must still be evaluated. In particular, I want to investigate, whether CROM provides a more suitable graphical representation for *design patterns*, when compared to UML in [Gamma et al., 1994] or *role models* in [Riehle and Gross, 1998]. An initial comparison conducted by Kassin [2015], indicated that the use of CROM could improve the number of visually representable properties of at least nine design patterns. However, further studies including both standard and advanced design patterns must be conducted to establish significant results. Along the same lines, I intend to formalize the *design pattern composition* [Riehle and Gross, 1998]. Naturally, the introduction of compartment types and role groups, establishes a solid formal foundation for this endeavor, especially, since role groups are more expressive then Riehle's role constraints [Kühn et al., 2015b]. Last but not least, the effects of the introduced family of RMLs on the research field must be studied, i.e., whether this family of languages can actually foster collaboration and reuse among the researchers. This, however, requires a continuous examination of the generated CROM variants and their use among researchers, e.g., other RoSI PhD students. In fact, this entails a continuous extension and reevaluation of the *FRaMED SPL*. In conclusion, only time can show the suitability of the presented approach, to **reconcile** and **harmonize** the research field on role-based languages.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ABBREVIATIONS

ACM	Association for Computing Machinery
AOP	aspect-oriented programming
BPM	Business Process Modeling
CROI	Compartment Role Object Instance
CROM	Compartment Role Object Model
CSCW	computer-supported cooperative work
DCI	Data Context Interaction
ER	Entity-Relationship Model
FRaMED	Full-fledged Role Modeling Editor
GORM	Graphical Object Relation Model
IDE	integrated development environment
IEEE	Institute of Electrical and Electronics Engineers
INM	Information Networking Model
JAWIRO	Java with Roles
LPL	Language Product Line
MAS	Multi-Agent Systems
MAS-ML	multi-agent system modeling language
ORM	Object-Role Modeling
OT/J	ObjectTeams/Java
RBAC	Role-Based Access Control
RBML	Role-Based Metamodeling Language
RICA	Role/Interaction/Communicative Action
RML	role-based modeling language
RoSI	Role-based Software Infrastructures for continuous-context-sensitive Systems
RPL	role-based programming language
RRN	Roles and Relationships in Reo
SAS	Self-Adaptive Systems
SCROLL	Scala Roles Language
SLR	Systematic Literature Review
SPL	Software Product Line
TAO	Taming Agents and Objects
UML	Unified Modeling Language

REFERENCES

Adamzadeh, T., Zamani, B., and Fatemi, A. (2014). A Modeling Language to Model Mitigation in Emergency Response Environments. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*, pages 302–307. IEEE.

Al-Zaghameem, A. O. (2010). Extending the Model of ObjectTeams/Java Programming Language to Distributed Environments. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, page 8. ACM.

Arnaudo, E., Baldoni, M., Boella, G., Genovese, V., and Grenna, R. (2007). An Implementation of Roles as Affordances: powerJava. In *WOA 2007: Dagli Oggetti agli Agenti. 8th AI*IA/TABOO Joint Workshop "From Objects to Agents": Agents and Industry: Technological Applications of Software Agents, 24-25 September 2007, Genova, Italy*, pages 8–13.

Aßmann, U. (2003). *Invasive Software Composition*. Springer-Verlag.

Atkinson, C. and Kühne, T. (2002). Rearchitecting the UML Infrastructure. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 12(4):290–321.

Bachman, C. W., Daya, M., Bachman, C. W., and Daya, M. (1977). The Role Concept in Data Models. In *Proceedings of the Third International Conference on Very Large Data Bases*, volume 3, pages 464–476.

Baldoni, M., Boella, G., Genovese, V., Grenna, R., and Van Der Torre, L. (2008). How to Program Organizations and Roles in the JADE Framework. In *Multiagent System Technologies*, pages 25–36. Springer.

Baldoni, M., Boella, G., and Van Der Torre, L. (2006a). Bridging Agent Theory and Object Orientation: Importing Social Roles in Object Oriented Languages. In *Programming Multi-Agent Systems*, pages 57–75. Springer.

Baldoni, M., Boella, G., and Van Der Torre, L. (2006b). powerJava: Ontologically Founded Roles in Object Oriented Programming Languages. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1414–1418. ACM.

Baldoni, M., Boella, G., and Van Der Torre, L. (2006c). Roles as a Coordination Construct: Introducing powerJava. *Electronic Notes in Theoretical Computer Science*, 150(1):9–29.

Baldoni, M., Boella, G., and Van Der Torre, L. (2010). The Interplay Between Relationships, Roles and Objects. In *Fundamentals of Software Engineering*, pages 402–415. Springer.

Baldoni, M., Boella, I. G., and van der Torre, I. L. (2007). Interaction Between Objects in powerJava. *Journal of Object Technology*, 6(2):5–30.

Balzer, S. (2011). *Rumer: a Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH Zürich.

Balzer, S., Eugster, P., and Gross, T. (2008). Relations: Abstracting Object Collaborations.

Balzer, S. and Gross, T. (2011). Verifying Multi-Object Invariants with Relationships. In Mezini, M., editor, *Lecture Notes in Computer Science*, volume 6813 of *25th European Conference on Object-Oriented Programming*, pages 359–383. Springer.

Balzer, S., Gross, T., and Eugster, P. (2007). A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In Ernst, E., editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 323–346. Springer.

Barbosa, F. S. and Aguiar, A. (2012). Modeling and Programming with Roles: Introducing JavaStage. *Frontiers in Artificial Intelligence and Applications*, 246:124–145.

Barbosa, F. S. and Aguiar, A. (2013). Using Roles to Model Crosscutting Concerns. In *Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development*, pages 97–108. ACM.

Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. (1998). The Role Object Pattern. In *Washington University Dept. of Computer Science*.

Bellifemine, F., Poggi, A., and Rimassa, G. (1999). JADE– A FIPA-Compliant Agent Framework. In *Proceedings of PAAM*, volume 99, page 33. London.

Benevides, A. B. and Guizzardi, G. (2009). A Model-Based Tool for Conceptual Modeling and Domain Ontology Engineering in OntoUML. In *Enterprise Information Systems*, pages 528–538. Springer.

Bierman, G. and Wren, A. (2005). First-Class Relationships in an Object-Oriented Language. In *ECOOP 2005 - Object-Oriented Programming*, pages 262–286. Springer-Verlag.

Bloesch, A. C. and Halpin, T. A. (1997). Conceptual Queries Using ConQuer-II. In *Conceptual Modeling—ER'97*, pages 113–126. Springer.

Boella, G. and Van Der Torre, L. (2007). The Ontological Properties of Social Roles in Multi-Agent Systems: Definitional Dependence, Powers and Roles Playing Roles. *Artificial Intelligence and Law*, 15(3):201–221.

Böhme, S. and Lippmann, M. (2015). Decidable Description Logics of Context with Rigid Roles. In *International Symposium on Frontiers of Combining Systems*, pages 17–32. Springer.

Box, G. E. (1979). Robustness in the Strategy of Scientific Model Building. Technical report, DTIC Document.

Chen, P. (1976). The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36.

Chrszon, P., Dubslaff, C., Baier, C., Klein, J., and Klüppelholz, S. (2016). Modeling Role-Based Systems with Exogenous Coordination. In *Theory and Practice of Formal Methods*, pages 122–139. Springer.

Clocksin, W. F. and Mellish, C. S. (2003). *Programming in PROLOG*. Springer Science & Business Media.

Coplien, J. and Bjørnvig, G. (2010). *Lean Architecture for Agile Software Development*. John Wiley & Sons.

Curland, M., Halpin, T., and Stirewalt, K. (2009). A Role Calculus for ORM. In *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, pages 692–703. Springer.

Da Silva, V., Garcia, A., Brandão, A., Chavez, C., Lucena, C., and Alencar, P. (2003). Taming Agents and Objects in Software Engineering. In *International Workshop on Software Engineering for Large-Scale Multi-agent Systems*, pages 1–26. Springer.

Da Silva, V. T. and De Lucena, C. J. (2004). From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language. *Autonomous Agents and Multi-Agent Systems*, 9(1-2):145–189.

Da Silva, V. T. and De Lucena, C. J. (2007). Modeling Multi-Agent Systems. *Communications of the ACM*, 50(5):103–108.

Dahchour, M., Pirotte, A., and Zimányi, E. (2002). A Generic Role Model for Dynamic Objects. In *Advanced Information Systems Engineering*, pages 643–658. Springer.

Dey, A. K. (2001). Understanding and Using Context. *Personal and ubiquitous computing*, 5(1):4–7.

Ernst, E. (2001). Family Polymorphism. In *ECOOP 2001—Object-Oriented Programming*, pages 303–326. Springer.

Ferber, J., Gutknecht, O., and Michel, F. (2004). From Agents to Organizations: An Organizational View of Multi-Agent Systems. In *International Workshop on Agent-Oriented Software Engineering*, pages 214–230. Springer.

Ferraiolo, D., Cugini, J., and Kuhn, D. R. (1995). Role-Based Access Control (RBAC): Features and Motivations. In *Proceedings of 11th Annual Computer Security Application Conference*, pages 241–48.

France, R. B., Kim, D.-K., Ghosh, S., and Song, E. (2004). A UML-Based Pattern Specification Technique. *Software Engineering, IEEE Transactions on*, 30(3):193–206.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Genovese, V. (2007). A Meta-Model for Roles: Introducing Sessions. In *Proceedings of the 2nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies*, pages 27–38. Technische Universität Berlin.

Graversen, K. B. (2006). *The Nature of Roles*. PhD thesis, PhD thesis:/Kasper Bilsted Graversen.–Copenhagen, IT University of Copenhagen Copenhagen.

Graversen, K. B. and Østerbye, K. (2002). Aspect Modelling as Role Modelling. In *OOPSLA'02 Workshop on Tool Support for Aspect Oriented Software Development*.

Graversen, K. B. and Østerbye, K. (2003). Implementation of a Role Language for Object-Specific Dynamic Separation of Concerns. In *AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies*.

Graverson, K. B. (2003). The Successes and Failures of a Language as a Language Extension. In *ECOOP Workshop on Object-Oriented Language Engineering for the Post-Java Era*, Darmstadt, Germany.

Guarino, N. and Guizzardi, G. (2015). We Need to Discuss the Relationship: Revisiting Relationships as Modeling Constructs. In *Advanced Information Systems Engineering*, pages 279–294. Springer.

Guarino, N. and Welty, C. (2000). A Formal Ontology of Properties. In *Knowledge Engineering and Knowledge Management Methods, Models, and Tools*, pages 97–112. Springer.

Guarino, N. and Welty, C. A. (2009). An Overview of OntoClean. In *Handbook on Ontologies*, pages 201–220. Springer.

Guizzardi, G. (2005). *Ontological Foundations for Structure Conceptual Models*. PhD thesis, Centre for Telematics and Information Technology, Enschede, Netherlands.

Guizzardi, G., Pires, L. F., and Van Sinderen, M. (2005). An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modeling Languages. In *International Conference on Model Driven Engineering Languages and Systems*, pages 691–705. Springer.

Guizzardi, G. and Wagner, G. (2005). *Towards Ontological Foundations for Agent Modelling Concepts Using the Unified Fundational Ontology (UFO)*. Springer.

Guizzardi, G. and Wagner, G. (2012). Conceptual Simulation Modeling with Onto-UML. In *Proceedings of the Winter Simulation Conference*, page 5. Winter Simulation Conference.

Guizzardi, G., Wagner, G., Guarino, N., and van Sinderen, M. (2004). An Ontologically Well-Founded Profile for UML Conceptual Models. In *Advanced Information Systems Engineering*, pages 112–126. Springer.

Halpin, T. (2005). ORM 2. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, pages 676–687. Springer.

Halpin, T. (2006). Object-Role Modeling (ORM/NIAM). In *Handbook on Architectures of Information Systems*, pages 81–103. Springer.

Halpin, T. (2007). Subtyping Revisited. In *Proc. CAiSE*, pages 131–141.

Halpin, T. (2009). Object-Role Modeling. *Encyclopedia of Database Systems*, pages 1941–1946.

Halpin, T. A. (1998). Object-Role Modeling (ORM/NIAM). In *Handbook on Architectures of Information Systems*, pages 81–102. Springer.

Harel, D. and Rumpe, B. (2004). Modeling Languages: Syntax, Semantics and all that Stuff. Technical report, Technische Universität Braunschweig.

Harkes, D. and Visser, E. (2014). Unifying and Generalizing Relations in Role-Based Data Modeling and Navigation. In *International Conference on Software Language Engineering*, pages 241–260. Springer.

Harkes, D. C., Groenewegen, D. M., and Visser, E. (2016). IceDust: Incremental and Eventual Computation of Derived Values in Persistent Object Graphs. In Krishnamurthi, S. and Lerner, B. S., editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:26, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

He, C., Nie, Z., Li, B., Cao, L., and He, K. (2006). Rava: Designing a Java Extension with Dynamic Object Roles. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 7–pp. IEEE.

Henderson-Sellers, B. (2012). *On the Mathematics of Modelling, Metamodelling, Ontologies and Modelling Languages*. Springer Science & Business Media.

Hennicker, R. and Klarl, A. (2014). Foundations for Ensemble Modeling – The Helena Approach. In *Specification, Algebra, and Software*, pages 359–381. Springer.

Hennicker, R., Klarl, A., and Wirsing, M. (2015). *Model-Checking Helena Ensembles with Spin*, pages 331–360. Springer International Publishing, Cham.

Herrmann, S. (2002). Object Teams: Improving Modularity for Crosscutting Collaborations. In Akşit, M. and Mezini, M., editors, *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*, pages 248–264.

Herrmann, S. (2005). Programming with Roles in ObjectTeams/Java. *AAAI Fall Symposium Roles, an interdisciplinary perspective*, (FS-05-08):73–80.

Herrmann, S. (2007). A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207.

Herrmann, S. (2010). Demystifying Object Schizophrenia. In *Proceedings of the 4th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, page 2. ACM.

Herrmann, S. (2013). Confined Roles and Decapsulation in Object Teams Contradiction or Synergy? In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 443–470. Springer.

Herrmann, S. and Hundt, C. (2013). ObjectTeams/Java Language Definition (OTJLD) Version 1.3.1. `http://www.objectteams.org/def/1.3.1`. [Online; accessed 28-May-2014].

Herrmann, S., Hundt, C., and Mehner, K. (2004). Translation Polymorphism in Object Teams. Technical report, TU Berlin.

Hu, J., Fu, Q., and Liu, M. (2010). Query Processing in INM Database System. In *Web-Age Information Management*, pages 525–536. Springer.

Hu, J. and Liu, M. (2009). Modeling Context-Dependent Information. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, pages 1669–1672. ACM.

Igarashi, A., Saito, C., and Viroli, M. (2005). Lightweight Family Polymorphism. In *Programming Languages and Systems*, pages 161–177. Springer.

Jäkel, T., Kühn, T., Hinkel, S., Voigt, H., and Lehner, W. (2015). Relationships for Dynamic Data Types in RSQL. In *Datenbanksysteme für Business, Technologie und Web (BTW)*.

Jäkel, T., Kühn, T., Voigt, H., and Lehner, W. (2016). Towards a Contextual Database. In *20th East-European Conference on Advances in Databases and Information Systems*.

Jäkel, T., Weißbach, M., Herrmann, K., Voigt, H., and Leuthäuser, M. (2016). Position Paper: Runtime Model for Role-Based Software Systems. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 380–387.

Kamina, T. and Tamai, T. (2009). Towards Safe and Flexible Object Adaptation. In *International Workshop on Context-Oriented Programming*, page 4. ACM.

Kamina, T. and Tamai, T. (2010). A Smooth Combination of Role-based Language and Context Activation. In Leavens, G. T., Katz, S., and Mezini, M., editors, *Ninth Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 15–24.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

Kassin, K. I. (2015). Aktualisierung des Rollenbasierten Entwurfsmusterkatalogs. bechalor thesis, Technische Universität Dresden, Fakultät Informatik, Nöthnitzer Str. 46, 01187 Dresden, Germany.

Kats, L. C. L. and Visser, E. (2010). The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In Rinard, M., Sullivan, K. J., and Steinberg, D. H., editors, *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 444–463, Reno, Nevada, USA. ACM.

Khabsa, M. and Giles, C. L. (2014). The Number of Scholarly Documents on the Public Web. *PloS one*, 9(5):e93949.

Kim, D.-K. (2008). Software Quality Improvement via Pattern-Based Model Refactoring. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 293–302. IEEE.

Kim, D.-K., France, R., Ghosh, S., and Song, E. (2002). Using Role-Based Modeling Language (RBML) to Characterize Model Families. In *Engineering of Complex Computer Systems, 2002. Proceedings. Eighth IEEE International Conference on*, pages 107–116. IEEE.

Kim, D.-K., France, R., Ghosh, S., and Song, E. (2003). A Role-Based Metamodeling Approach to Specifying Design Patterns. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 452–457. IEEE.

Kim, D.-K. and Lee, B. (2015). Pattern-based Transformation of Sequence Diagrams Using QVT. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1492–1497. ACM.

Kim, D.-K. and Lu, L. (2008). Pattern-Based Transformation Rules for Developing Interaction Models of Access Control Systems. In *High Confidence Software Reuse in Large Systems*, pages 306–317. Springer.

Kim, D.-K. and Shen, W. (2007). An Approach to Evaluating Structural Pattern Conformance of UML Models. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1404–1408. ACM.

Kim, D.-K. and Whittle, J. (2005). Generating UML Models from Domain Patterns. In *Software Engineering Research, Management and Applications, 2005. Third ACIS International Conference on*, pages 166–173. IEEE.

Kim, S.-K. and Carrington, D. (2004). Using Integrated Metamodeling to Define OO Design Patterns with Object-Z and UML. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 257–264. IEEE.

Kim, S.-K. and Carrington, D. (2005). A Rigorous Foundation for Pattern-Based Design Models. In *ZB 2005: Formal Specification and Development in Z and B*, pages 242–261. Springer.

Kim, S.-K. and Carrington, D. (2009). A Formalism to Describe Design Patterns Based on Role Concepts. *Formal aspects of computing*, 21(5):397–420.

Kim, S.-K. and David, C. (1999). Formalizing the UML Class Diagram Using Object-Z. In *International Conference on the Unified Modeling Language*, pages 83–98. Springer.

Kitchenham, B. (2004). Procedures for Performing Systematic Reviews. *Keele, UK, Keele University*, 33(2004):1–26.

Klarl, A., Cichella, L., and Hennicker, R. (2015). *From Helena Ensemble Specifications to Executable Code*, pages 183–190. Springer International Publishing, Cham.

Klarl, A., Mayer, P., and Hennicker, R. (2014). Helena@work: Modeling the Science Cloud Platform. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 99–116. Springer.

Kühn, T., Bierzynski, K., Richly, S., and Aßmann, U. (2016). FRaMED: Full-Fledge Role Modeling Editor (Tool Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 132–136, New York, NY, USA. ACM.

Kühn, T., Böhme, S., Götz, S., and Aßmann, U. (2015a). A Combined Formal Model for Relational Context-Dependent Roles. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, pages 113–124. ACM.

Kühn, T., Böhme, S., Götz, S., and Aßmann, U. (2015b). A Combined Formal Model for Relational Context-Dependent Roles (Extended). Technical Report TUD-FI15-04-Sept-2015, Technische Universität Dresden.

Kühn, T. and Cazzola, W. (2016). Apples and Oranges: Comparing Top-down and Bottom-up Language Product Lines. In *Proceedings of the 20th International Systems and Software Product Line Conference*, SPLC '16, pages 50–59, New York, NY, USA. ACM.

Kühn, T., Cazzola, W., and Olivares, D. M. (2015c). Choosy and Picky: Configuration of Language Product Lines. In Botterweck, G. and White, J., editors, *Proceedings of the 19th International Software Product Line Conference(SPLC'15)*, Nashville, TN, USA. ACM.

Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., and Aßmann, U. (2014). A Metamodel Family for Role-Based Modeling and Programming Languages. In *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 141–160. Springer.

Leuthäuser, M. (2015). SCROLL - A Scala-Based Library for Roles at Runtime. In van der Storm, Tijs and Erdweg, Sebastian., editor, *Proceedings of the 3rd Workshop on Domain-Specific Language Design and Implementation (DSLDI 2015)*, volume abs/1508.03536, pages 7–8. van der Storm, Tijs and Erdweg, Sebastian.

Leuthäuser, M. and Aßmann, U. (2015). Enabling View-Based Programming with SCROLL: Using Roles and Dynamic Dispatch for Establishing View-Based Programming. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 25–33. ACM.

Liu, D., Teng, S., Zhu, H., and Tang, Y. (2014). Minimal Role Playing Logic in Role-Based Collaboration. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1420–1425. IEEE.

Liu, L. and Zhu, H. (2006). Implementing Agent Evolution with Roles in Collaborative Systems. In *Networking, Sensing and Control, 2006. ICNSC'06. Proceedings of the 2006 IEEE International Conference on*, pages 819–824. IEEE.

Liu, M. and Hu, J. (2009a). Information Networking Model. In *Conceptual Modeling-ER 2009*, pages 131–144. Springer.

Liu, M. and Hu, J. (2009b). Modeling Complex Relationships. In *Database and Expert Systems Applications*, pages 719–726. Springer.

Lodwick, F. (1647). *A Common Writing*. Longman Publishing Group. Printed in The Works of Francis Lodwick: A study of his writings in the intellectual context of the seventeenth century. Vivian Salmon, Longman Publishing Group, 1972.

Loebe, F. (2005). Abstract vs. Social Roles – A Refined Top-Level Ontological Analysis. In *In Procs. of AAAI Fall Symposium Roles, an interdisciplinary perspective*. Citeseer.

Masolo, C., Vieu, L., Bottazzi, E., Catenacci, C., Ferrario, R., Gangemi, A., and Guarino, N. (2004). Social Roles and Their Descriptions. In *KR*, pages 267–277.

Mikhajlov, L. and Sekerinski, E. (1998). *A Study of the Fragile Base Class Problem*, pages 355–382. Springer Berlin Heidelberg, Berlin, Heidelberg.

Mizoguchi, R., Kozaki, K., and Kitamura, Y. (2012). Ontological Analyses of Roles. In *Computer Science and Information Systems (FedCSIS), 2012 Federated Conference on*, pages 489–496. IEEE.

Monpratarnchai, S. and Tetsuo, T. (2008). The Design and Implementation of a Role Model Based Language, EpsilonJ. In *5th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON 2008)*, volume 1, pages 37–40. IEEE.

Monpratarnchai, S. and Tetsuo, T. (2011). Applying Adaptive Role-Based Model to Self-Adaptive System Constructing Problems: A Case Study. In *Engineering of Autonomic and Autonomous Systems (EASe), 2011 8th IEEE International Conference and Workshops on*, pages 69–78. IEEE.

Moody, D. (2009). The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *Software Engineering, IEEE Transactions on*, 35(6):756–779.

Murer, S., Worms, C., and Furrer, F. J. (2008). Managed Evolution. *Informatik-Spektrum*, 31(6):537–547.

Mylopoulos, J. (1992). Conceptual Modelling and Telos. In Loucopoulos, P. and Zicari, R., editors, *Conceptual Modeling, Databases, and Case: An Integrated View of Information Systems Development*, pages 49–68. John Wiley & Sons, Inc.

Nelson, S., Pearce, D. J., and Noble, J. (2008). First Class Relationships for OO Languages. In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL 2008)*.

Pearce, D. J. and Noble, J. (2006). Relationship Aspects. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 75–86. ACM.

Piechnick, C., Richly, S., Götz, S., Wilke, C., and Aßmann, U. (2012). Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation-Smart Application Grids. In *ADAPTIVE 2012, The Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications*, pages 93–102.

Pradel, M. and Odersky, M. (2009). Scala Roles: Reusable Object Collaborations in a Library. In *Software and Data Technologies*, pages 23–36. Springer.

Qing, C. and Zhong, Y. (2012). A Seamless Software Development Approach Using DCI. In *2012 IEEE International Conference on Computer Science and Automation Engineering*, pages 139–142. IEEE.

Reenskaug, T. (2011). A DCI Execution Model. Trygve's Webpage.

Reenskaug, T. and Coplien, J. O. (2009). The DCI Architecture: A New Vision of Object-Oriented Programming. *An article starting a new blog:(14pp) http://www. artima. com/articles/dci_vision. html*.

Riehle, D. (1997). A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose. Technical report, Ubilab, Union Bank of Switzerland.

Riehle, D. and Gross, T. (1998). Role Model Based Framework Design and Integration. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 117–133.

Rothenberg, J., Widman, L. E., Loparo, K. A., and Nielsen, N. R. (1989). *The Nature of Modeling*, volume 3027. Rand.

Rumbaugh, J., Jacobson, R., and Booch, G. (1999). *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1st edition.

Rumbaugh, J. E. (1987). Relations as Semantic Constructs in an Object-Oriented Language. In *OOPSLA*, pages 466–481.

Seidl, C., Schaefer, I., and Aßmann, U. (2014). DeltaEcore–A Model-Based Delta Language Generation Framework. In *Modellierung*, pages 81–96.

Sekharaiah, K. C. and Ram, D. J. (2002). Object Schizophrenia Problem in Object Role System Design. In *International Conference on Object-Oriented Information Systems*, pages 494–506. Springer.

Selçuk, Y. E. and Erdoğan, N. (2004). JAWIRO: Enhancing Java with Roles. In *International Symposium on Computer and Information Sciences*, pages 927–934. Springer.

Selçuk, Y. E. and Erdoğan, N. (2006). A Role Model for Description of Agent Behavior and Coordination. In *Engineering Societies in the Agents World VI*, pages 29–48. Springer.

Serrano, J. M. and Ossowski, S. (2004). On the Impact of Agent Communication Languages on the Implementation of Agent Systems. In *International Workshop on Cooperative Information Agents*, pages 92–106. Springer.

Serrano, J. M., Ossowski, S., and Saugar, S. (2006). Reusable Components for Implementing Agent Interactions. In *Programming Multi-Agent Systems*, pages 101–119. Springer.

Shakespeare, W. (1763). *Mr. William Shakespeare's Comedies, Histories, & Tragedies*. Edward Blount and William Jaggard and Isaac Jaggard, London.

Sheng, Y., Zhu, H., Zhou, X., and Hu, W. (2016). Effective Approaches to Adaptive Collaboration via Dynamic Role Assignment. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, 46(1):76–92.

Sheng, Y., Zhu, H., Zhou, X., and Wang, Y. (2014). Effective Approaches to Group Role Assignment with a Flexible Formation. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1426–1431. IEEE.

Smith, G. (2012). *The Object-Z Specification Language*, volume 1. Springer Science & Business Media.

Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading, MA.

Spivey, J. M. (1998). *The Z Notation: A Reference Manual*. Prentice Hall Hemel Hempstead, Oriel College, Oxford, OX1 4EW, England, 2 edition.

Steimann, F. (2000a). *Formale Modellierung mit Rollen*. PhD thesis, Universität Hannover. Habilitation thesis.

Steimann, F. (2000b). On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Data & Knowledge Engineering*, 35(1):83–106.

Steimann, F. (2000c). A Radical Revision of UML's Role Concept. In *UML 2000 - The Unified Modeling Language*, pages 194–209. Springer.

Taing, N., Springer, T., Cardozo, N., and Schill, A. (2016). A Dynamic Instance Binding Mechanism Supporting Run-Time Variability of Role-Based Software Systems. In *Companion Proceedings of the 15th International Conference on Modularity*, pages 137–142. ACM.

Tamai, T. and Monpratarnchai, S. (2014). A Context-Role Based Modeling Framework for Engineering Adaptive Software Systems. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 103–110. IEEE.

Tamai, T., Ubayashi, N., and Ichiyama, R. (2005). An Adaptive Object Model with Dynamic Role Binding. In *Proceedings of the 27th International Conference on Software Engineering*, pages 166–175. ACM.

Tamai, T., Ubayashi, N., and Ichiyama, R. (2007). Objects as Actors Assuming Roles in the Environment. In *Software Engineering for Multi-Agent Systems V*, pages 185–203. Springer.

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014). FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 79:70–85.

Ubayashi, N. and Tamai, T. (2000). RoleEP: Role-Based Evolutionary Programming for Cooperative Mobile Agent Applications. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 232–240. IEEE.

Ubayashi, N. and Tamai, T. (2001). Separation of Concerns in Mobile Agent Applications. In *International Conference on Metalevel Architectures and Reflection*, pages 89–109. Springer.

Van Hentenryck, P. and Deville, Y. (1990). *The Cardinality Operator: A New Logical Connective for Constraint Logic Programming*. Brown University, Department of Computer Science.

Warmer, J. B. and Kleppe, A. G. (1998). *The Object Constraint Language: Precise Modeling with Uml (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional.

Zat'ko, J. and Vranic, V. (2015). Assessing the DCI Approach to Preserving Use Cases in Code: Qi4J and Beyond. In *Intelligent Engineering Systems (INES), 2015 IEEE 19th International Conference on*, pages 51–56. IEEE.

Zhu, H. (2005). Encourage Participants&# 8217; Contributions by Roles. In *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, volume 2, pages 1574–1579. IEEE.

Zhu, H. (2007). Improving Object-Oriented Analysis with Roles. In *Cognitive Informatics, 6th IEEE International Conference on*, pages 430–439. IEEE.

Zhu, H. (2016). Avoiding Conflicts by Group Role Assignment. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46(4):535–547.

Zhu, H. and Zhou, M. (2006). Role-Based Collaboration and Its Kernel Mechanisms. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(4):578–589.

Zhu, H. and Zhou, M. (2008a). Role Transfer Problems and Algorithms. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 38(6):1442–1450.

Zhu, H. and Zhou, M. (2008b). Roles in Information Systems: A Survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(3):377–396.

Zhu, H. and Zhou, M. (2009). M–M Role-Transfer Problems and Their Solutions. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 39(2):448–459.