A Rigorous Approach to Comparison of Representational Properties of Object-Oriented Analysis and Design Methods

Xiaobing Zhang

August, 1997
External Technical Report
ISSN-0836-0227-
1997-410

Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared November 21, 1997

Abstract

Several object-oriented analysis and design (OOAD) methods have been developed in recent years. Although they are more similar to each other than they are to other types of software development methods such as structured analysis and design, the OOAD methods still differ from each other in many aspects. Our interest is in studying the modeling techniques of the OOAD methods and understanding their similarities and differences. This dissertation addresses the problem of systematically comparing the representational properties of the OOAD methods.

The dissertation presents a formal approach to specifying design theories for OOADs' representational properties, and presents a systematic mechanism for determining the similarities and differences between different design theories. Specifically, it describes (1) the formalization of the representational properties under the Theory-Model paradigm, and (2) the development of the formal core theory among the variety of the design theories and the extensions of these theories to the core. The formalization can lead to a rigorous comparison, which provides a deeper insight into the OOAD methods.

We illustrate the approach by applying it to the object model of OMT [RBP+91] and the information model of Shlaer-Mellor OOA [SM88, SM92]. OMT and Shlaer-Mellor OOA are popular among the existing OOAD methods, and are being used in industry. After the formalization and the comparison, we reveal some ambiguities residing in the methods and record their similarities and differences in detail.

Acknowledgements

This dissertation would not have been possible without so many people's indispensable help. To them, I wish to express my sincere thanks.

I am deeply grateful to my supervisor, Professor David Alex Lamb, for his guidance, encouragement and support during the entire course of this work. His confidence in me provided me the perseverance to seek the right way in the intricate maze of academic research. His illuminating comments and ideas greatly improved this dissertation.

I would like to thank the members of my thesis examining committee, Professors Paulo Alencar of Computer Science Department at the University of Waterloo, Jon Davis of Mathematics and Statistics Department, Janice Glasgow, Nick Graham, and the committee chair, Professor Victoria Remenda of Geological Sciences Department. They gave my dissertation a careful review and made numerous valuable comments.

I would like to express my appreciation to the members of my supervisory committee, Professors James Cordy and Andrew Malton, for their valuable advice on the work.

My thanks go to Dr. Arthur Ryman at the Center for Advance Studies, IBM Toronto, for giving me the opportunity to work on the 4Thought project, which is one of the initial inspirations of this work, and for his many constructive comments and suggestions.

My thanks go to Professor Roger Browse for his supervision in the first year of my Ph.D. program, and for his understanding, support, and help in many aspects.

Without doubt, my thanks are extended to Ms. Brenda P. Ifill, Ms. Medha Shukla Sarkar, Dr. Minzhi Zhang, Dr. Ayaz Isazadeh, Mr. Jianchun Zhang and all the members of the Software Technology Lab for their willingness to listen to my ideas, and encouraging me with their helpful comments and suggestions; also I feel very much indebted to them and to many other wonderful

friends at Queen's University for making my years as a graduate student truly enjoyable.

The faculty and staff of the Department of Computing and Information Science deserve my heartfelt thanks. In particular, I am grateful to Ms. Irene LaFleche, Professor Dorothea Blostein, and Ms. Debby Robertson for the help and encouragement they gave to me over the past few years.

Finally, I especially thank my parents, Yanwen Zhang and Jingfen Li, and my sister, Xiaofei Zhang, for being my inexhaustible source of peace and happiness. My very special thanks also go to my husband, Dr. Jun Liu, for his love, support, tolerance and sacrifice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation presents an investigation into the problem of formalizing the object-oriented analysis and design (OOAD) methods and systematically comparing them. This introduction explains why comparison of OOAD methods is necessary, motivates a rigorous approach to this problem, and outlines how this dissertation addresses the problem.

1.1 Motivation

The object-oriented approach is increasingly popular in developing large, complex software systems that can be easily understood, maintained, extended, and reused. It provides a more natural way to manage software systems than traditional software development approaches. An object-oriented method represents a system with a collection of correlated objects. Objects integrate data and functions, and cooperate with each other to accomplish the system's responsibility.

Object-oriented concepts have matured for nearly three decades; however, object-oriented system development methodologies have emerged only in recent years [Boo94, CY91a, CY91b, EKW92, JCJO92, MO92, RBP+91, SM88, SM92, WBWW90]. Since the object-oriented methods are not as

well-developed as other traditional software development techniques such as structured analysis and design, their modeling techniques may not be well-understood.

1.1.1 Differences among Methods

Although they are more similar to each other than they are to other types of software development methods, the object-oriented methods still differ from each other in many aspects. They lack standardization in that the constructs and terminologies used for the object-oriented concepts differ from method to method. The modeling techniques of different methods may also be slightly different in semantics. As a result, users of different methods sometimes find it hard to understand each other.

Thus, there is an increasing need to understand the similarities and differences among the methods. This understanding can be achieved by a rigorous and detailed comparison of the methods.

There are several obvious reasons for comparing methods [CCW91]:

- Comparison discovers real differences between methods. This will suggest which method is more suitable in a particular work environment, or for a particular type of problem.
- Comparison discovers compatibility between methods. This will help develop a degree of interchangeability and integration of the methods.
- Comparison discovers weaknesses in individual methods. This will prompt the improvement of a method, possibly by incorporation of features of other methods.
- Comparison uncovers false assumptions about methods.

Many attempts have already been made to compare and evaluate various object-oriented methods [WBJ90, CMR92, MP92, FK92, vdG92, Fow93, SC93,

YP93, ICO95, Obj95, Zha95]. They are useful in helping to understand the methods; however, some limitations still exist:

- The results depend largely on people who perform the comparisons: they choose the comparison topics according to which aspects of the methods they want to focus on, or according to the application domains of their own interest.
- There are few detailed comparisons: most of them give only an overall comparison which is neither comprehensive nor detailed enough to explain and interpret many differences and similarities among the methods.
- The comparisons fail to explain why the components can be compared and how the conclusions are drawn.

Song and Osterweil [SO94b, SO94a] have observed that because of these limitations, the conclusions drawn by different people sometimes conflict. This deficiency inspires us to seek the way to an objective, detailed, and systematic comparison of object-oriented methods.

1.1.2 The Need for Formalization

Performing a comparison of the object-oriented methods is a difficult task due to:

- *The lack of formalism in current object-oriented methods*: Most of the methods are expressive but do not have formal foundations. Among the methods mentioned at the beginning of this section, only Embley *et al.*'s OOSA [EKW92] provides formally defined models. Based only on the informal descriptions, it is hard to obtain a rigorous comparison because the descriptions of the methods themselves are not rigorous.
- *The variety of constructs and terminologies used by the methods*: The comparability of the constructs and the consistency of the terminologies

among the methods are more difficult to identify when the constructs and the terms are not formally defined.

Therefore, before a comparison takes place, it is essential to establish a formal specification of each method.

Compared with specifications in natural language, formal specifications have many advantages [Geh82, GE86, Fuc92]. Since a formal language has well-defined syntax and semantics, all details of a method must be stated explicitly; thus missing, ambiguous or inconsistent information can be found more easily. In addition, formal reasoning about the specification, especially verification and validation of the designs and the underlying design methods, is possible.

1.1.3 Statement of Thesis

A group of methods may be compared in several aspects. To narrow our focus, we concentrate on the representational properties (i.e., design artifacts and the relationships among the artifacts) of the methods at analysis and design level.

The thesis of this dissertation is

> *Following a rigorous approach, the comparison of the representational properties of the object-oriented analysis and design methods can be precise, detailed and objective.*

1.2 Problem Statement and A Solution

This section defines the problem to be tackled and briefly describes a proposed solution to the problem.

1.2.1 Problem Statement

The problem to be tackled is to develop a systematic comparison of the representational properties of the OOAD methods to better understand the similarities and differences among them. The methods, with perhaps different notations (syntaxes), are to be compared based on their meaning (semantics). To derive as clear and precise a comparison as possible, we shall follow a formal approach.

Methods may be compared and evaluated with respect to three sets of properties: technical, managerial, and usage properties [KC93]. *Technical properties* deal with a method's representational notations and procedures for applying them to solve technical problems. *Managerial properties* are concerned with the organized, cost-effective development of the end product, including such issues as staffing, project planning and cost estimation. *Usage properties* are concerned with practical issues such as the availability of training and tool support. Of the technical properties, the *representational properties*, also called architectural properties, are defined as structures and patterns for specifying and organizing the work products of the analysis and design. They also include the rules that must be followed and the criteria that must be met in specifying the analysis and design. The *procedural properties* are defined as processes for performing the analysis and design. They also include the method guidelines. Our comparison focuses on the representational properties.

The central theme of the work is a formal approach. From a mathematical point of view, an analysis or a design is a complex mathematical object. Its concepts, work products, and rules can be described using set theory and predicate logic.

1.2.2 A Solution

We decompose the problem into two sub-problems and then devise solutions for each of them.

- *Establishing a formal meta-model for each OOAD method by extracting it from the informal description of the method.* The meta-model, expressed in formal notation, provides the specification for the OOAD method. The semantics of the method is determined by its formally defined components (e.g., concepts, work products) and a set of well-formed rules that these components must satisfy. The meta-model is also called the *design theory* of the method.

  Our formalization applies a standard mathematical representation, expressed in Z [Spi89], a formal specification language, to the OOAD methods description.

- *Developing a systematic mechanism for the comparison of the OOAD methods.* A systematic mechanism can help manage comparison activities relatively easily, and also help judge the completeness of a comparison. The design theories serve as a firm theoretical foundation for the comparison. The basic idea of our approach is to extract from the formal specifications of the OOAD methods a *core* which contains features common to the methods being compared, and then characterize each method by an *extension* to the core. The process of comparison is essentially the process of deriving the core and the extensions, which clearly reveal the similarities and differences among the methods.

We apply the approach to the object model of Object Modeling Technique (OMT) [RBP+91] and the information model of Shlaer-Mellor OOA (SMOOA) [SM88, SM92].

Figure 1.1 sketches the processes embodied in the above approach. The processes of both formalization and comparison are iterative.

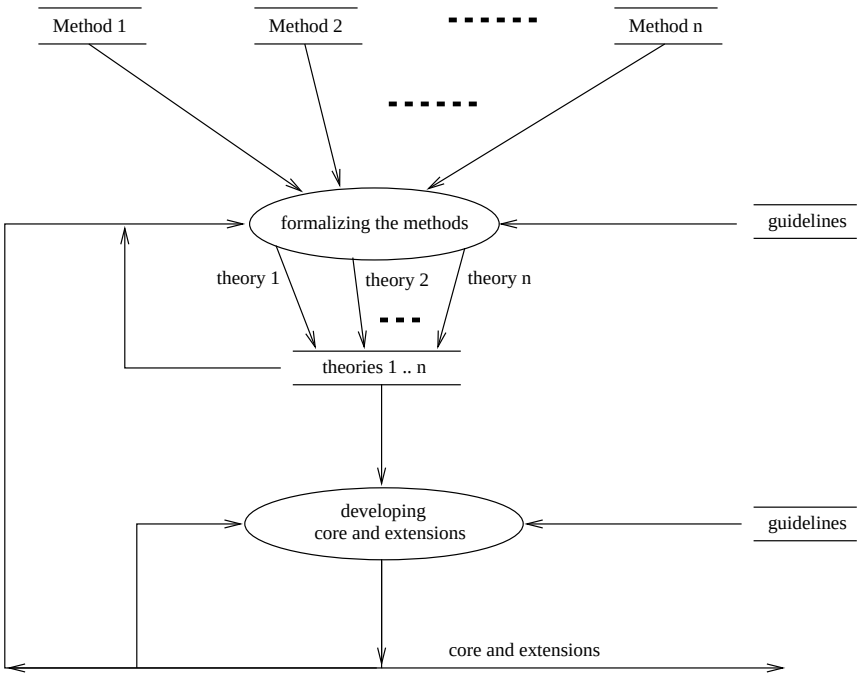


Figure 1.1: Processes of formalization and comparison

1.3 Desirable Properties of A Solution

A set of desirable properties of a solution determines whether or not the solution is satisfactory. Our approach to the comparative study of the representational properties of the OOAD methods should possess the following properties:

General. It should be general enough to accommodate the existing object-oriented analysis and design methods. The comparison framework should be able to cover various aspects with regard to the representational features of the methods.

Precise. It should be precise enough to facilitate identifying similarities and differences. A formal description of the method is precise and rigorous;

it provides a basis for precise comparison. The precision and formality are also preconditions to make the comparative approach objective and systematic.

Objective and detailed. It should provide an objective and detailed comparison in order to avoid any bias result due to the specialized interests of individuals who perform the comparison. By objective we mean that we do not choose the comparison topics. Instead, we compare all aspects with regard to the design theories. This objective and detailed comparison is very important in gaining a thorough and comprehensive understanding of the methods.

Systematic. It should provide a systematic comparison. The comparison is accomplished through a series of steps, each of which is based on a formalism. By systematic we also mean that a well-established mechanism is used to reason about the comparability of the components.

The way of formalizing the methods and systematically comparing their design theories guarantees that these properties can be satisfied.

1.4 Thesis Outline

The next chapter gives background information. It mainly consists of three topics: (1) a review of previous work on studying and comparing software design methods; (2) an introduction to the Theory-Model paradigm [Rym89, RLJ91, LMZ96], which is the theoretical foundation for our formalization approach; and (3) the Z notation, which is the vehicle we use to convey semantics of a design method.

Chapter 3 discusses the approach to establishing a formal design theory of an OOAD method from its informal descriptions. It describes the principles and guidelines for fulfilling the formalization task, the style of formalization,

and how to deal with ambiguity, missing details, and other relevant issues. It also discusses the possible ways of verifying a theory.

Chapter 4 and 5 apply the formalization approach to the object model of OMT and the information model of SMOOA. We obtain a set of Z schemas for each theory.

Chapter 6 presents the systematic comparison mechanism. It gives formal definitions of core and extension, describes the procedure that a comparison process should follow, and provides a set of decision rules for classifying the Z schemas, which represent the design theories, into the core and extensions.

Chapter 7 discusses the comparison of the two formalized methods, i.e., OMT and SMOOA.

In Chapter 8, we conclude the dissertation with a summary of the work that supports the thesis claim and meets the desirable properties of a solution. We also suggest directions for future research and improvements.

Appendix A gives a brief overview of the Z notation.

Appendix B presents a set of rules for classifying a core into four characterizing sets, which further describe the differences among the concepts in the core. It shows the result of applying the rules to the core obtained in Chapter 7.

Chapter 2

Background

This chapter gives a review of the literature and the background material. It mainly consists of three topics: (1) a review of previous work on studying and comparing software design methods. It is relevant to the problem addressed in Section 1.2.2; (2) an introduction to the Theory-Model paradigm, which is the theoretical foundation for the formalization approach; and (3) Z notation, a formal specification language we use to describe semantics of a design method.

2.1 Related Work

This section deals with the related work on two topics: formal approaches to studying software design methods (SDMs) and approaches to the comparison of SDMs.

2.1.1 Formal Approaches to Studying SDMs

Formal methods are increasingly used in the software development life-cycle to specify a system's desired structural and behavioral properties [SFD92, Nic91, PT91, Cly93, LX93, Fei93, Pay93]. However, applying formal approaches to the domain of software development itself is a relatively undeveloped research

area. The work in this area would be establishing a formal meta-model to capture the underlying system description techniques of a method. Most of the meta-modeling techniques surveyed by Moser [Mos95] and Isazadeh and Lamb [IL97] do not have a formal basis. No matter how they vary in textual and graphical forms, these meta-modeling techniques are basically variants or extensions of the entity-relationship (ER) modeling technique [Che76]. Some of them, such as Metaview [Zhu94], provide means of defining consistency constraints and other simple constraints like cardinality. They do not have formal reasoning capability and therefore the correctness of a meta-model is not ensured.

Several attempts have been made to formalize SDMs and the individual methods they include. Bourdeau and Cheng [BC95] provide an algebraic formalization of the semantics for the object model of OMT, using Larch Shared Language. They state that the semantics of an object model O is the set of algebras consistent with the algebraic specification for O. Their work formalizes the primary concepts in the object model such as class, association, aggregation, and subtyping; but it omits many details of the object model, including the concepts like role, operation, key, and so on. On the other hand, it introduces a few extensions to the object model: graph-based notations for attributes, object states, external classes, and an error object for each class. Misic *et al.* [MVL92] establish a formal basis for the extended ER (XER) model concepts, using the Z notation. They formalize data structures, integrity constraints, and update and retrieval operations. Larsen *et al.* [LPT93] provide a formal semantics of data flow diagrams, using VDM. The semantics consists of a collection of VDM functions, which transform an abstract syntax representation of a data flow diagram into an abstract syntax representation of a VDM specification.

Rice and Seidman [RS94] propose a model that formalizes Module Interconnection Languages (MILs). The model is specified by a collection of Z schemas that are fundamental to all of these languages. A particular applica-

tion is described by specifying the values of generic parameters of the schemas and adding application-specific declarations and constraints to the schema definitions. The model provides a formal semantics basis for various MILs. Dean and Lamb [DL94] formalize MILs under the Theory-Model paradigm (which is to be discussed in Section 2.2). The properties fundamental to all languages are recorded in a core theory, and the properties specific to each individual language are in an extension. The essential difference between the two approaches is that the former is top-down and latter is bottom-up. Dean and Lamb do not use generic schemas in the first place; rather, they formalize each individual MIL and then derive the core theory of the MILs from the formalizations. In this way, the core is based on the solid understanding of the MILs.

Other software design methods formalized under the Theory-Model paradigm include Jackson System Development (JSD) by Lamb *et al.* [LJR89], information hiding design method by Lamb and Schneider [LS92], and the information model of SMOOA [Zha94]. These are preliminary work towards the formalization approach discussed in the dissertation.

2.1.2 Approaches to Comparing SDMs

Sol [Sol83] suggests a number of approaches to comparing and evaluating various methods. They can be classified into two basic categories.

A comparison of the first category distills a set of important features from a number of existing methods. The software development methods can then be compared against these features. Using this approach, the success of the comparison and evaluation relies on the features selected and the relative weight given to a feature. Many comparisons, including those discussed in Section 1.1 on page 2, fall into this category.

The other category defines a meta-model, or framework, as a vehicle for communication or as a frame reference in which various methods can be de-

scribed. Sol states that the attractiveness of this approach lies in the fact that implicit, contextual features as well as process aspects of a method can be made explicit. However, the success of the comparison and evaluation relies on the quality and expressiveness of the meta-model. Research that attempts to provide a basis for the comparison and evaluation falls into the second category because most of its effort concentrates on developing a framework [KC93, CCW91, Par93, SO92, SO94a, SO94b, Tse91]. Its aim is to develop a general approach to comparison.

Karam and Casselman's cataloging framework [KC93] lists 21 SDM properties, along with guidelines for their application. The properties are documented by *rating values* and/or *narrative values*. Comparisons are made against these properties. The limitation of this approach resides in the ambiguity of the three rating levels and the narrative descriptions (of the properties), in which a method could not be precisely described.

Cameron *et al.*'s classification scheme [CCW91] has similarity to the cataloging framework in that it also suggests a set of properties. Its distinguishing feature is that it proposes two schemes to compare notations. One scheme is to derive an algorithm for translating one notation (or portion of that notation) into the other. The other scheme is to define a more abstract notation to which both notations can be reduced, and to design algorithms for the reduction of both notations to the underlying notation. Unfortunately, a general approach to translation and reduction is not given; rather, translations are carried out case by case, informally through examples with graphical notations and textual explanations. Moreover, a foundation for determining the equivalence of two notations is not provided.

Parsons' knowledge representation framework MIMIC [Par93] is used to examine mechanisms and constructs found in object-oriented methods. The cognitive foundation embedded in MIMIC is that information systems represent human knowledge about *things* in some domain, and the knowledge is organized into categories. MIMIC consists of a number of constructs for

representing knowledge about things. It encapsulates the concepts of *object*, *attribute* (*structural, relational*, and *behavioral*), *state*, *event*, *potential class*, and *class structure*. Derived from the results of research in cognitive sciences, MIMIC is independent of any object-oriented method. In this sense, it can be viewed as an "ideal" model used to compare object-oriented methods. This ideal model may reflect the knowledge representation scheme naturally; however, sometimes it is too simple to evaluate complex representational notations in software modeling. Further, informal design methods are not formalized before being compared against the formal framework.

Song and Osterweil's comparison approach [SO92, SO94a, SO94b], called CDM (Comparing Design Methodologies), is more sophisticated than those of others. CDM differs from the above approaches (which compare SDMs in a casual way) in that it helps compare SDMs more systematically. It emphasizes discovering differences between similar design components. CDM comprises several steps:

1. Decide and define the aspects of SDMs to be compared, and develop or adopt a modeling formalism and classification framework.

2. Develop and validate the *meta-models* of SDMs. The modeling formalisms adopted include software process modeling, mathematical set-definition notation, Ada-like notation, ER modeling and IR (inheritance relation) modeling formalisms.

3. Classify the components of the SDMs within the classification framework. The classification helps to show which components address the same or similar design issues.

4. Select the comparison topics based on the classification.

5. Make the comparisons.

From the viewpoint of our proposed approach, we find CDM has two limitations. Firstly, although the meta-model developed during the first two steps is used to "formalize" the design methodologies, it does not provide a thorough formal foundation. CDM uses ER and IR models to "formalize" an artifact model (which comprises representational properties); nevertheless, ER and IR models themselves are not formal in the mathematical sense, and they cannot express many kinds of constraints. Secondly, the classification framework is not formally described. The framework has a hierarchical structure. At each level, a set of types is used to characterize the parts of the methods. A decomposition of types at the next level represents a more detailed description of the methods. The definition of types in the framework is in textual form, which is by no means formal.

The above discussion shows that Parsons, Song and Osterweil urge the need of a formal theoretical foundation to compare software modeling methods. But their work remains informal to a certain extent.

Tse's unifying framework [Tse91] establishes a theoretical link between structured analysis and design methodologies. Tse defines a term algebra of structured systems, which can be mapped by unique homomorphisms to a DeMarco algebra of data flow diagrams, a Yourdon algebra of structure charts and a Jackson algebra of structure texts. He also finds that the proposed term algebra as well as the deMarco, Yourdon and Jackson notations fit into a category-theoretic framework. DeMarco data flow diagrams can be mapped to term algebras through free functors. Conversely, specifications in term algebras can be mapped to other notations such as Yourdon structure charts by means of functors. Tse proves that, under the framework, a design developed using one method can be transformed to a design supported by another method. The limitation of the unifying framework is that it concentrates on examining compatibility among the methods but ignores the differences.

Our approach has advantages over the above approaches in that it is based on formal specifications of design methods, it can discover very detailed aspects

of the methods, it provides a systematic mechanism, and it reveals not only similarities but also differences among the methods.

It is worthwhile to mention the work done by Dean and Lamb [DL94], which is an experiment of the initial form of the proposed comparison approach. They conclude that the approach can be used to formalize and compare the structural representation of MILs. The conclusion demonstrates that our approach is feasible for comparing notations on a relatively smaller scale in comparison with the more complicated SDMs.

Last, we present a brief discussion on the Unified Modeling Language (UML) [Rat97]. The UML is a visual modeling language for specifying, visualizing, and constructing the artifacts of object-oriented systems. One of the key motivations behind the development of the UML was to create a notation and semantics that adequately addresses all scales of architectural complexity, across all domains. The UML integrates the concepts used by Booch [Boo94], OMT [RBP+91], and OOSE (use-cases) [JCJO92], as well as other object-oriented methods. Its specification consists of two parts:

- UML Semantics. A metamodel that specifies the abstract syntax and semantics of UML object modeling concepts.
- UML Notation. A graphic notation for the visual representation of the UML semantics.

The language is still under development.

Despite the emergence of the UML, our idea of formalization and comparison is still necessary. The UML semantics, i.e. the metamodel, is defined in UML class diagrams, Object Constraint Language (OCL), and English prose. OCL is a specification language that uses simple logic for specifying invariant properties of systems comprising sets and relationships between sets. OCL is used in the UML to formally specify simple constraints on the concepts. Some parts of the semantics which are described in natural language could be more precise if formalized under our approach.

The UML claims to consolidate a set of core modeling concepts that are generally accepted across many current methods and modeling tools. We have not yet seen justification for the UML concepts, so we have to trust that it accurately reflects the original methods. The comparison of the UML and the original methods can show what concepts are adapted by the UML and what are omitted. The UML is still one of the existing object-oriented methods. Some OOAD methods are used in industry but are not merged into the UML, such as Shlaer-Mellor OOAD. Comparing these methods with the UML is still essential in understanding their similarities and differences. In addition, the comparison results are important for defining interchange of tools that support different methods, or customizing meta-CASE tools. Furthermore, migration from the systems developed under other OOAD methods to the UML requires knowledge of the commonality and differences between the methods, which our work can provide.

2.2 The Theory-Model Paradigm

The *Theory-Model Paradigm* [Rym89, RLJ91, LMZ96] is a way of describing and understanding design methods, designs, and design verification. The ideas originate in the "model theory" [CK73] of mathematics. There, a *theory* is a collection of expressions or statements made in an formal language of uninterpreted operations and symbols. For example, *group theory*, having expressions over variables, one constant, one unary operation, and one binary operation, is a collection of equations induced by the usual rules of equality and a set of five axioms. An *interpretation* of a theory gives meaning to the uninterpreted symbols of the theory. For example, an interpretation of group theory might involve letting variables vary over integers, letting the constant be 0, the unary operation be negation, and the binary operation be addition. A *model* of the theory is an interpretation that satisfies all the axioms. Thus, the integers under addition are a model of group theory. A different interpretation

might not be a model of the theory: you may not let the binary operation be multiplication over the integers.

These ideas are applied to software design in the following way:

- A design method corresponds to a theory: it introduces certain categories; its design rules and their consequences correspond to theorems about the categories.
- A design, that is the work produced by a designer, corresponds to an interpretation: a particular set for each category.
- A design is "correct" if it follows the design rules; correspondingly, an interpretation is valid (i.e., is a model of the corresponding theory) if it satisfies all the axioms.

The development of a design theory and the design verification proceeds as follows:

1. Develop a formal description of the design method, using mathematical notation where possible for precision. Lately, we have been using the Z notation for this step. It is the hardest step, requiring about 80% of the effort [LMZ96].
2. Develop a data model for the theory. A tool supporting the design method must be able to represent the design information, and so it is appropriate to develop a data model for it using typical data modeling methods. At present we use ER diagrams [Che76] for this purpose. There is usually a close correspondence between some of the sets described in the mathematical theory and some of the entity sets of the ER model, but the relationship is not necessarily direct.
3. Develop an executable form for the checks of whether any axioms are violated. In the past we have used Prolog for this purpose; in the future we will consider using Gödel [HL94].

There are feedback loops among the three steps; while developing the data model, for example, one might find a simpler method of expressing a concept from the formal description.

2.3 The Z Notation

Several formal specification techniques have been proposed so far and most of them are based on some well-developed mathematical notations [Win90]. The Z notation [Spi89], one of the most popular techniques among these, is being used as a tool for expressing the mathematical aspects of SDMs. It is based on a typed set theory and the first order predicate logic. For precision, we use the Z notation to write formal specifications of design methods.

There seems to be a perceived difference between model-oriented specification languages, like Z and VDM [AI91], and algebraic (or property-oriented) specification languages, like the Larch Shared Language (LSL) [GH93], regarding their applicability to the specification of software systems. Model-oriented specification languages are assumed to be suited better for the description of state based systems, abstract machines or abstract data types with state, while algebraic specification languages are assumed to be better for (functional) abstract data type specifications.

What formal language we choose as the vehicle of formalization is mainly determined by the aspects of the methods we will formalize as well as the availability of supporting tools of the formal language.

The set-oriented style of Z matches our requirement. We are going to specify the representational features (which includes concepts, work products, relationships, and rules) of a method, ignoring its modeling process (which describes a designer's behavior). Moreover, Z provides a structuring mechanism, the "schema", which we can use to organize our specifications.

Available with Z is a type-checker, *f*UZZ, which can help catch many simple

mistakes in drafts of the mathematical expressions[1]. For example, we can check the Z specification for compliance with the Z scope and type rules; we can also check that the specification itself is consistent in syntax.

Appendix A provides a brief overview of the Z notation.

[1]Z/EVES [MS95] is a more powerful tool for parsing, type checking, well-definedness checking, and proving theorems about Z specifications. However, it was not available to us until very late.

Chapter 3

Approach to Formalizing the OOAD Methods

This chapter discusses the approach to formalizing the representational properties of the OOAD methods under the Theory-Model paradigm[1]. A design theory specifies the semantics of a design method. A method can be formalized in many different but logically equivalent ways. The fundamental principle for formalization is to make the descriptions of design theories precise, simple, and readable. This chapter describes the principles and guidelines for fulfilling the formalization task, the style of formalization, and how to deal with ambiguity, missing details, and other relevant issues. It also discusses the possible ways of verifying a theory. The formalization process is iterative.

3.1 Investigating the Original Materials

The methods we study are documented in book format. Design categories and rules are expressed in various forms in the books, such as natural language,

[1]Most of the discussion in this chapter has previously appeared in the technical report [LMZ96].

diagrams, charts, and tables. They may be given formal definitions or explained only via examples. The informal nature of method descriptions means that misunderstandings are possible and that formal verification of a model is not possible. Graphical notations may not guarantee a precise understanding of a method. Therefore, the first step of the formalization is to develop a reasonable understanding of the method through a careful examination of the original source material, to avoid incorrect interpretations of the informal descriptions.

The mapping from informal to formal is typically achieved through an iterative process not subject to proof. At all times, the formal specification is only a mathematical representation of the method's architectural properties. Any inconsistencies in the method would be preserved in the mapping. The formalization is to improve the degree of precision of the method, but not to improve the method itself. For every lack of precision in the original, some choice must be made about how to fill in missing details; however, the formalization should not diverge from the original.

A formalization requires decisions about how to represent things precisely while the original materials define them imprecisely. In documenting such decisions, we cite the original sources, supplying page numbers or page ranges for the discussion of the imprecise concepts.

3.2 Basic Rhetorical Principle

The formalization is fundamentally intended to give precise but readable descriptions of design methods. This means that concerns of rhetoric (the art of conveying meaning) are at least as important as the associated technical concerns.

A basic rhetorical principle for this work is that the formal description cannot stand on its own. Between the original informal description and our formal description there must be additional material, in informal but techni-

cally precise language. This is called the *technical description*.

The technical description should be able to stand on its own, with the formal description to make it precise. In principle, it should be possible to obtain a sensible (though possibly imprecise) view of the design theory by omitting the formal description and leaving only the technical one. While many styles of composition are appropriate, in our view it is better to develop the technical and formal descriptions concurrently. The technical description must be written eventually, and each helps make the other clearer as choices are taken and revised.

The situation somewhat resembles that of a software requirements document and the code that implements it. The requirements correspond to the informal description; the technical description corresponds to the software design document and code comments; the formal description corresponds to the code. Well written design documents and comments can stand alone. Software engineers generally accept that they should write design documents and comments with the implementation.

3.3 The Formalization Task

This section describes the fundamental principles of developing and recording design theories. We use the Z notation to capture information of design theories; and use the formalization of OMT's object model as an example[2].

3.3.1 Design-Time Concepts and Run-Time Concepts

The concepts involved in a design method are classified into two kinds: run-time and design-time concepts. *Run-time* concepts are concerned with the elements that build up a run-time system; such concepts include object, at-

[2]The Z code for OMT presented in this Chapter is a simplified version of that in Chapter 4.

tribute value, and link in OMT. *Design-time* concepts are concerned with patterns of run-time elements; such concepts include class, attribute, operation, association, aggregation, and generalization in OMT.

The theory of a design method is a formalization of the design-time concepts and the design rules. It represents the formal semantics of the method. Since the two kinds of concepts are closely associated with each other, the semantics of some design-time concepts could not be complete without the consideration of run-time concepts. For example, in OMT the objects (i.e., instances of classes) which play the role on a "many" side of an association sometimes are ordered. Such a role is called "ordered role". With only the design-time concepts, we can introduce ordered roles by stating that they are "multiple roles":

$$orderedRole : \mathbb{P}\, ROLE$$
$$\hline$$
$$orderedRole \subseteq multipleRole$$

After introducing the run-time concept "object" and the relation *mulOfRole* among the run-time and the design-time concepts, we are able to explain further the meaning of an ordered role:

$$ordering : ROLE \leftrightarrow \text{iseq}\, OBJECT$$
$$\hline$$
$$\text{dom}\, ordering = orderedRole$$
$$\forall\, r : orderedRole;\ os : \mathbb{P}\, object \mid r \mapsto os \in mulOfRole \bullet$$
$$(\exists_1\, os' : \text{iseq}\, object \mid os = \text{ran}\, os' \bullet r \mapsto os' \in ordering)$$

This shows that the complete semantics of ordered roles is expressed in terms of both the design-time and the run-time concepts.

Run-time concepts in our formalization are only a supplement for specifying the semantics of a design theory. In other words, the theory about run-time concepts is not yet complete. However, we can say it is complete from the viewpoint that a run-time instance should conform to the constraints on the

design-time model. That is, the semantics of a run-time instance is implicitly expressed in the theory about the design-time concepts as well. For example, in OMT, each class has a name and this property is formalized as follows.

$$nameOfClass \in class \rightarrowtail\!\!\!\!\!\!\rightarrow className$$

This also implies that the instances of a class have a name, which is exactly the name of the class.

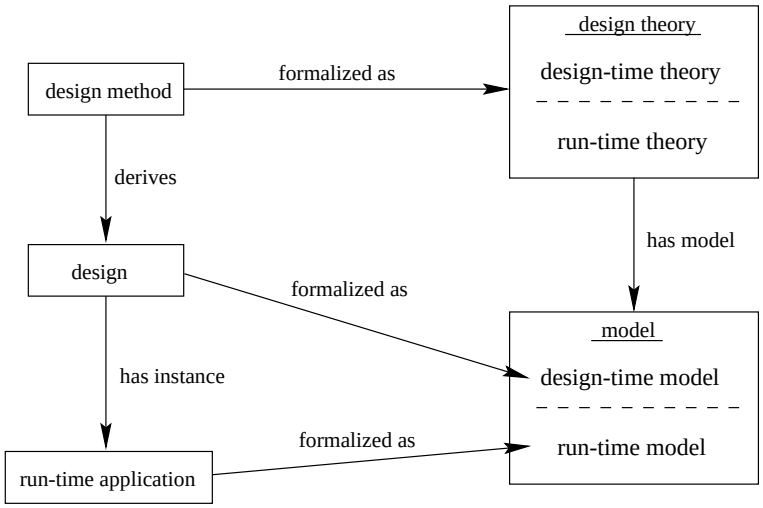


Figure 3.1: Relationships among design method, design, run-time application, design theory, and model.

Our approach deals with concepts at three different levels: design method, design, and run-time application. Figure 3.1 illustrates the relationships of these three kinds of concepts with design theory and model. A design theory is composed of a design-time theory and a run-time theory; a model is composed of a design-time model and a run-time model. A design method is formalized as a design theory; a concrete design is formalized as a design-time model; a run-time application is formalized as a run-time model. Figure 3.2 shows a portion of the design theory of OMT's object model and two models of this

theory: (a) is the ER diagram of the design theory; (b) is a concrete design of (a); (c) and (d) are two run-time instances of (b). The combination of (b) and (c) is a model of (a), so is the combination of (b) and (d). (b), (c), and (d) use OMT notations.

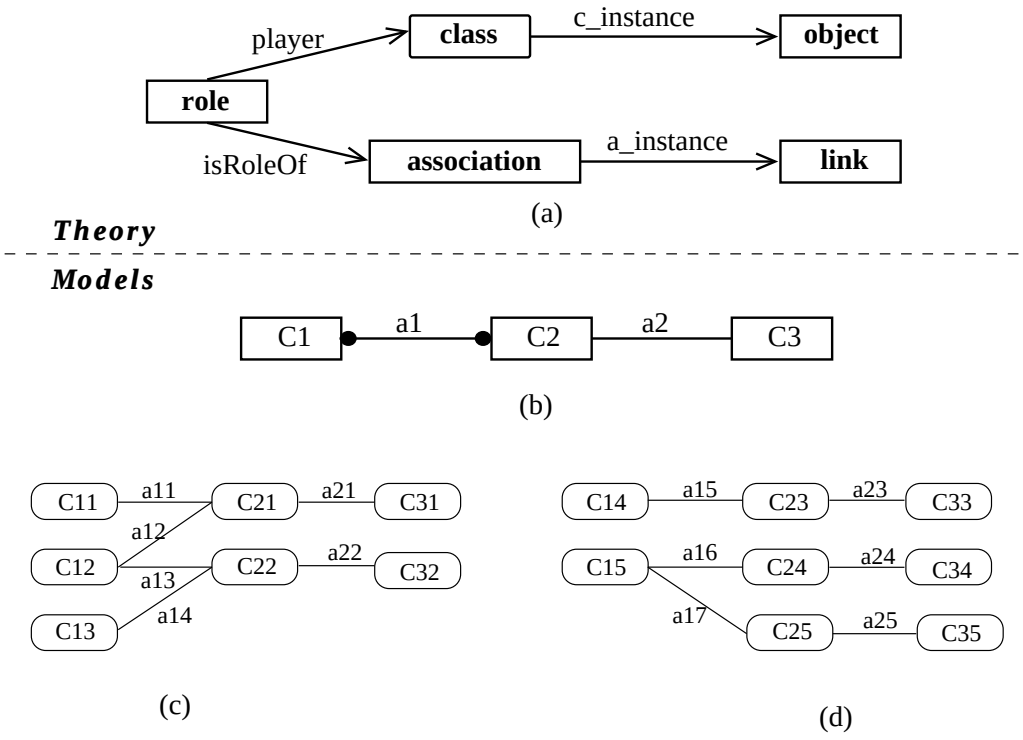


Figure 3.2: The Theory-Model formalization of a portion of OMT's object model.

3.3.2 Design Categories – Introducing Types

A design method is expressed wholly within some limited set of graphical and textual notations which are used to describe the elements of the design. Every class of the design elements is a *design category*.

During the formalization, we must develop formal representations for each

design category. The formalization language, Z, is strongly typed. For each design category, we introduce a type to correspond to it. We choose one of three ways of formalizing a category:

- Introducing a *given set*.
- Introducing a subset of some previously introduced set.
- Introducing a *schema*, the variables of which formalize essential properties of design elements in that category.

Given Sets

On recognizing a design category, the simplest way of formalization is the introduction of a new given set. As an example, in formalizing OMT, we may introduce given sets as follows.

$$[CLASS, ASSOCIATION]$$

Different members of these sets, with their unique identity, correspond to different design elements.

Combining Given Sets

Since Z is strongly typed, two different given sets have no elements or operations in common. If two design categories share similarities but are defined as given sets, strong typing would prevent the two categories from sharing assertions and operators between each other. There are three alternatives for this situation:

1. Duplicate Z text for both types. This violates the "separation of concerns" principle.
2. Withdraw the disjoint given sets, and replace them with a single given set to represent their union. We then define the assertions and operators

once only, and introduce the two originally separate categories as subsets of the union set.

In OMT, "classes" and "associations" share common features, and so we prefer to consider them together as "entities". We introduce a new given set,

$$[ENTITY]$$

and two disjoint global subsets, one each for classes and associations,

$$\begin{array}{|l} CLASS : \mathbb{P}\, ENTITY \\ ASSOCIATION : \mathbb{P}\, ENTITY \\ \hline \langle CLASS, ASSOCIATION \rangle \ \mathsf{partition}\ ENTITY \end{array}$$

This consideration is very similar to that for designing an inheritance hierarchy for an object-oriented system. Unfortunately, this reduces the amount of checking available with *f*UZZ, since it only checks types, and different subsets of the same given set have the same type.

3. Combine the two given sets in a disjoint union, which can be clumsy and hard to read. For example,

$$\begin{aligned} ENTITY ::= & \ ClassToEntity\langle\langle CLASS \rangle\rangle \\ | & \ AssociationToEntity\langle\langle ASSOCIATION \rangle\rangle \end{aligned}$$

In this way, we may be forced to introduce more variables than the second alternative as the formalization proceeds, because $ENTITY$, $CLASS$, and $ASSOCIATION$ are still three different types. This can be seen from the following two sets of code, which specify the same thing in different ways.

The variables in the following Z code

$$
\begin{array}{|l}
entity : \mathbb{P}\, ENTITY \\
class : \mathbb{P}\, CLASS \\
association : \mathbb{P}\, ASSOCIATION \\
\hline
\langle class, association \rangle \ \mathsf{partition}\ entity
\end{array}
$$

are defined based on the definition that $CLASS$ and $ASSOCIATION$ are subsets of $ENTITY$ (the second alternative). The variables defined as follows are based on the definition that $CLASS$ and $ASSOCIATION$ are different given sets (the third alternative).

$$
\begin{array}{|l}
entity : \mathbb{P}\, ENTITY \\
class : \mathbb{P}\, CLASS \\
association : \mathbb{P}\, ASSOCIATION \\
classEntity, associationEntity : \mathbb{P}\, ENTITY \\
\hline
classEntity = ClassToEntity(\!|\ class\ |\!) \\
associationEntity = AssociationToEntity(\!|\ association\ |\!) \\
entity = classEntity \cup associationEntity
\end{array}
$$

It is obvious that, to use the third alternative, we have to introduce more variables. To retain clarity and brevity, and to avoid duplicate assertions, we choose the second alternative.

Similar to "class" and "association", in OMT, "attribute", "operation", and "role" appear at first glance to be significantly different. It would seem natural to introduce three given sets. But they share some commonalities. This leads us to consider a new category "feature":

$$[FEATURE]$$

Attributes, operations, and roles will be known collectively as features:

$$
\begin{array}{|l}
ATTRIBUTE : \mathbb{P}\, FEATURE \\
OPERATION : \mathbb{P}\, FEATURE \\
ROLE : \mathbb{P}\, FEATURE \\
\hline
\langle ATTRIBUTE, OPERATION, ROLE \rangle \text{ partition } FEATURE
\end{array}
$$

Schemas for Design Categories

Schemas arise in Z specifications for collecting several pieces of related information. Formally, a Z schema means a subset of a cross product; it is an "indexed product" because the individual components of the product have individual names in Z (which are called *variables*). When elements of a design category are characterized by certain essentials, it might well be appropriate to formalize the category as a schema, and so collect the essential characteristics together as components of the product. There are, however, two phenomena that prevent or condition using a schema to formalize a design category:

- a requirement for *unique identity*, and
- a lack of *uniformity*.

Since a schema means an indexed product, instances of a schema type are distinguished only by the values of their component variables. However, design methods often allow for two elements of a design category to be present in a design with identical attribute values, or allow for an element to have different attribute values within different context. The *unique identity* of an element is not characterized by its attributes but by context. Our solution to this problem is to use a given set for the identity, and use relations to associate the design category with their essential characteristics, which may be in various forms such as given set, schema, and relation.

Since a schema means an indexed product, every attribute (variable) in the product must always be present. Optional attributes cannot be introduced as schema variables, and if a basic category lacks the *uniformity* of a small set of

required attributes, formalizing it as a schema will be difficult to understand. The way we solve this problem is to use partial functions for optional elements.

Furthermore, using schemas in defining design categories may increase complexity of formalization, because a variable in a schema may as well be a design category, which in turn may be defined as a schema. A schema is thus nested and the formalization of its relationships to other design categories becomes complicated. The way we formalize the design categories provides a simple solution.

How to formalize the relationships between design categories using relations and functions is discussed in the following section.

3.3.3 Design-Time Formalization – Introducing Variables and Predicates

Category Variables

In any software design method there are design categories; a particular design is made up of elements of those categories, related to one another according to the design rules of the method. This aspect is called the *design-time formalization* of a theory. A design category, viewed as a set of possible design elements, is usually *infinite*, because there are infinitely many *possible* designs. The elements of a particular design, viewed as a subset of possible design elements, is always *finite*, because any given design is a finite construction. We use Z types to formalize the infinite potential of design categories, and introduce Z variables to stand for their finite subsets that are the elements of particular designs. We call these variables *category variables*. They are usually set-valued. Variables *entity*, *class*, and *association* declared in Section 3.3.2 are such variables.

Relation Variables

The required relationships between design categories are captured by relations and functions in Z. Typically the relationships in a particular design apply not to *all possible* design elements of the categories, but only to those elements which have actually been introduced into the present design. We call the relations and functions to model the relationships in particular designs *relation variables*. These variables are also set-valued.

We take the formalization of relationships between entities (including classes and associations) and features (including attributes, operations, and roles) in OMT as an example to illustrate how we make decision on formalizing the relationships.

In OMT, features never exist independently, but are always associated with some entity. Given a particular feature in a valid design, we can discover the particular class or association in which the feature is introduced. In mathematical English we would say something like "for every feature, there is a corresponding point of definition in some class or association". Such a "for every ... there is a ..." suggests a formalization as a function in the mathematical sense. Here, the function

$$definedIn : FEATURE \pfun ENTITY$$

yields an $ENTITY$ element in the design (the introduction point) for any $FEATURE$ element introduced in the design. It is a partial function ($\pfun$) instead of a total one ($\fun$) because the design theory does not insist that each $FEATURE$ necessarily has a corresponding $ENTITY$. A design theory should make few claims about the set of all possible design elements, and restrict itself to the properties of the design elements in the context of particular designs.

Sometimes the relationships called for in a design theory do not suggest a functional style of formalization, but a more general relational style. However, it is better to write a relation in a functional style if possible. A functional

style leads itself to equations, and equational presentations and reasoning are clearer and more elegant.

In OMT, we need to express the meaning of "all the features of an entity". There are two ways to formalize this. One way is to define a function from entities to sets of features, that is,

$$f : ENTITY \pfun \mathbb{P}\, FEATURE$$

and "all the features of c" is $f(c)$. The other way is to define a general relation between entities and features, which happens to be the relational inverse of the function *definedIn*, i.e., *definedIn*$^\sim$; and "all the features of c" is *definedIn*$^\sim(\!|\, \{c\} \,|\!)$. We prefer the second way in the formalization because the type of the relation (which is $\mathbb{P}(ENTITY \times FEATURE)$) is simpler than the type of the function (which is $\mathbb{P}(ENTITY \times \mathbb{P}\, FEATURE)$). The complexity of one relation can make the expression of its relationships to other relations complicated.

A "minimalist" attitude in the formalization is to define a single expression of each idea. Since the purpose in specification is to make the design theory not only simple but also readable, there is a need for trade-off between minimality and readability. It is usually helpful to introduce by name all the relations and functions that will be used regularly later. If we will be writing equations in each direction of a relation, we give names to each direction. We have mentioned *definedIn* formalizing the relationship between each feature and its defined entity; we also need *hasFeature* formalizing the inverse relation:

$$\begin{array}{l} definedIn : FEATURE \pfun ENTITY \\ hasFeature : ENTITY \rel FEATURE \\ \hline definedIn \in feature \fun entity \\ hasFeature \in entity \rel feature \\ hasFeature = definedIn^\sim \end{array}$$

Predicates

The *predicates* in Z introduce restrictions over the variables. These restrictions represent the rules that the design categories must follow. It should be noted that there is no way in a theory to state that a property is not required or supported; one may only be explicit about such matters in the technical description.

If displayed in an ER diagram, category variables become entities, and relation variables become relationships. For example, in Figure 3.2 "class", "association", "role", "object", and "link" are category variables, and "player", "isRoleOf", "c_instance", and "a_instance" are relation variables.

3.3.4 Building Theories Piece by Piece – Introducing Schemas

The variables and the predicates for a design theory may all be defined globally, such as the examples given earlier in this chapter, or can be incorporated into one huge schema. In order to make a theory more comprehensible, in the formalization, we group the related categories and constraints into a set of schemas. The schemas are used as a modularity mechanism that decompose a large specification into units. Another reason for using schemas is the need for incremental comparison and the need for referencing predicates, which will be discussed in Chapter 6.

A basic principle of rhetoric is to introduce new information *evenly*. We therefore introduce new design elements and their essentials as variables in small schemas. Subordinate design properties and relationships are formalized later, by introducing new schemas focussed on the subordinate material.

For example, we use schema *OMTentity* to incorporate entities, classes, and associations. $ENTITY$ is the type of potential design elements that are classes

or associations; a given design will involve particular classes and associations, formalized as a set variable *entity* in the basic schema, having subsets *class* and *association*.

OMTentity

$$entity : \mathbb{P}\, ENTITY$$
$$class : \mathbb{P}\, CLASS$$
$$association : \mathbb{P}\, ASSOCIATION$$

$$\langle class, association \rangle \text{ partition } entity$$

Similarly, *FEATURE* is the type of potential design elements which are attributes, operations, or roles:

OMTfeature

$$feature : \mathbb{P}\, FEATURE$$
$$attribute : \mathbb{P}\, ATTRIBUTE$$
$$operation : \mathbb{P}\, OPERATION$$
$$role : \mathbb{P}\, ROLE$$

$$\langle attribute, operation, role \rangle \text{ partition } feature$$

To collect the rules about basic concepts for future reference, we define schema that combines the properties of the above two schemas.

$$OMTBasic \mathrel{\widehat=} OMTentity \land OMTfeature$$

This construct, called *schema conjunction*, essentially means that *OMTBasic* includes all the definitions and predicates of both *OMTentity* and *OMTfeature*.

We use schema *OMTFeature* to represent the Z code in Section 3.3.3.

OMTFeature

$$OMTBasic$$
$$definedIn : FEATURE \pfun ENTITY$$
$$hasFeature : ENTITY \rel FEATURE$$

$$definedIn \in feature \fun entity$$
$$hasFeature \in entity \rel feature$$
$$hasFeature = definedIn^{\sim}$$

Because of the scope rule, when schema *OMTFeature* wants to use variables *entity* and *feature* declared in schema *OMTBasic*, *OMTBasic* must be included in *OMTFeature*.

A recommended style of the technical descriptions for each schema is to provide a list of descriptions, one per predicate. Thus, for *OMTFeature*, we have

- Every *feature* has a corresponding *entity*.
- There may be more than one *feature* for each *entity*.
- *hasFeature* is the relational inverse of *definedIn*.

All the schemas about a method are regarded as the theory of the method. In fact, the theory is the combination of all the variables and predicates defined in these schemas. After establishing individual schemas, we must eventually put all these parts together to form an entire formalization. We do so in defining a new schema which incorporates the properties of all the previous schemas.

3.3.5 Formalizing Properties of Design Elements

Sets introduced for design categories have members which formalize design elements. This captures the identity of design elements. However, design

theories usually allow for additional properties of design elements, optional for their category; and impose or allow for further relationships between elements.

In OMT, a "role" of an "association" must have two data:

- the "class" whose objects can play the role in an association, and
- the "multiplicity", which governs how many objects can play the role (in a dynamic sense).

In general, this kind of relationship-determining data can be formalized in either of two ways.

1. Represent each datum separately as a function of the design category.

   Every role has an associated class and a multiplicity; thus we introduce

   OMTRoleProperty

   $$OMTBasic$$
   $$player : ROLE \pfun CLASS$$
   $$multiplicity : ROLE \rel \mathbb{N}$$

   $$player \in role \fun class$$
   $$\dom multiplicity = role$$

2. Represent the data collectively as variables of a new schema which formalizes the relationship.

   Role

   $$role : ROLE$$
   $$player : CLASS$$
   $$multiplicity : \mathbb{P}_1 \mathbb{N}$$

   Some predicates are needed to quantify over members of the *role* set; to talk about the corresponding properties, we then need a way to find the schema representing the properties of a role, given the role itself:

$$
\begin{array}{|l}
\hline
xOMTRoleProperty \\
\hline
OMTBasic \\
roleData : ROLE \pfun Role \\
\hline
\dom roleData = role \\
\{r : role \bullet (roleData(r)).player\} \subset class \\
\forall r : role \bullet (roleData(r)).role = r \\
\hline
\end{array}
$$

The second alternative seems attractive, since it gathers all the information about a type in one place. However, we choose the first one for several reasons:

- The need to use the *roleData* function complicates several of the predicates.
- Using a schema in this way requires that *every* element of the design category in question possesses the properties to be described. "Name", for example, cannot be formalized as a variable of a schema describing roles in OMT, because not all roles have names.
- Defining a schema for a design category requires that we discuss all the properties of the design category in one place. It makes it more difficult for us to distribute our discussion of properties among separate sections of the formalization.

3.3.6 Evolving and Revising a Formalization

Software design methods are typically defined in large books. It is common in such books to cover basic features first, then introduce advanced features separately. This sometimes means that decisions made earlier in the formalization process require revision later.

What we have for roles is adequate for most kinds of associations one finds in an object model in OMT. However, two of the advanced features

complicate the representation. In both aggregation and generalization, there can be several sets that play a particular role. In aggregation there is a single container class and possibly several "part of" classes. In generalization there is a single superclass and possibly several subclasses.

In the earlier stage of our thinking about this problem, it seemed necessary to introduce several different roles to represent the different parts in an aggregation, and similarly several different roles to represent the subclasses of a generalization. The formalization began to look more and more complicated as we introduced several new schemas and other elements to represent properties of aggregation and generalization.

Whenever things look like they are getting too complicated, it is wise to look over the material again to see if one can discover an appropriate simplification. For example, we considered representing all the different parts of a fixed aggregate as one role, where the role simply had several distinct players. This would have required changing the *OMTRoleProperty* schema so that the *player* function (from *ROLE* to *CLASS*) became a general relation. This introduced its own complications, but would let us talk about the "part-of" role as meaning all the parts of an aggregation. However, [RBP+91] gives examples of aggregates where, for example, a microcomputer is an aggregate of a monitor, a system box, a mouse, and a keyboard, and each of those parts of the microcomputer aggregate could potentially have different multiplicities. For example, there are one or more monitors and zero or one mouse (Figure 3.22 on p.38 of [RBP+91]). Thus, we are forced to regard an aggregate as having many different "part of" roles. This vindicates our original idea that the "part of" elements of an aggregate could each be thought of as a distinct role – but it required examination of the original material to ensure that this was an essential aspect of the formalization, and not just a convenience.

The fundamental question arises at this point: can we still view the relationships between the container and its parts in an aggregation as associations, or must we introduce a new category? If aggregates were to be completely dif-

ferent from other kinds of associations, they would require a new Z base type as their representation in the formalization. However, aggregates still seem to have roles, just as simple associations do, but require additional information and additional structure in their descriptions.

Thus, we introduce a given set to represent aggregations, and a schema to represent the basic information associated with an aggregation:

$$[AGGREGATION]$$

OMTAggregationBasic

$$OMTBasic$$
$$aggregation : \mathbb{P}\, AGGREGATION$$
$$collectedIn : ASSOCIATION \pfun AGGREGATION$$

$$collectedIn \in association \pfun aggregation$$

We define the set of aggregations in a particular design (*aggregation*), and the relationship between associations and the aggregations that collect them. An aggregation isn't just an arbitrary collection of associations. We need to express several consistency requirements. The detail is discussed in Section 4.1.5.

Regardless of the specific choice we make, incorporating new pieces of the formalization requires going back to edit old ones. We regard such revision cycles as normal.

Even in the course of comparison, we may need to go back to revise some parts. One of the purposes of formalization is to provide theoretical basis for method comparison. Roughly speaking, two design categories in two methods can be compared if they have similar properties. It will be beneficial to the comparison if these properties are formalized in a similar way. Further, when comparing similar properties of the two methods, we may find that our understanding of one method is not accurate with respect to this property. We then need to go back to revise the formalization at this point.

3.3.7 Levels of Abstraction

When formalizing a concept and its properties, we need to decide what level of abstraction the formalization should stay at.

In OMT, entities and features may have names, which may be strings. But unless we need to model naming conventions explicitly, it is unnecessary to distinguish names to this level of detail. Instead, we simply introduce a given set for names:

$$[NAME]$$

In OMT, an operation can be a "query" or an "update" operation. A query operation is an operation that computes a functional value without modifying any object; an update operation changes attribute values. OMT does not describe in further details how a query or update operation behaves, and we do not intend to improve the method; thus, we just simply formalize the two kinds of operations as subsets of operations:

OMTOperationType

$$OMTBasic$$
$$queryOp : \mathbb{P}\, OPERATION$$
$$updateOp : \mathbb{P}\, OPERATION$$

$$\langle queryOp, updateOp \rangle \text{ partition } operation$$

3.3.8 Adding Auxiliary Concepts and Missing Details

We sometimes use *auxiliary concepts* in applying the formalization to make the theory easier understood and more precise. These concepts are not part of the original, but are created under the condition that the added concepts do not alter the semantics of the method. As an example, OMT discusses different kinds of associations, such as one-to-one and many-to-many associations, in

terms of multiplicity of their roles; but it does not define different kinds of roles in terms of multiplicity. It is possible to capture this level of abstraction by directly referencing the multiplicity attribute of the *OMTRoleProperty*. However, we find that the formalization will become less complex if we give formal definition of different kinds of roles. Thus, we have

OMTRoleType

$$
\begin{array}{l}
OMTRoleProperty \\
singularRole, optionalRole, multipleRole : \mathbb{P}\, ROLE \\
\hline
\langle singularRole, optionalRole, multipleRole \rangle \ \mathsf{partition}\ role \\
\forall\, s : singularRole \bullet multiplicity(\!|\ \{s\}\ |\!) = \{1\} \\
\forall\, o : optionalRole \bullet multiplicity(\!|\ \{o\}\ |\!) = \{0, 1\} \\
\forall\, m : multipleRole \bullet (\exists\, n : multiplicityOfRole(\!|\ \{m\}\ |\!) \bullet n > 1)
\end{array}
$$

Another example of auxiliary concepts is the use of "entity" as already discussed. By introducing this concept, we can specify properties common to both *class* and *association* in one set of predicates that should otherwise be two sets; it thus avoids the duplicates.

Some categories are not explicitly discussed in the methods; some are only mentioned in examples. During the formalization, we need to make implicit concepts in the method explicit. For example, OMT only mentions one-to-many, optional-to-many, and many-to-many binary associations. In fact, there are other possible combinations, such as one-to-one, optional-to-one, and so on. In our formalization, we make these missing details explicit.

3.4 Verification of Theories

Verification of a design theory is to use various techniques (including mathematical analysis techniques, but not limited to them) for gaining confidence

in the correctness of the theory. The verification task includes type-checking, validation, consistency checking, and completeness checking.

Type-checking is to check syntactic inconsistency. Validation is to show that certain properties are logical consequences of the theory. Consistency checking is to prove that no conflicting properties can be deduced from the theory. Completeness checking is to ensure that all the properties of the method are captured in the theory. Consistency and completeness of a theory are interrelated: often inconsistency leads to incompleteness and vice versa. If an error is found during the verification, it implies that either the theory or the corresponding design method is incomplete or inconsistent. This is a way to improve the theory and gain deep insight into the method.

Type-checking is accomplished with the aid of the *f*UZZ type checker. This is an automatic process. Besides the automatic type-checking, there are basically three ways to verify a theory: inspection, reasoning, and execution.

Inspection

Finding defects by inspection should be more efficient than finding them by simulation or other testing. We do the inspection manually by examining the theory against the informal descriptions in the publications about the method. Comparing one theory with another can also help improve the accuracy of the theories. For example, in schema *OMIsKindOf* (on p.78) which describes the properties of inheritance in OMT, we initially did not include predicate $isDescendentOf \cap \mathrm{id}\, class = \emptyset$. However, in schema *IMInheritanceBasic* (on p.137) of SMOOA, there is a predicate similar to this one, i.e., $inherit \cap \mathrm{id}\, object = \emptyset$. When we compared the two theories, we found that we missed the predicate in the theory for OMT's object model. There are a few other similar cases.

Reasoning

A rigorous way of verification is via theorem proving or model checking. This is a research topic related to formal reasoning [ABC82, Boe84, VW86, MP91, BMZ91, Hun93, AK94, LvK94, Gra96], and is beyond the scope of our research. In the absence of automated tool support, rigorous proofs are extremely laborious. We performed some reasoning manually to examine whether certain properties of a method can be deduced from its theory. Z/EVES can be used to prove theorems about Z specifications; however, it became available too late for us to use during the thesis work.

Execution

An alternative and simple way to check consistency is to establish a concrete model of the theory. According to the completeness theorem of model theory [CK73], if a theory is inconsistent, it is impossible to find a model of it. In this way, we can also validate and debug a design theory. This requires us to make the formal description executable, that is to develop an executable representation of the theory.

We have developed a reasonable straightforward method for translating our style of formal description into Prolog [LMZ96]. Since, in general, the sets and design rules we are dealing with are all finite, we know that it is theoretically possible to test whether a concrete design is a model of the corresponding theory, by searching for contradictions to the predicates. We hope it will also prove to be feasible. The translation method so far developed falls into four main steps:

- Represent the entity sets.
- Represent the relationship sets.
- Represent basic integrity constraints, such as type and multiplicity constraints.

- Represent other axioms of the theory.

Each member x of a given set M is represented by a Prolog ground fact, of the form $M(x)$, asserting that its single argument is an element of the set, and defining x's identity. There are several possibilities for representing entities:

- A simple method is to pick Prolog atoms formed by appending a number to a base name. Thus, for example, roles might be represented as

  ```
  role(role_001).
  role(role_002).
  ```

- If a collection of given sets forms a single generalization hierarchy, we might consider using the representation of the parent of the hierarchy for the children as well. Thus, for example, classes and associations are both entities:

  ```
  class(entity_001).
  class(entity_002).
  association(entity_003).
  ```

  With this approach, it would be natural to represent the parent by a rule rather than by enumeration:

  ```
  entity(X) :- class(X).
  entity(X) :- association(X).
  ```

N-ary relationship sets can be represented by n-ary Prolog predicates. Thus, we might represent part of the information about an "employee works-in department" relationship as:

```
nameOfEntity(entity_001,"employee").
nameOfEntity(entity_002,"department").
```

```
nameOfEntity(entity_003,"works in").
player(role_001,entity_001).
isRoleOf(role_001,entity_003).
```

Integrity constraints are straightforward. In general, constraints are verified by searching for a counter-example; on failure, the constraint is considered to be satisfied.

- Type constraints on relationships can be checked by rules that search for elements of the wrong type:

```
badNameOfEntityFirstArg(X) :-
  nameOfEntity(X,_),
  not(entity(X)).
```

- Cardinality constraints on functions and relations can be checked by stylized rules that build sets and check their sizes.

More complex axioms may require hand-translation.

Running Prolog program on small examples serves to increase our confidence in the correctness of the formalization.

Chapter 4

Formalization of OMT's Object Model

This chapter describes a formalization of the object model of OMT [RBP+91]. The product of the formalization is the design theory of the object model described in Z notation.

OMT uses three models to capture three related but different aspects of a system. These models are *object model*, *dynamic model*, and *functional model*. They separate a system into orthogonal views that can be represented and manipulated with a uniform notation. The different models are not completely independent – the interconnections between them are limited and explicit – but each model can be examined and understood by itself to a large extent.

Object model. The object model describes the static structure of objects in a system – their identity, their relationships to other objects, their attributes, and their operations that characterize each class of objects.

Dynamic model. The dynamic model describes those aspects of a system concerned with time and the sequencing of operations – events that mark changes, sequences of events, states that define

the context for events, and the organization of events and states.

Functional model. The functional model describes those aspects of a system concerned with transformations of values – functions, mappings, constraints, and functional dependencies.

The object model provides the essential framework into which the dynamic and functional models can be placed.

4.1 Basic Design-Time Concepts

Design-time concepts describe the possible patterns of run-time applications. These concepts in an object model include class, association, attribute, operation, method, role, aggregation, generalization, and module. This section gives the formal definitions of the design-time concepts and formalizes some design rules on these concepts. The original material on which the following discussion is based can be found in Chapter 3 of [RBP+91].

4.1.1 Fundamental Concepts

Class and *association* are the central concepts in an object model. A *class* describes a group of objects[1] with similar properties (attributes), common behaviors (operations), common relationships to other objects, and common semantics. An *association* describes a group of links (of objects) with common structure and common semantics. We introduce a given set,

$$[ENTITY]$$

and two global sets, one each for classes and associations,

$$CLASS : \mathbb{P}\, ENTITY$$
$$ASSOCIATION : \mathbb{P}\, ENTITY$$

[1] Object, link, attribute value, and other run-time concepts are discussed in Section 4.3.

As discussed in Chapter 3, the reason we formalize both classes and associations to be elements of the same given set is that they have properties in common. Both a class and an association may have attributes.

An association may be modeled as a class (see [RBP+91]: p.33). There are two interpretations for this. One interpretation is formally represented by the above two subset definitions. In this case, the intersection of "classes" and "associations" might be an non-empty set, meaning that an association may also be a class and vice versa. The other interpretation is formally represented by the two subset definitions as well as a predicate

$$\langle CLASS, ASSOCIATION \rangle \text{ partition } ENTITY$$

indicating that a class and an association are distinct; however, the properties common to both are attached to entities, of which classes and associations are subsets. From the example given on page 34 of [RBP+91], we find that, while an association is modeled as a class, the association and the association class may have different names. This indicates that the two concepts should be considered distinct. Therefore, we adopt the second interpretation.

Attribute, *operation*, and *role* are basic concepts with some properties in common. An *attribute* is a data value that may be held by the objects in a class. An *operation* is a function or transformation that may be applied to, or by, objects in a class. A *role* is an end of an association. Each role "starts" from an association and "ends" in a class. The three concepts have the following common aspects:

- They do not exist independently, but rather are always associated with some class or association.
- They may have names, although roles are not necessary named. Their items associated with an entity should have different names from each other.

- In generalization relationships, which are formalized in Section 4.1.5, subclasses can inherit attributes and operations from superclasses.

We introduce a given set,

$$[FEATURE]$$

and three disjoint subsets,

$$\begin{array}{|l} ATTRIBUTE : \mathbb{P}\, FEATURE \\ OPERATION : \mathbb{P}\, FEATURE \\ ROLE : \mathbb{P}\, FEATURE \\ \hline \langle ATTRIBUTE, OPERATION, ROLE \rangle \text{ partition } FEATURE \end{array}$$

for attributes, operations, and roles respectively.

The global given sets and subsets represent the design categories; they are the sets of all possible elements of all possible object models. We will need to define the sets that represent a particular object model. An object model consists of

- a set of entities,
- a set of classes,
- a set of associations,
- a set of features,
- a set of attributes,
- a set of operations, and
- a set of roles.

$$
\begin{array}{|l}
\hline
OMDTData \\
\hline
entity : \mathbb{P}\, ENTITY \\
class : \mathbb{P}\, CLASS \\
association : \mathbb{P}\, ASSOCIATION \\
feature : \mathbb{P}\, FEATURE \\
attribute : \mathbb{P}\, ATTRIBUTE \\
operation : \mathbb{P}\, OPERATION \\
role : \mathbb{P}\, ROLE \\
\hline
class \neq \emptyset \\
\langle class, association \rangle \ \mathsf{partition}\ entity \\
\langle attribute, operation, role \rangle \ \mathsf{partition}\ feature \\
\hline
\end{array}
$$

- The object model must contain some classes.
- Classes and associations are subsets of the entity set.
- Attributes, operations, and roles are subsets of the feature set.

4.1.2 Features of Entities

A basic property of attributes, operations, and roles is that they are always associated with some particular class or association. Thus, given a "feature", one can discover the particular "entity" in which it is first defined. This section formalizes some basic properties of the features as well as the relationships between the features and the entities.

Attributes

The value of an attribute belongs to certain value type. We introduce given sets

$$[ATOMIC\_VALUE, ATT\_TYPE]$$

for all the possible atomic values and value types.

Each attribute type defines a set of values:

$$valueOfType : ATT\_TYPE \leftrightarrow ATOMIC\_VALUE$$

$$valueOfType^{\sim} \in ATOMIC\_VALUE \twoheadrightarrow ATT\_TYPE$$

Schema *OMAttributeProperty* defines a function specifying the value type of which the values of each attribute are.

OMAttributeProperty

$$OMDTData$$

$$typeOfAttr : ATTRIBUTE \twoheadrightarrow ATT\_TYPE$$

$$\mathrm{dom}\, typeOfAttr = attribute$$

- Each attribute has a defined value type.

Schema *OMAttribute* defines a relation indicating the entity in which each attribute is originally defined.

OMAttribute

$$OMDTData$$

$$attrDefinedIn : ATTRIBUTE \leftrightarrow ENTITY$$

$$\mathrm{dom}\, attrDefinedIn = attribute$$

$$\mathrm{ran}\, attrDefinedIn \subseteq entity$$

- Each attribute must be defined in some class(es) or association(s) (see [RBP+91]: pp.31-33 about link attribute).

- A class or an association may have one or more attributes.

Operations

Every operation has a predefined signature. The *signature* of an operation is the number and types of arguments and the type of the result value if any. Each operation has a target object as an implicit argument; but there is no indication of the forms that other arguments can take (see [RBP+91]: p.25). At this level of detail, we define a given set

$$[ARG\_TYPE]$$

for the types of arguments and results.

Schema *OMOperationProperty* denotes

- a function specifying the types of the arguments of each operation, and
- a function specifying the type of the result of each operation.

OMOperationProperty

$$OMDTData$$
$$argumentOfOp : OPERATION \pfun \mathrm{seq}_1\ ARG\_TYPE$$
$$resultOfOp : OPERATION \pfun \mathrm{seq}\ ARG\_TYPE$$

$$\mathrm{dom}\ argumentOfOp = operation$$
$$\mathrm{dom}\ resultOfOp \subseteq operation$$

- Each operation takes some arguments. An operation has at least one argument, since it always has a target object as an argument.
- An operation may return some result.

Schema *OMOperation* defines a relation indicating the entity in which each operation is originally defined.

OMOperation

$OMDTData$

$opDefinedIn : OPERATION \leftrightarrow CLASS$

$\text{dom}\ opDefinedIn = operation$

$\text{ran}\ opDefinedIn \subseteq class$

- Each operation must be applied to, or by, one or more classes.
- A class may have one or more operations on it.

An operation can be a *query* or an *update* operation (see [RBP+91]: p.26, p.64). A query operation is an operation that computes a functional value without modifying any object. An update operation changes attribute values.

Schema *OMOperationType*1 denotes that query operations and update operations in the object model partition the operation set.

*OMOperationType*1

$OMDTData$

$queryOp : \mathbb{P}\ OPERATION$

$updateOp : \mathbb{P}\ OPERATION$

$\langle queryOp, updateOp \rangle$ partition $operation$

Roles

The formalization of the relationships between classes and associations are based on the concept of roles. Although attributes, operations, and roles are all defined as features, roles have some special properties. This is because attributes and operations are merely attached to classes, while roles are related to both classes and associations.

Schema *OMRoleProperty* denotes

- a function specifying the class that plays each role, and
- a relation specifying the multiplicity of each role. The *multiplicity* of a role at run-time specifies the number of instances of one class that appear in a given role of an association when the instances that can appear in the remaining roles are fixed. Multiplicity is a (possibly infinite) subset of the non-negative integers. For the role r of an association, each number n in the set $multiplicityOfRole(\!| \{r\} |\!)$ indicates that the multiplicity of r can be n. Section 4.3.5 gives the semantics of multiplicity in terms of run-time model.

OMRoleProperty

$$OMDTData$$
$$playerOfRole : ROLE \pfun CLASS$$
$$multiplicityOfRole : ROLE \rel \mathbb{N}$$

$$playerOfRole \in role \fun class$$
$$\mathrm{dom}\, multiplicityOfRole = role$$

- Each role has a player, which is a class.
- Each role has a predefined multiplicity.

Schema *OMRole* defines two relations indicating the association in which each role is involved.

OMRole

$OMDTData$

$roleDefinedIn : ROLE \pfun ASSOCIATION$

$hasRole : ASSOCIATION \rel ROLE$

$roleDefinedIn \in role \fpfun association$

$hasRole = roleDefinedIn^{\sim}$

$\forall a : association \bullet \#(hasRole(\!| \{a\} |\!)) \geq 2$

- Each role must be associated with one and only one association.
- Each association must have roles associated with it.
- An association may have two or more roles, i.e., an association may be binary, ternary, or higher order.

In discussing associations, it is common that there is need to refer to roles that are optional, singlular or multiple. Thus, for example, a many-to-many association has two multiple roles whereas a one-to-one association has either two singleton roles or two optional roles or one of each. It is possible to capture this level of abstraction by referencing the multiplicity attribute of the *OMRoleProperty* schema. But it is more convenient to be able to directly refer to a property of role as to whether it is in one of the above mentioned set. Thus, we extend the role description schema as follows.

OMRoleType

$$OMRoleProperty$$

$$singularRole, optionalRole, multipleRole : \mathbb{P}\, ROLE$$

$$\langle singularRole, optionalRole, multipleRole \rangle \text{ partition } role$$

$$\forall\, s : singularRole \bullet multiplicityOfRole(\!|\, \{s\} \,|\!) = \{1\}$$

$$\forall\, o : optionalRole \bullet multiplicityOfRole(\!|\, \{o\} \,|\!) = \{0, 1\}$$

$$\forall\, m : multipleRole \bullet (\exists\, n : multiplicityOfRole(\!|\, \{m\} \,|\!) \bullet n > 1)$$

- Singular, optional, and multiple roles are special types of roles with respect to multiplicity.
- Given a role of an association, when the instances of other roles in the same association are fixed in a run-time model,
  - if the role is singular, then there is exactly one object playing the role;
  - if the role is optional, then there is one object, or none at all, playing the role;
  - if the role is multiple, then there may be zero, one or more objects playing the role. And there must be the cases, in which two or more objects play the role.

Features

Putting schemas *OMAttribute*, *OMOperation*, and *OMRole* together, we introduce the following schema to express the relations between features and entities in the object model.

OMFeatureBasic

$OMAttribute$

$OMOperation$

$OMRole$

$definedIn : FEATURE \leftrightarrow ENTITY$

$hasFeature : ENTITY \leftrightarrow FEATURE$

$definedIn = attrDefinedIn \cup opDefinedIn \cup roleDefinedIn$

$hasFeature \in entity \leftrightarrow feature$

$definedIn^{\sim} \subseteq hasFeature$

- Function *definedIn* relates each feature to the entity in which it is first defined.
- Function *hasFeature* has different semantics from that of relation $definedIn^{\sim}$. Because of inheritance, an attribute or an operation of a class may not be defined in the class, but rather inherited from the ancestors of the class. Function *hasFeature* specifies the features of each entity, disregarding whether they are first defined or inherited.

Default Attribute Values

An attribute may have a predefined default value. The following schema defines a function which associates the attributes in an entity with their default values.

OMDefaultValueBasic

$$
\begin{array}{l}
OMAttributeProperty \\
OMFeatureBasic \\
defaultValOfAttr : ENTITY \pfun (ATTRIBUTE \pfun ATOMIC\_VALUE) \\
\hline
\dom defaultValOfAttr \subseteq entity \\
\forall\, e : \dom defaultValOfAttr \bullet \dom(defaultValOfAttr(e)) \subseteq hasFeature(\!| \{e\} |\!) \\
\forall\, e : entity;\ a : attribute;\ v : ATOMIC\_VALUE \mid \\
\qquad v = defaultValOfAttr(e)(a) \bullet v \in valueOfType(\!| \{typeOfAttr(a)\} |\!)
\end{array}
$$

- The first two predicates state that the attributes of an entity may have predefined default values.
- The default value of an attribute must be of the value type of the attribute.

Methods

A *method* is an implementation of an operation for a class. We introduce a given set

$$[METHOD]$$

for all of possible methods.

Similar to schema *OMOperationProperty*, schema *OMMethodProperty* denotes

- a set of methods,
- a function specifying the types of the arguments of each method, and
- a function specifying the type of the result of each method.

OMMethodProperty

$$
\begin{array}{l}
OMDTData \\
method : \mathbb{P}\, METHOD \\
argumentOfMethod : METHOD \pfun \mathrm{seq}_1\, ARG\_TYPE \\
resultOfMethod : METHOD \pfun \mathrm{seq}\, ARG\_TYPE \\
\hline
\mathrm{dom}\, argumentOfMethod = method \\
\mathrm{dom}\, resultOfMethod \subseteq method
\end{array}
$$

Schema *OMMethod* formalizes the relationship between operations and their methods. It denotes

- a function associating each operation with its implementations, and
- a function indicating the implementation of each operation in different classes.

OMMethod

$$
\begin{array}{l}
OMOperationProperty \\
OMMethodProperty \\
OMFeatureBasic \\
implement : METHOD \pfun OPERATION \\
methodLookup : OPERATION \pfun (CLASS \pfun METHOD) \\
\hline
implement \in method \fun operation \\
\forall\, op : operation;\ m : method \mid implement(m) = op \bullet \\
\qquad argumentOfOp(op) = argumentOfMethod(m) \land \\
\qquad resultOfOp(op) = resultOfMethod(m) \\
\mathrm{dom}\, methodLookup = \mathrm{ran}\, implement \\
\forall\, op : \mathrm{dom}\, methodLookup \bullet \\
\quad methodLookup(op) \in hasFeature^{\sim}(\!|\, \{op\} \,|\!) \pfun implement^{\sim}(\!|\, \{op\} \,|\!)
\end{array}
$$

- Each method is an implementation of some operation.
- A method has the same signature as the operation which it implements.
- The last two predicates indicate that, when applied to more than one class, an operation may be implemented in same or different way in different classes.

Schema *OMFeature* is a combination of the schemas that specify the properties of features.

$$
\begin{aligned}
&OMFeature \mathrel{\widehat=} \\
&\quad OMFeatureBasic \land \\
&\quad OMAttributeProperty \land OMOperationProperty \land OMRoleProperty \land \\
&\quad OMOperationType1 \land OMRoleType \land \\
&\quad OMDefaultValueBasic \land OMMethod
\end{aligned}
$$

4.1.3 Names and Naming

In order to formalize the names that various elements may have, we introduce a given set

$$[NAME]$$

for all possible names.

The following three schemas formalize the naming of entities.

Schema *OMClassName* defines

- a set of class names in the object model, and
- a function from classes to class names, specifying the name of each class.

$$
\begin{array}{l}
\textbf{\textit{OMClassName}} \\
\hline
OMDTData \\
className : \mathbb{P}\, NAME \\
nameOfClass : CLASS \rightarrowtail\!\!\!\!\!+\;\, NAME \\
\hline
nameOfClass \in class \rightarrowtail className \\
\hline
\end{array}
$$

- Each class has a unique name.

Schema *OMAssociationName* defines

- a set of association names in the object model, and
- a function from associations to association names, specifying the name of each association.

$$
\begin{array}{l}
\textbf{\textit{OMAssociationName}} \\
\hline
OMDTData \\
assocName : \mathbb{P}\, NAME \\
nameOfAssoc : ASSOCIATION \mathrel{+\!\!\!\twoheadrightarrow} NAME \\
\hline
nameOfAssoc \in association \twoheadrightarrow assocName \\
\hline
\end{array}
$$

- In OMT, an association may be left unnamed (see [RBP$^+$91]: p.28). If associations have names, some of them may have the same name.

Schema *OMEntityName* specifies that a class and an association should not have the same name.

$$
\begin{array}{l}
\textbf{\textit{OMEntityName}} \\
\hline
OMClassName \\
OMAssociationName \\
\hline
className \cap assocName = \emptyset \\
\hline
\end{array}
$$

The naming of features is slightly more complicated, because the scope of feature names is not global. The scope for features is the entity with which they are associated. When formalizing the naming in the local scope, we parameterize the sets and functions by the scope. The following two schemas formalize the naming of the features.

Schema *OMFeatureNameBasic* defines, within a given entity,

- a set of feature names,
- a set of named features, and
- a function from features to names, specifying the name of a feature.

OMFeatureNameBasic

$$
\begin{array}{l}
OMFeature \\
featureName : ENTITY \pfun \mathbb{P}\, NAME \\
namedFeature : ENTITY \pfun \mathbb{P}\, FEATURE \\
nameOfFeature : ENTITY \pfun (FEATURE \pinj NAME) \\
\hline
\mathrm{dom}\, featureName = \mathrm{dom}\, namedFeature = \mathrm{dom}\, nameOfFeature \\
\mathrm{dom}\, featureName \subseteq \mathrm{dom}\, hasFeature \\
\forall\, e : \mathrm{dom}\, namedFeature \bullet \\
\quad namedFeature(e) \subseteq hasFeature(\!|\, \{e\} \,|\!) \land \\
\quad nameOfFeature(e) \in namedFeature(e) \pfun\!\!\!\rightarrow featureName(e)
\end{array}
$$

Given an entity,

- features of the entity may have names;
- if features have names, they have distinct names.

In OMT, both attributes and operations must be named; the use of role name is optional (see [RBP+91]: pp.34-35).

OMFeatureName

$$OMFeatureNameBasic$$

$$\forall e : entity;\ f : attribute \cup operation \mid f \in hasFeature(\!|\ \{e\}\ |\!) \bullet e \in \mathrm{dom}\, namedFeature \land f \in namedFeature(e)$$

The following schema combines the properties of the previous name-related schemas.

$$OMName \mathrel{\widehat=} OMEntityName \land OMFeatureName$$

4.1.4 More Concepts Concerning Associations and Roles

This section discusses four concepts related to special types of associations and roles. They are: ordered role, association class, binary association, and qualified association.

Ordering

Usually the objects which play a role on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes, however, the objects are explicitly ordered (see [RBP+91]: p.35). Such ordered roles in the object model are specified in the following schema:

OMOrderedRole

$$OMFeature$$
$$orderedRole : \mathbb{P}\, ROLE$$

$$orderedRole \subseteq multipleRole$$

- The ordered roles are a subset of the "many" roles.

The semantics of ordering implies the existence of a run-time function which, given a role, returns a sequence of objects in some order. This semantics is formalized in Section 4.3.5.

Association Classes

An association can be modeled as a class. In such a case, the attributes of the association become the attributes of the class, and the links of the association become the instance of the class. The latter is formalized in Section 4.3.4.

The following schema defines

- a set of association classes, and
- a function that maps the associations to the association classes.

$$\textit{OMAssociationClass}$$

$$\textit{OMFeature}$$

$$\textit{associationClass} : \mathbb{P}\ \textit{CLASS}$$

$$\textit{modeledAs} : \textit{ASSOCIATION} \rightarrowtail\!\!\!\!\!\!\!\!+\;\; \textit{CLASS}$$

$$\textit{associationClass} \subset \textit{class}$$

$$\textit{modeledAs} \in \textit{association} \rightarrowtail\!\!\!\!\!\!\!\!+\;\; \textit{associationClass}$$

$$\operatorname{ran} \textit{modeledAs} = \textit{associationClass}$$

$$\operatorname{dom} \textit{modeledAs} \cap \operatorname{ran} \textit{attrDefinedIn} = \emptyset$$

- Association classes are a subset of classes.
- An association may be modeled as a class.
- Each association class models an association.
- If an association is modeled as a class, the association itself no longer possesses attributes; instead, the attributes of the association become the attributes of the class.

Binary Associations

In practice, the majority of the associations are binary. The binary associations in the object model are formalized as follows.

OMBinaryAssociationBasic

$$\begin{array}{l} OMFeature \\ binaryAssociation : \mathbb{P}\, ASSOCIATION \\ \hline binaryAssociation \subseteq association \\ \forall\, a : binaryAssociation \bullet \#(hasRole(\!|\, \{a\}\, |\!)) = 2 \end{array}$$

- The binary associations are a subset of the associations.
- A binary association has exactly two roles.

OMT puts certain constraints on the names of the roles in binary associations (see [RBP+91]: p.35).

$$
\begin{array}{l}
\textit{OMBinaryAssociationRoleName} \\
\hline
\textit{OMName} \\
\textit{OMBinaryAssociationBasic} \\
\hline
\forall c : class;\ rs : \mathbb{P}\, role;\ bs : \mathbb{P}\, binaryAssociation \mid \\
\quad rs = playerOfRole^{\sim}(\!|\ \{c\}\ |\!) \land \\
\quad bs = roleDefinedIn(\!|\ rs\ |\!) \cap binaryAssociation \bullet \\
\quad\quad (\exists f : ROLE \rightarrowtail\!\!\!\!\!\!\rightarrow NAME \bullet \\
\quad\quad\quad \{b : bs;\ r : rs;\ r' : role;\ n : NAME \mid \\
\quad\quad\quad\quad hasRole(\!|\ \{b\}\ |\!) = \{r, r'\} \land \\
\quad\quad\quad\quad n = nameOfFeature(b)(r') \bullet r' \mapsto n\} \subseteq f) \\
\\
\forall ba : binaryAssociation;\ r1, r2 : role;\ c1, c2 : class \mid \\
\quad hasRole(\!|\ \{ba\}\ |\!) = \{r1, r2\} \land \\
\quad playerOfRole(r1) = c1 \land \\
\quad playerOfRole(r2) = c2 \bullet \\
\quad\quad nameOfFeature(ba)(r1) \notin \\
\quad\quad\quad (nameOfFeature(c2))(\!|\ hasFeature(\!|\ \{c2\}\ |\!) \cap attribute\ |\!) \land \\
\quad\quad nameOfFeature(ba)(r2) \notin \\
\quad\quad\quad (nameOfFeature(c1))(\!|\ hasFeature(\!|\ \{c1\}\ |\!) \cap attribute\ |\!)
\end{array}
$$

- All role names on the far end of binary associations attached to a class must be unique.
- In a binary association, no role name should be the same as an attribute name of the player of the other role.

According to the multiplicity of the two roles, binary associations can be classified as one-to-one, optional-to-one, one-to-many, optional-to-many, many-to-many, and so on. Among the various binary associations, one-to-many, optional-to-many, and many-to-many are further discussed in the context of *keys* (see Section 4.2.8). We define these binary associations as subsets

of *binaryAssociation* with specific properties.

OMBinaryAssociationType

$$
\begin{array}{l}
OMBinaryAssociationBasic \\
one\_oneAssoc, one\_mulAssoc, mul\_mulAssoc : \mathbb{P}\, ASSOCIATION \\
one\_optAssoc, opt\_mulAssoc, opt\_optAssoc : \mathbb{P}\, ASSOCIATION \\
\hline
\langle one\_oneAssoc, one\_mulAssoc, mul\_mulAssoc, one\_optAssoc, \\
\quad opt\_mulAssoc, opt\_optAssoc\rangle \ \mathsf{partition}\ binaryAssociation \\
\forall\, a : one\_oneAssoc \bullet \\
\quad (\exists\, r1, r2 : singularRole \bullet hasRole(\!|\, \{a\} \,|\!) = \{r1, r2\}) \\
\forall\, a : one\_mulAssoc \bullet \\
\quad (\exists\, r1 : singularRole;\ r2 : multipleRole \bullet hasRole(\!|\, \{a\} \,|\!) = \{r1, r2\}) \\
\forall\, a : mul\_mulAssoc \bullet \\
\quad (\exists\, r1, r2 : multipleRole \bullet hasRole(\!|\, \{a\} \,|\!) = \{r1, r2\}) \\
\forall\, a : one\_optAssoc \bullet \\
\quad (\exists\, r1 : singularRole;\ r2 : optionalRole \bullet hasRole(\!|\, \{a\} \,|\!) = \{r1, r2\}) \\
\forall\, a : opt\_mulAssoc \bullet \\
\quad (\exists\, r1 : optionalRole;\ r2 : multipleRole \bullet hasRole(\!|\, \{a\} \,|\!) = \{r1, r2\}) \\
\forall\, a : opt\_optAssoc \bullet \\
\quad (\exists\, r1, r2 : optionalRole \bullet hasRole(\!|\, \{a\} \,|\!) = \{r1, r2\})
\end{array}
$$

- One-to-one, one-to-many, many-to-many, one-to-optional, optional-to-many, and optional-to-optional associations are binary associations.
- A one-to-one association has two singular roles.
- A one-to-many association has one singular role and one multiple role.
- A many-to-many association has two multiple roles.

- A one-to-optional association has one singular role and one optional role.
- An optional-to-many association has one optional role and one multiple role.
- An optional-to-optional association has two optional roles.

Qualification

A *qualified association* relates a qualifier and two classes, which play two roles. A *qualifier* is a special attribute that reduces the effective multiplicity of an association.

Schema $OMQualifiedAssociation$ denotes

- a set of qualified associations in the object model, and
- a function relating each qualified association to its qualifier and the qualified role to which the qualifier is attached.

$$
\begin{array}{|l}
\hline
OMQualifiedAssociation \\
\hline
OMRoleProperty \\
OMBinaryAssociationBasic \\
qualifiedAssociation : \mathbb{P}\, ASSOCIATION \\
qualify : ASSOCIATION \rightarrowtail ATTRIBUTE \times ROLE \\
\hline
qualifiedAssociation \subseteq binaryAssociation \\
\\
\mathrm{dom}\, qualify = qualifiedAssociation \\
\\
\forall\, q : qualifiedAssociation;\ r, r' : role;\ a : attribute \mid \\
\quad hasRole(\!|\, \{q\} \,|\!) = \{r, r'\} \land qualify(q) = (a, r) \bullet \\
\quad\quad r \in multipleRole \land a \mapsto playerOfRole(r') \in attrDefinedIn \\
\hline
\end{array}
$$

- The qualified associations are binary associations.

- Only qualified associations have qualifiers and qualified roles.
- For each qualified association, the qualified role is on a "many" side of the association; the qualifier is an attribute of the player of the other role in the qualified association.

To collect properties about various binary associations, we have

$$
\begin{aligned}
&\mathit{OMBinaryAssociation} \mathrel{\widehat{=}} \\
&\quad \mathit{OMBinaryAssociationBasic} \wedge \\
&\quad \mathit{OMBinaryAssociationRoleName} \wedge \\
&\quad \mathit{OMBinaryAssociationType} \wedge \\
&\quad \mathit{OMQualifiedAssociation}
\end{aligned}
$$

Putting the schemas for special associations and roles together, we obtain the following schema:

$$
\begin{aligned}
&\mathit{OMAssociation} \mathrel{\widehat{=}} \\
&\quad \mathit{OMOrderedRole} \wedge \\
&\quad \mathit{OMAssociationClass} \wedge \\
&\quad \mathit{OMBinaryAssociation}
\end{aligned}
$$

4.1.5 Aggregations and Generalizations

Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the *components* of something are associated with an object representing the entire *assembly*. *Generalization* is the relationship between a class and one or more refined versions of the class. In a generalization relationship, one class is called superclass, and its refined versions are called subclasses. The superclass is considered to be a generalization of its subclasses, while the subclasses might be viewed as specializations of their superclass.

Grouping

Aggregation and generalization have similar structure. They both can be viewed as a grouping of a set of classes, with a single class at one side, and several classes at the other side. Let

$$[GROUPING]$$

represent such grouping relationships among classes.

We define aggregation and generalization to be two kinds of the grouping relationships. Let two disjoint subsets of set $GROUPING$

$$\begin{array}{|l} AGGREGATION : \mathbb{P}\, GROUPING \\ GENERALIZATION : \mathbb{P}\, GROUPING \\ \hline \langle AGGREGATION, GENERALIZATION \rangle \text{ partition } GROUPING \end{array}$$

represent all possible aggregations and generalizations.

Schema $OMGroupingBasic$ denotes

- a set of grouping relationships in the object model,
- a set of aggregations in the object model, and
- a set of generalizations in the object model.

$$\begin{array}{|l} \textit{OMGroupingBasic} \\ \hline OMDTData \\ grouping : \mathbb{P}\, GROUPING \\ aggregation : \mathbb{P}\, AGGREGATION \\ generalization : \mathbb{P}\, GENERALIZATION \\ \hline \langle aggregation, generalization \rangle \text{ partition } grouping \end{array}$$

- Aggregation set and generalization set are subsets of grouping set.

Schema *OMGroupedClass* denotes

- a function, mapping each grouping relationship to a single class, which is an assembly class if the relationship is an aggregation, or is a superclass if the relationship is a generalization, and
- a relation, mapping each grouping relationship to a group of classes, which are component classes or subclasses.

OMGroupedClass

$$
\begin{array}{l}
OMGroupingBasic \\
single : GROUPING \pfun CLASS \\
group : GROUPING \rel CLASS \\
\hline
single \in grouping \fun class \\
\\
\dom group = grouping \land \ran group \subset class
\end{array}
$$

- Each aggregation has an assembly class; each generalization has a superclass.
- Each aggregation has several component classes; each generalization has several subclasses.

OMT views an aggregation as a set of binary associations, each of which associates an assembly with one of its components. Similarly, a generalization represents relationships between a superclass and each of its individual subclasses. Therefore, we treat a generalization as a set of binary associations, each of which associates a superclass with one of its subclasses.

Schema *OMGroupedAssociation* denotes

- a set of grouped associations that make up the aggregations and the generalizations,

- a set of aggregation associations,
- a set of generalization associations,
- a function specifying the parent roles played by assemblies in the aggregation associations or by superclasses in the generalization associations, and
- a function specifying the child roles played by components in the aggregation associations or by subclasses in the generalization associations.

$$
\begin{array}{l}
\textit{OMGroupedAssociation} \\
\hline
\textit{OMAssociation} \\
\textit{groupedAssociation} : \mathbb{P}\, \textit{ASSOCIATION} \\
\textit{aggAssociation}, \textit{genAssociation} : \mathbb{P}\, \textit{ASSOCIATION} \\
\textit{parent} : \textit{ASSOCIATION} \rightarrowtail\!\!\!\!\!\!\!+\;\, \textit{ROLE} \\
\textit{child} : \textit{ASSOCIATION} \rightarrowtail\!\!\!\!\!\!\!+\;\, \textit{ROLE} \\
\hline
\textit{groupedAssociation} \subseteq \textit{binaryAssociation} \\
\langle \textit{aggAssociation}, \textit{genAssociation} \rangle \ \mathsf{partition}\ \textit{groupedAssociation} \\
\textit{parent} \in \textit{groupedAssociation} \rightarrowtail \textit{singularRole} \\
\textit{child} \in \textit{groupedAssociation} \rightarrowtail \textit{role} \\
\forall\, a : \textit{groupedAssociation} \bullet \textit{hasRole}(\!|\ \{a\}\ |\!) = \{\textit{parent}(a), \textit{child}(a)\}
\end{array}
$$

- The grouped associations are binary associations.
- A grouped association is either an aggregation or a generalization association.
- Each grouped association has one parent role, which is a singular role, according to the definition of aggregation and generalization.

- Each grouped association has one child role.
- The roles of a grouped association is made up of a parent and a child of the parent.

Schema *OMGrouping* defines the mapping between the grouping relationships and the grouped associations.

OMGrouping

$$
\begin{array}{l}
OMGroupedClass \\
OMGroupedAssociation \\
collectedIn : ASSOCIATION \pfun GROUPING \\
\hline
collectedIn \in (aggAssociation \pfunsur aggregation) \cup (genAssociation \pfunsur generalization) \\
\forall\, a : groupedAssociation;\ g : grouping \mid collectedIn(a) = g \bullet \\
\quad playerOfRole(parent(a)) = single(g) \land \\
\quad playerOfRole(child(a)) \in group(\!|\, \{g\} \,|\!) \\
\forall\, g : grouping \bullet \#(group(\!|\, \{g\} \,|\!)) = \#(collectedIn^{\sim}(\!|\, \{g\} \,|\!))
\end{array}
$$

- Each grouped association belongs to a grouping relationship. A grouping relationship may be associated with a set of grouped associations.
- For a grouped association of an aggregation, the parent role is the assembly class, and the child role is one of the component classes. It is similar to generalization.
- The number of the grouped associations of an aggregation (or generalization) is the same with the number of the components (or subclasses) in the aggregation (or generalization).

Besides the similar characteristics discussed above, both aggregations and generalizations are

- transitive, that is, if a is a part of (or a kind of) b and b is a part of (or a kind of) c, then a is a part of (or a kind of) c; and
- asymmetric, that is, if a is a part of (or a kind of) b, then b is not a part of (or a kind of) a.

We define a relation po that is transitive and antisymmetric.

$$[X]$$

$$po : X \leftrightarrow X$$

$$\forall a, b, c : X \mid a \mapsto b \in po \land b \mapsto c \in po \bullet a \mapsto c \in po$$

$$\forall a, b : X \mid a \mapsto b \in po \bullet b \mapsto a \notin po$$

The remainder of this section discusses the properties specific to aggregation or generalization.

Aggregations and Their Characteristics

A component and the assembly in an aggregate are in the "is-part-of" relationship, that is, the component is a part of the assembly.

OMIsPartOf

$$OMGrouping$$

$$isPartOf : CLASS \leftrightarrow CLASS$$

$$isPartOf = \{a : aggregation;\ c : class \mid c \in group(\!|\ \{a\}\ |\!) \bullet (c, single(a))\}$$

$$isPartOf^+ \subset po[CLASS]$$

- *isPartOf* relates each individual component to its assembly.
- An aggregation is transitive and asymmetric.

OMT's examples show that aggregation associations do not have names.

OMAggregationName

$$OMName$$
$$OMGroupedAssociation$$

$$aggAssociation \cap \operatorname{dom} nameOfAssoc = \emptyset$$

The existence of a component object may depend on the existence of the aggregate object of which it is a part (see [RBP+91]: p.38). In other cases, the component objects have an independent existence. Schema *OMDependence* defines a relation *existenceDependOn* to specify this dependency relationship. Schema *OMRTDependence* in Section 4.3.6 further specifies the semantics of dependency in terms of run-time concepts.

OMDependence

$$OMIsPartOf$$
$$existenceDependOn : CLASS \leftrightarrow CLASS$$

$$existenceDependOn \subset isPartOf$$

Propagation

In aggregation associations, some properties of the assembly *propagate* to the components, possibly with some local modifications (see [RBP+91]: p.37). As an example, the speed and the location of a door handle are obtained from the door of which it is a part; the door in turn obtains its properties from the car of which it is a part. *Propagation* of an operation (also called *triggering*) is the automatic application of an operation, according to propagation rules, to a network of objects when the operation is applied to some starting object (see [RBP+91]: p.60). This means that both attributes and operations of the assembly can propagate to the components.

Function *featurePropagate* in the following schema specifies what attributes and operations in aggregations may propagate.

OMPropagation

$$
\begin{array}{l}
OMGrouping \\
featurePropagate : AGGREGATION \leftrightarrow FEATURE \\
\hline
featurePropagate \in aggregation \leftrightarrow (attribute \cup operation) \\
\forall\, a : \mathrm{dom}\, featurePropagate;\ f : feature \mid f \in featurePropagate(\!|\ \{a\}\ |\!) \bullet \\
\quad f \in hasFeature(\!|\ \{single(a)\}\ |\!) \land \\
\quad (\forall\, c : group(\!|\ \{a\}\ |\!) \bullet f \in hasFeature(\!|\ \{c\}\ |\!))
\end{array}
$$

- Certain operations and attributes of some aggregations may propagate.
- The propagation of an operation or an attribute of an assembly to its components means that the components also possess the operation or attribute.

The local modification of an operation is to apply a different method of the operation to the components. This is implicitly formalized by function *methodLookup* in schema *OMMethod* in Section 4.1.2.

Putting the schemas for aggregation together, we obtain

$$
\begin{array}{l}
OMAggregation \mathrel{\widehat=} \\
\quad OMIsPartOf \land \\
\quad OMAggregationName \land \\
\quad OMDependence \land OMPropagation
\end{array}
$$

Generalizations and Their Characteristics

A generalization association shares fewer similarities with other types of association. It is a way of structuring the description of a single object. However, it still can be formalized as a special association, with the roles played by the

subclasses having multiplicity $\{0, 1\}$. In this way, an instance of a subclass is simultaneously an instance of the superclass; but not vice-versa.

OMGeneralizationBasic

$$OMGrouping$$

$$\forall g : generalization \bullet (collectedIn^{\sim} \fatsemi child)(\!| \{g\} |\!) \subseteq optionalRole$$

A subclass and the superclass in a generalization are in the "is-kind-of" relationship, that is, a subclass is a refined version of the superclass. Schema *OMIsKindOf* defines three relations between classes:

- *isKindOf* from subclasses to superclasses,
- *isDescendentOf* from descendents to ancestors, and
- *isAncestorOf* from ancestors to descendents.

OMIsKindOf

$$OMGeneralizationBasic$$
$$isKindOf : CLASS \leftrightarrow CLASS$$
$$isDescendentOf : CLASS \leftrightarrow CLASS$$
$$isAncestorOf : CLASS \leftrightarrow CLASS$$

$$isKindOf = \{g : generalization;\ sub : class \mid sub \in group(\!| \{g\} |\!) \bullet sub \mapsto single(g)\}$$

$$isDescendentOf = isKindOf^{+}$$

$$isDescendentOf \cap \mathrm{id}\ class = \emptyset$$

$$isDescendentOf \subset po[CLASS]$$

$$isAncestorOf = isDescendentOf^{\sim}$$

- *isKindOf* associates each subclass with its superclass.

- *isDescendentOf* relates the descendents to their ancestors.
- Any class cannot be a descendent of its own.
- A generalization is transitive and asymmetric.
- *isAncestorOf* relates the ancestors to their descendents. A class should not be the ancestor of itself.

The property of inheritance in a generalization is formalized as follows.

OMInheritance

$$OMIsKindOf$$

$$\forall super, sub : class \mid sub \mapsto super \in isKindOf \bullet hasFeature(\!| \{super\} |\!) \subseteq hasFeature(\!| \{sub\} |\!)$$

- Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to inherit the features of its superclass.

A *discriminator* is defined to be an attribute of enumeration type that indicates which property of an object is being abstracted by a particular generalization. OMT views the discriminator as simply a name for the basis of the generalization, and it is optional. Thus, discriminators would appear to be closely resembling names of associations.

OMGeneralizationName

$$OMName$$
$$OMGrouping$$

$$\forall g : generalization \bullet (collectedIn^{\sim}(\!| \{g\} |\!) \cap \operatorname{dom} nameOfAssoc = \emptyset \lor (\forall a1, a2 : collectedIn^{\sim}(\!| \{g\} |\!) \bullet nameOfAssoc(a1) = nameOfAssoc(a2)))$$

- The generalization associations that belong to the same generalization construct either have the same name or have no name.

Overriding

A subclass may *override* a superclass feature. First of all, we introduce a schema with a generic parameter X. The function *featureOverride* in the schema specifies that when some features (attributes and operations) of a superclass are inherited by a subclass, elements of type X that are related to these features can be overridden by other elements of the same type.

$$
\begin{array}{|l}
\hline
OMOverride[X] \\
OMIsKindOf \\
featureOverride : CLASS \pfun (CLASS \pfun (FEATURE \pfun X)) \\
\hline
\dom featureOverride \subseteq \ran isKindOf \\
\forall\, c : \dom featureOverride \bullet \\
\quad featureOverride(c) \in isKindOf^{\sim}(\!|\, \{c\} \,|\!) \pfun \\
\qquad (hasFeature(\!|\, \{c\} \,|\!) \cap (attribute \cup operation) \pfun X) \\
\hline
\end{array}
$$

The default value of an attribute can be overridden.

$OMDefaultValueOverride$

$OMOverride[ATOMIC\_VALUE][defaultValOverride/featureOverride]$

$$
\begin{array}{l}
\forall\, super, sub : class;\ a : attribute \mid \\
\quad sub \mapsto super \in isKindOf\ \wedge \\
\quad a \in \mathrm{dom}(defaultValOfAttr(super)) \bullet \\
\qquad a \in \mathrm{dom}(defaultValOfAttr(sub))\ \wedge \\
\qquad (a \in \mathrm{dom}(defaultValOverride(super)(sub)) \Rightarrow \\
\qquad\qquad defaultValOverride(super)(sub)(a) = \\
\qquad\qquad defaultValOfAttr(sub)(a) \neq \\
\qquad\qquad defaultValOfAttr(super)(a)\ \wedge \\
\qquad a \notin \mathrm{dom}(defaultValOverride(super)(sub)) \Rightarrow \\
\qquad\qquad defaultValOfAttr(sub)(a) = \\
\qquad\qquad defaultValOfAttr(super)(a))
\end{array}
$$

- The attribute a of the subclass *sub* is inherited from its superclass *super*. If the default value of a is overridden, then the default values of a in *super* and *sub* are different; otherwise, they are the same.

The method of an operation can be overridden by another method of the same operation, but the signature of an operation should never be overridden.

OMMethodOverride

$$OMOverride[METHOD][methodOverride/featureOverride]$$

$$\begin{aligned}
&\forall\, super, sub : class;\ o : operation \mid \\
&\quad sub \mapsto super \in isKindOf \ \wedge \\
&\quad o \in \operatorname{ran} implement \ \wedge \\
&\quad super \in \operatorname{dom}(methodLookup(o)) \bullet \\
&\qquad sub \in \operatorname{dom}(methodLookup(o)) \wedge \\
&\qquad (o \in \operatorname{dom}(methodOverride(super)(sub)) \Rightarrow \\
&\qquad\qquad methodOverride(super)(sub)(o) = \\
&\qquad\qquad methodLookup(o)(sub) \neq \\
&\qquad\qquad methodLookup(o)(super) \ \wedge \\
&\qquad o \notin \operatorname{dom}(methodOverride(super)(sub)) \Rightarrow \\
&\qquad\qquad methodLookup(o)(sub) = \\
&\qquad\qquad methodLookup(o)(super))
\end{aligned}$$

- The operation o of the subclass *sub* is inherited from its superclass *super*. If the implementation of o is overridden, then the methods of o in *super* and *sub* are different; otherwise, they are the same.

Putting the schemas for generalization together, we obtain,

$$\begin{aligned}
&OMGeneralization \mathrel{\widehat=} \\
&\quad OMGeneralizationBasic \ \wedge \\
&\quad OMIsKindOf \ \wedge \ OMInheritance \ \wedge \\
&\quad OMGeneralizationName \ \wedge \\
&\quad OMDefaultValueOverride \ \wedge \ OMMethodOverride
\end{aligned}$$

4.1.6 Modules

A *module* is a logical construct for grouping classes, associations, and generalizations. We introduce a given set

$$[MODULE]$$

for all of the possible modules.

An object model consists of several modules; each module contains a set of classes.

OMModule

$$OMDTData$$
$$module : \mathbb{P}\, MODULE$$
$$containedIn : CLASS \leftrightarrow MODULE$$

$$\text{dom}\, containedIn = class \land \text{ran}\, containedIn = module$$

- The classes of the object model can be grouped into several modules.

4.1.7 Object Model with Basic Concepts

Schema *OMDTBasic* defines the object model involving the concepts discussed in previous sections.

$$OMDTBasic \mathrel{\widehat=}$$
$$OMDTData \land OMFeature \land OMName \land$$
$$OMAssociation \land OMAggregation \land$$
$$OMGeneralization \land OMModule$$

4.2 Advanced Design-Time Concepts

This section formalizes the advanced object modeling concepts, based on the original material in Chapter 4 of [RBP+91].

4.2.1 Fixed, Variable, and Recursive Aggregations

There are three kinds of aggregations: fixed, variable, or recursive (see [RBP+91]: p.59).

OMAggregationType

$$
\begin{array}{l}
OMAggregation \\
OMGeneralization \\
fixedAgg : \mathbb{P}\, AGGREGATION \\
variableAgg : \mathbb{P}\, AGGREGATION \\
recursiveAgg : \mathbb{P}\, AGGREGATION \\
\hline
fixedAgg \cup variableAgg \cup recursiveAgg \subseteq aggregation \\
\mathsf{disjoint}\ \langle fixedAgg, variableAgg, recursiveAgg \rangle \\
\forall\, fa : fixedAgg;\ comp : role \mid \\
\quad (child^{\sim} \fatsemi collectedIn)(comp) = fa \bullet \\
\quad\quad (\exists\, n : \mathbb{N}_1 \bullet multiplicityOfRole(\!|\, \{comp\} \,|\!) = \{n\}) \\
\forall\, ra : recursiveAgg;\ asse : class \mid \\
\quad asse = single(ra) \bullet \\
\quad\quad (\exists\, comp : group(\!|\, \{ra\} \,|\!) \bullet comp = asse \lor \\
\quad\quad\quad comp \in isKindOf(\!|\, \{asse\} \,|\!))
\end{array}
$$

- An aggregation can be fixed, variable, or recursive.
- The three types of aggregations are distinct.
- A fixed aggregate has a fixed structure; the number and types of subparts are predefined.
- A recursive aggregate, directly or indirectly, contains an instance of the same kind of aggregate; the number of potential levels is unlimited.

There is no predicate for specifying variable aggregations, since they are the most general type of aggregations. A variable aggregation differs from a fixed aggregation in that the number of its parts may vary. Generally, the multiplicity of a component class in a variable aggregation is a set of integers, which implies that the number of the parts of the assembly may vary. The variable aggregate differs from a recursive aggregate in that the assembly does not contain the instances of its own or of its ancestors; therefore the potential levels of aggregation are limited.

4.2.2 Multiple Inheritance

Multiple inheritance permits a class to have more than one superclass and to inherit features from all parents. A class with more than one superclass is called a *join class*. Multiple inheritance can be viewed as a relationship across the superclasses and the subclasses in an object model. The multiple inheritance in the object model is formalized as follows.

$$
\begin{array}{|l}
\hline
OMMultipleInheritance \\
OMGeneralization \\
joinClass : \mathbb{P}\, CLASS \\
\hline
joinClass \subset \mathrm{dom}\, isKindOf \\
\forall\, jc : joinClass \bullet \#(isKindOf(\!| \{jc\} |\!)) > 1 \\
\hline
\end{array}
$$

- Join classes are a subset of classes. A join class is always a subclass in some generalizations.
- A join class has more than one superclass.

4.2.3 Disjoint and Overlapping Generalizations

The generalization subclasses may or may not be disjoint. Schema *OMGeneralizationType* defines that disjoint generalizations and overlapping generalizations are subsets of generalizations in an object model. The two types of generalizations are distinct.

OMGeneralizationType

$$
\begin{array}{l}
OMGeneralization \\
OMMultipleInheritance \\
disjointGen : \mathbb{P}\, GENERALIZATION \\
overlappingGen : \mathbb{P}\, GENERALIZATION \\
\hline
\langle disjointGen, overlappingGen \rangle \ \mathsf{partition}\ generalization \\
\forall\, c1, c2 : class;\ g : generalization \mid \\
\quad \{c1, c2\} \subseteq group(\!|\, \{g\} \,|\!) \bullet \\
\quad\quad g \in disjointGen \Rightarrow \\
\quad\quad \mathrm{dom}(isDescendentOf \rhd \{c1\}) \cap \mathrm{dom}(isDescendentOf \rhd \{c2\}) = \emptyset \\
\forall\, jc : joinClass;\ c1, c2 : class;\ g : generalization \mid \\
\quad \{c1, c2\} \subseteq group(\!|\, \{g\} \,|\!) \land \\
\quad \{c1, c2\} \subset isDescendentOf(\!|\, \{jc\} \,|\!) \bullet g \in overlappingGen
\end{array}
$$

- A generalization is either disjoint or overlapping.
- A class never multiply inherits from two classes in the same disjoint generalization.
- If there is a class that multiply inherits from two classes in the same generalization, then the generalization is an overlapping generalization.

The semantics of disjoint and overlapping generalizations is further discussed in Section 4.3.8

4.2.4 Generalization as Extension and Restriction

Extension

A subclass may add new features. This is called *extension*. The extension of a generalization by adding new attributes or operations is formalized in schema *OMGenExtension*.

OMGenExtension

$$
\begin{array}{l}
OMGeneralization \\
extension : CLASS \pfun (CLASS \rel FEATURE) \\
\hline
\dom extension \subseteq \ran isKindOf \\
\forall c : \dom extension \bullet extension(c) \in \\
\qquad isKindOf^{\sim}(\!| \{c\} |\!) \rel (attribute \cup operation) \setminus hasFeature(\!| \{c\} |\!) \\
\forall c, c' : class; f : feature \mid c' \mapsto f \in extension(c) \bullet definedIn(f) = c'
\end{array}
$$

- The domain of function *extension* is a set of superclasses.
- For a given generalization, function *extension* maps each new feature (attribute or operation) to the subclass to which the feature is added.
- The extension of a generalization means that certain subclasses have attributes or operations that the superclass does not have.

Restriction

A subclass may also constrain its ancestors' attributes. This is called *restriction*. A restriction restricts the values an attribute can assume, or renames the inherited attributes in a subclass. A restriction should follow certain rule. Let

$[RULE]$

represent these rules.

Function *restrict* in the following schema specifies that the values of some attributes, inherited from superclasses, of subclasses may be restricted under certain rules. *restrict* has the similar signature to the function *featureOverride*.

OMGenAttrRestriction

$$OMGeneralization$$
$$OMOverride[RULE][restrict/featureOverride]$$

Further semantics of value restrictions will be discussed in Section 4.2.5.

Inherited attributes may be renamed. The following schema formalizes the renaming of the attributes in subclasses.

OMGenRenaming

$$OMName$$
$$OMGeneralization$$
$$OMOverride[NAME][rename/featureOverride]$$

$$\forall super, sub : class;\ f : feature \mid$$
$$sub \mapsto super \in isKindOf \land$$
$$f \in \mathrm{dom}(nameOfFeature(super)) \bullet$$
$$f \in \mathrm{dom}(nameOfFeature(sub)) \land$$
$$(f \in \mathrm{dom}(rename(super)(sub)) \Rightarrow rename(super)(sub)(f) = nameOfFeature(sub)(f) \neq nameOfFeature(super)(f) \land$$
$$f \notin \mathrm{dom}(rename(super)(sub)) \Rightarrow nameOfFeature(sub)(f) = nameOfFeature(super)(f))$$

- While renaming an inherited attribute, one assigns the attribute in the subclass a name that is different from the name for the same attribute in the superclass. Otherwise, the names of the attribute are the same in both superclass and subclass.

4.2.5 Constraints

OMT mentions three kinds of constraints. They are constraints on attributes, constraints on links, and general constraints. Constraints on attributes restrict the values that the attributes can assume (see [RBP+91]: pp.73-74). In Section 4.2.4, function *restrict* in schema *OMGenAttrRestriction* is defined to formalize the attribute restrictions in generalizations. Here, we define a more generic function used to formalize constraints on the attributes of any kinds of classes or associations under certain restriction rules.

OMAttributeConstraintBasic

$$
\begin{array}{l}
OMFeature \\
generalRestrict : ENTITY \pfun (ATTRIBUTE \rel RULE) \\
\hline
\dom generalRestrict \subseteq entity \\
\forall\, c : \dom generalRestrict \bullet \\
\quad \dom(generalRestrict(c)) \subseteq hasFeature(\!|\, \{c\} \,|\!) \cap attribute
\end{array}
$$

Since the definition for "rule" is very vague in [RBP+91], we can only formalize it at a very abstract level.

Certain constraint rules restrict the range of the values that an attribute can assume. Given an entity, an attribute of the entity and a constraint rule, function *constraint* defined in the following schema returns a set of atomic values which are values the attributes can take on.

$$
\begin{array}{|l}
\hline
\textit{OMAttributeConstraint} \\
\hline
\textit{OMGenAttrRestriction} \\
\textit{OMAttributeConstraintBasic} \\
\textit{constraint} : \textit{ENTITY} \times \textit{ATTRIBUTE} \pfun (\textit{RULE} \rel \textit{ATOMIC\_VALUE}) \\
\hline
\dom \textit{constraint} \subseteq \{ e : \textit{entity};\ a : \textit{attribute} \mid a \in \textit{hasFeature}(\!| \{e\} |\!) \bullet (e, a) \} \\
\forall\, e : \textit{entity};\ a : \textit{attribute};\ r : \textit{RULE};\ \textit{vs} : \power \textit{ATOMIC\_VALUE} \mid \\
\quad \textit{vs} = (\textit{constraint}(e, a))(\!| \{r\} |\!) \bullet \\
\quad\quad ((\exists\, e' : \textit{entity} \bullet r \in (\textit{restrict}(e')(e))(\!| \{a\} |\!)) \lor \\
\quad\quad r \in (\textit{generalRestrict}(e))(\!| \{a\} |\!)) \land \\
\quad\quad \textit{vs} \subset \textit{valueOfType}(\!| \{\textit{typeOfAttr}(a)\} |\!) \\
\hline
\end{array}
$$

- The domain of *constraint* is a set of pairs of entities and the attributes of the entities.
- If a rule is applied to an attribute in an entity to restrict the attribute values, then the attribute can only take on the values within a restricted range.

After formalizing the constraints on attributes, we are able to define the value range for each attribute of a given entity.

OMAttributeRange

$$
\begin{array}{l}
OMAttributeConstraint \\
valueRange : ENTITY \pfun (ATTRIBUTE \rel ATOMIC\_VALUE) \\
\hline
\dom valueRange = \dom(hasFeature \rhd attribute) \\
\forall\, e : \dom valueRange \bullet \\
\quad \dom(valueRange(e)) = hasFeature(\!| \{e\} |\!) \cap attribute \\
\forall\, e : entity;\ a : attribute \bullet \\
\quad (e, a) \in \dom constraint \Rightarrow \\
\quad\quad (valueRange(e))(\!| \{a\} |\!) = valueOfType(\!| \{typeOfAttr(a)\} |\!) \cap \\
\quad\quad\quad \bigcap \{r : RULE \bullet (constraint(e, a))(\!| \{r\} |\!)\} \land \\
\quad (e, a) \notin \dom constraint \Rightarrow \\
\quad\quad (valueRange(e))(\!| \{a\} |\!) = valueOfType(\!| \{typeOfAttr(a)\} |\!)
\end{array}
$$

- For each attribute in a class or an association, if there are constraint rules applied to it, then the value range of the attribute is the intersection of the restricted range by each rule; otherwise the value range of the attribute is the same as the original range.

The default value of an attribute can only take on the value within the value range of the attribute.

OMDefaultValue

$$
\begin{array}{l}
OMDefaultValueBasic \\
OMAttributeRange \\
\hline
\forall\, e : entity;\ a : attribute;\ v : ATOMIC\_VALUE \mid \\
\quad v = defaultValOfAttr(e)(a) \bullet v \in (valueRange(e))(\!| \{a\} |\!)
\end{array}
$$

If there are constraints on an inherited attribute, then the value range of the attribute in the subclass is a subset of the value range of the attribute in the superclass.

OMSubclassAttributeRange

OMGeneralization

OMAttributeRange

$$\forall\, super, sub : class;\ a : attribute \mid$$
$$a \in \mathrm{dom}(restrict(super)(sub)) \bullet$$
$$(valueRange(sub))(\!|\ \{a\}\ |\!) \subset (valueRange(super))(\!|\ \{a\}\ |\!)$$

Constraints on links include multiplicity, "ordered", and "qualified" (see [RBP+91]: p.74), which are formalized in Section 4.1.4.

OMT does not give explicit semantics on general constraints.

4.2.6 Abstract Classes

A class in a generalization relationship may be an abstract class. An *abstract class* is a class that has no direct instances but whose descendent classes have direct instances. A *concrete class* is a class that is instantiable; that is, it can have direct instances. In schema *OMClassType*1, we define three sets of classes:

- a set of abstract classes,
- a set of concrete classes, and
- a set of leaf classes.

*OMClassType*1

$$
\begin{array}{l}
OMGeneralization \\
abstractClass : \mathbb{P}\, CLASS \\
concreteClass : \mathbb{P}\, CLASS \\
leafClass : \mathbb{P}\, CLASS \\
\hline
concreteClass \subseteq class \\
abstractClass \subseteq \mathrm{dom}(isAncestorOf \rhd concreteClass) \\
leafClass = (\mathrm{ran}\, isAncestorOf \setminus \mathrm{dom}\, isAncestorOf) \subseteq concreteClass
\end{array}
$$

- A concrete class is a class in the object model.
- Any abstract class must have one or more concrete descendents.
- A leaf class is a leaf in inheritance trees. Only a concrete class may be a leaf class.

The semantics of concrete and abstract classes is further expressed in terms of run-time concepts in Section 4.3.2.

Accordingly, an operation can be abstract or concrete. An *abstract operation* is the protocol of an operation without a corresponding method.

*OMOperationType*2

$$
\begin{array}{l}
OMMethod \\
OMClassType1 \\
abstractOp : \mathbb{P}\, OPERATION \\
concreteOp : \mathbb{P}\, OPERATION \\
\hline
\langle abstractOp, concreteOp \rangle \ \mathsf{partition}\ operation \\
concreteOp = \mathrm{ran}\, implement \\
\mathrm{dom}(opDefinedIn \rhd concreteClass) \subseteq concreteOp
\end{array}
$$

- An operation is either an abstract operation or a concrete operation.
- A concrete operation has methods as its implementations.
- A concrete class may not contain abstract operations.

4.2.7 Metadata

Metadata is data that describes other data. OMT discusses two types of metadata: patterns and class descriptors. An example of a *pattern* is class `CarModel`, which is the pattern of class `Car`. A *class descriptor* has *class attribute*s, which describe values common to an entire class of objects, and *class operations*, which are operations on the class itself (see [RBP+91]: pp.69-71).

Schema *OMPattern* denotes

- a set of pattern classes in the object model, and
- a function from pattern classes to instance classes, indicating the instantiation relationship.

OMPattern

$$
\begin{array}{l}
OMDTData \\
pattern : \mathbb{P}\, CLASS \\
instantiation : CLASS \pfun CLASS \\
\hline
pattern \subset class \\
instantiation \in class \pfun pattern \\
instantiation \cap \mathrm{id}\, class = \emptyset
\end{array}
$$

- Pattern classes are a subset of classes in the object model.
- Pattern classes may have some instance classes.

- A class cannot be the pattern of itself.

Schema *OMDescriptor* denotes

- a set of class descriptors in the object model, and
- a set of class features, i.e., class attributes and class operations, of the class descriptors.

$$
\begin{array}{|l}
\textit{OMDescriptor} \\
\hline
\textit{OMFeature} \\
\textit{classDescriptor} : \mathbb{P}\, \textit{CLASS} \\
\textit{classFeature} : \mathbb{P}\, \textit{FEATURE} \\
\hline
\textit{classDescriptor} \subset \textit{class} \\
\textit{classFeature} \subset \textit{attribute} \cup \textit{operation} \\
\forall f : \textit{classFeature};\ c : \textit{class} \mid \textit{definedIn}(f) = c \bullet c \in \textit{classDescriptor} \\
\hline
\end{array}
$$

- Class descriptors are a subset of classes in the object model.
- Class attributes and class operations are subsets of attributes and operations in the object model.
- A class feature must belong to a class descriptor, not to other kind of classes.

The following schema combines the above two kinds of metadata.

$$\textit{OMMetadata} \mathrel{\widehat=} \textit{OMPattern} \land \textit{OMDescriptor}$$

4.2.8 Candidate Keys

A *candidate key* is a minimal set of attributes that uniquely identifies an object or a link (see [RBP+91]: pp.71-73). OMT does not consider the effect of inheritance on candidate keys to avoid making any presumptions about whether

the same set of attributes could serve as a candidate key for more than one class. We introduce a unique identity for each candidate key specification:

$[KEY]$

The following schema defines

- a set of candidate keys in the object model, and
- a relation specifying the features that form each candidate key.

OMCandidateKeyBasic

$$OMFeature$$
$$key : \mathbb{P}\, KEY$$
$$hasKeyElement : KEY \leftrightarrow FEATURE$$

$$\operatorname{dom} hasKeyElement = key$$
$$\operatorname{ran} hasKeyElement \subseteq feature$$

- Every candidate key is composed of a set of features.

Schema *OMCandidateKeyBasic* denotes

- a set of entities that have specified candidate keys,
- a mapping from each candidate key to its entity, and
- a mapping from each entity to its candidate keys.

OMCandidateKey

$$OMCandidateKeyBasic$$

$$keyedEntity : \mathbb{P}\, ENTITY$$

$$isKeyOf : KEY \pfun ENTITY$$

$$theKey : ENTITY \rel KEY$$

$$keyedEntity \subseteq entity$$

$$isKeyOf \in key \surj keyedEntity$$

$$theKey = isKeyOf^{\sim}$$

$$\forall\, c : keyedEntity \cap class \bullet$$
$$(\forall\, k : theKey(\!|\, \{c\} \,|\!) \bullet$$
$$hasKeyElement(\!|\, \{k\} \,|\!) \subseteq attribute \cap hasFeature(\!|\, \{c\} \,|\!))$$

$$\forall\, a : keyedEntity \cap association \bullet$$
$$(\forall\, k : theKey(\!|\, \{a\} \,|\!) \bullet$$
$$hasKeyElement(\!|\, \{k\} \,|\!) \subseteq role \cap hasFeature(\!|\, \{a\} \,|\!))$$

$$\forall\, k1, k2 : key \mid$$
$$isKeyOf(k1) = isKeyOf(k2) \land k1 \neq k2 \bullet$$
$$\neg\, hasKeyElement(\!|\, \{k1\} \,|\!) \subseteq hasKeyElement(\!|\, \{k2\} \,|\!) \land$$
$$\neg\, hasKeyElement(\!|\, \{k2\} \,|\!) \subseteq hasKeyElement(\!|\, \{k1\} \,|\!)$$

- Entities in the object model may have candidate keys.
- Each candidate key is associated with an entity.
- An entity may have one or more candidate keys.
- A candidate key for a class is a combination of its attributes.
- A candidate key for an association is a combination of its related roles.

- If an entity has more than one candidate key, the feature combination of one key is different from that of other keys.

Furthermore, the patterns of candidate keys for many-to-many, one-to-many, and optional-to-one binary associations are formalized as follows (see pp.71-73 of [RBP+91] for informal explanation).

A many-to-many association requires both related roles to uniquely identify each link:

OMm_mKey

OMAssociation

OMCandidateKey

$$\forall\, m : mul\_mulAssoc \cap keyedEntity \bullet (\exists\, k : key \bullet theKey(\!|\, \{m\} \,|\!) = \{k\} \land hasKeyElement(\!|\, \{k\} \,|\!) = hasRole(\!|\, \{m\} \,|\!))$$

A one-to-many association has a single candidate key, that is, the role on the "many" end:

OMone_mKey

OMAssociation

OMCandidateKey

$$\forall\, o : one\_mulAssoc \cap keyedEntity;\ r : multipleRole \mid r \in hasRole(\!|\, \{o\} \,|\!) \bullet (\exists\, k : key \bullet theKey(\!|\, \{o\} \,|\!) = \{k\} \land hasKeyElement(\!|\, \{k\} \,|\!) = \{r\})$$

An optional-to-one association has two candidate keys, that is, either of the roles:

OMopt_oneKey

OMAssociation

OMCandidateKey

$$\forall p : one\_optAssoc \cap keyedEntity;\ r1 : optionalRole;\ r2 : singularRole \mid$$
$$\{r1, r2\} = hasRole(\!|\ \{p\}\ |\!) \bullet$$
$$(\exists k1, k2 : key \bullet theKey(\!|\ \{p\}\ |\!) = \{k1, k2\} \land$$
$$hasKeyElement(\!|\ \{k1\}\ |\!) = \{r1\} \land$$
$$hasKeyElement(\!|\ \{k2\}\ |\!) = \{r2\})$$

The following schema combines the above schemas related to candidate keys.

$$OMCandidateKeys \mathrel{\widehat=}$$
$$OMCandidateKey \land$$
$$OMm\_mKey \land OMone\_mKey \land OMopt\_oneKey$$

More about the semantics of candidate keys will be discussed in Section 4.3.9.

4.2.9 Derived Classes, Associations, and Attributes

Classes can be classified into base and derived classes. A *base class* is independent of any other classes; and a *derived class* is defined as a function of one or more classes, which in turn can be derived. Similarly, there are also *base association*, *derived association*, *base attribute*, and *derived attribute*. For example, "birthday" may be a base attribute of class `Person`, and "age" may be a derived attribute, the value of which can be derived by a function of "birthday" and "current date". Let

$$[FUNCTION]$$

represent the functions that determine the derived entities and features.

Schema *DerivedElement* describes

- a derived element *derived* with type X, where X may be $CLASS$, $ASSOCIATION$, or $ATTRIBUTE$,
- a set of elements, from which the derived element is derived, and
- a function which is used to compute the derived element.

$$
\begin{array}{|l}
\hline
DerivedElement[X] \\
derived : X \\
deriving : \mathbb{P}\, X \\
computeDerived : FUNCTION \\
\hline
\end{array}
$$

Schema *OMClassType*2 denotes

- a set of base classes in the object model,
- a set of derived classes in the object model, and
- a function from each derived class to the schema describing how the class is derived.

$$
\begin{array}{|l}
\hline
OMClassType2 \\
OMDTData \\
baseClass : \mathbb{P}\, CLASS \\
derivedClass : \mathbb{P}\, CLASS \\
clsDeriving : CLASS \pfun DerivedElement[CLASS] \\
\hline
\langle baseClass, derivedClass \rangle \ \mathsf{partition}\ class \\
\mathrm{dom}\, clsDeriving = derivedClass \\
\forall\, dc : derivedClass \bullet (clsDeriving(dc)).derived = dc \land \\
\qquad\qquad (clsDeriving(dc)).deriving \subset class \\
\hline
\end{array}
$$

- Base classes and derived classes partition the class set in the object model.
- Each derived class satisfies a $DerivedElement[CLASS]$ schema.

Similarly, the base and derived associations and the base and derived attributes are defined as follows.

$$
\begin{array}{|l}
\hline
OMAssociationType \\
\hline
OMDTData \\
baseAssoc : \mathbb{P}\, ASSOCIATION \\
derivedAssoc : \mathbb{P}\, ASSOCIATION \\
assocDeriving : ASSOCIATION \pfun DerivedElement[ASSOCIATION] \\
\hline
\langle baseAssoc, derivedAssoc \rangle \ \mathsf{partition}\ association \\
\mathrm{dom}\, assocDeriving = derivedAssoc \\
\forall\, da : derivedAssoc \bullet (assocDeriving(da)).derived = da \land \\
\qquad\qquad (assocDeriving(da)).deriving \subset association \\
\hline
\end{array}
$$

- Base associations and derived associations partition the association set in the object model.
- Each derived association satisfies a $DerivedElement[ASSOCIATION]$ schema.

$$
\begin{array}{|l}
\textit{OMAttributeType} \\
\hline
\textit{OMDTData} \\
\textit{baseAttr} : \mathbb{P}\, \textit{ATTRIBUTE} \\
\textit{derivedAttr} : \mathbb{P}\, \textit{ATTRIBUTE} \\
\textit{attrDeriving} : \textit{ATTRIBUTE} \pfun \textit{DerivedElement}[\textit{ATTRIBUTE}] \\
\hline
\langle \textit{baseAttr}, \textit{derivedAttr} \rangle \ \mathsf{partition}\ \textit{attribute} \\
\mathrm{dom}\, \textit{attrDeriving} = \textit{derivedAttr} \\
\forall\, da : \textit{derivedAttr} \bullet (\textit{attrDeriving}(da)).\textit{derived} = da \land \\
\qquad\qquad (\textit{attrDeriving}(da)).\textit{deriving} \subset \textit{attribute} \\
\hline
\end{array}
$$

- Base attributes and derived attributes partition the attributes set in the object model.
- Each derived attribute satisfies a $\textit{DerivedElement}[\textit{ATTRIBUTE}]$ schema.

4.2.10 Object Model with Advanced Concepts

Schema *OMDTAdvanced* defines the object model involving the advanced concepts.

$$
\begin{array}{l}
\textit{OMDTAdvanced} \mathrel{\widehat=} \\
\quad \textit{OMAggregationType} \land \textit{OMGeneralizationType} \land \\
\quad \textit{OMMultipleInheritance} \land \\
\quad \textit{OMGenExtension} \land \textit{OMGenAttrRestriction} \land \textit{OMGenRenaming} \land \\
\quad \textit{OMAttributeConstraint} \land \textit{OMAttributeRange} \land \textit{OMDefaultValue} \land \\
\quad \textit{OMClassType}1 \land \textit{OMOperationType}2 \land \\
\quad \textit{OMMetadata} \land \\
\quad \textit{OMCandidateKeys} \land \\
\quad \textit{OMClassType}2 \land \textit{OMAssociationType} \land \textit{OMAttributeType}
\end{array}
$$

A design-time object model combines both basic and advanced concepts and their properties.

$$OMDTModel \mathrel{\widehat=} OMDTBasic \land OMDTAdvanced$$

4.3 Run-Time Concepts

A run-time model represents the internal operation of a running application. It is described with a set of run-time concepts, which represent the instances of the design-time elements of the system. In choosing how to formalize the run-time concepts, we focus on choosing features required to explain the semantics of an object model. Since an object model describes the static view of a system, we do not discuss the concepts related to timing and order of operations.

In this section we formally define the run-time concepts, including *object*, *link*, and *attribute value*. Moreover, we further formalize the semantics of the design-time concepts by specifying the relationships between the design-time and run-time concepts.

4.3.1 Fundamental Concepts

An *object* is defined as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. *Link* is a physical or conceptual connection between objects. We introduce a given set,

$$[INSTANCE]$$

and two global subsets, one each for objects and links,

$$OBJECT : \mathbb{P}\, INSTANCE$$
$$LINK : \mathbb{P}\, INSTANCE$$
$$\langle OBJECT, LINK \rangle \text{ partition } INSTANCE$$

The formalization of object and link is parallel to that of class and association. We define both objects and links to be elements of *INSTANCE* because a link can be modeled as an object in certain cases (see Section 4.1.1 and [RBP+91]: p.33).

It should be noted that we define *INSTANCE* and *ATOMIC_VALUE* as two different given sets. This guarantees that an attribute is a pure data value, not an object (see [RBP+91]: p.23).

A basic run-time system consists of

- a set of instances,
- a set of objects, and
- a set of links that connect the objects.

OMRTData

$$instance : \mathbb{P}\, INSTANCE$$
$$object : \mathbb{P}\, OBJECT$$
$$link : \mathbb{P}\, LINK$$

$$\langle object, link \rangle \text{ partition } instance$$

- Objects and links are instances of the run-time system.

4.3.2 Entities and Their Instances

The following schema describes the connection between the entities (i.e., classes and associations) in an object model and the instances (i.e., objects and links) in a run-time system. We define

- function *directInstanceOf* relating each entity to its direct instances, and
- relation *hasInstance* relating each class to its instances, which include its direct instances and the instances of its subclasses if any.

In a generalization relationship, an abstract class does not have direct instances but its descendent classes have. The abstract class, however, has instances which are the instances of its descendent classes (see [RBP+91]: p.61). This property of the abstract class is formalized via *hasInstance*.

OMRTInstance

$$
\begin{array}{l}
OMDTModel \\
OMRTData \\
directInstanceOf : INSTANCE \pfun ENTITY \\
hasInstance : CLASS \rel OBJECT \\
\hline
directInstanceOf \in (object \fun concreteClass) \cup (link \fun association) \\
\dom hasInstance \subseteq class \land \ran hasInstance = object \\
\forall o : object;\ c : class \mid directInstanceOf(o) = c \bullet o \in hasInstance(\!| \{c\} |\!) \\
\forall super, sub : class \mid sub \mapsto super \in isDescendentOf \bullet \\
\qquad hasInstance(\!| \{sub\} |\!) \subseteq hasInstance(\!| \{super\} |\!)
\end{array}
$$

- Every object is a direct instance of some concrete class, and every link is a direct instance of some association.
- Every object is a (direct or indirect) instance of one or more classes.
- If an object is a direct instance of a class, then it is an instance of the class.
- In an inheritance hierarchy, an instance of a subclass is simultaneously an instance of all its ancestor classes.

4.3.3 Attributes and Their Values

An *attribute value* is a data value held by an object. Schema *OMRTAttributeValue* defines a function indicating the value of each attribute in each object.

$$
\begin{array}{|l}
\textit{OMRTAttributeValue} \\
\hline
\textit{OMRTInstance} \\
\textit{holdValue} : \textit{INSTANCE} \pfun (\textit{ATTRIBUTE} \pfun \textit{ATOMIC\_VALUE}) \\
\hline
\dom \textit{holdValue} = \dom((\textit{directInstanceOf} \fatsemi \textit{hasFeature}) \rhd \textit{attribute}) \\
\forall\, i : \dom \textit{holdValue} \bullet \\
\quad \dom(\textit{holdValue}(i)) = ((\textit{directInstanceOf} \fatsemi \textit{hasFeature}) \rhd \textit{attribute})(\!|\, \{i\} \,|\!) \\
\forall\, i : \textit{instance};\ a : \textit{attribute};\ v : \textit{ATOMIC\_VALUE} \mid \\
\quad v = \textit{holdValue}(i)(a) \bullet v \in (\textit{valueRange}(\textit{directInstanceOf}(i)))(\!|\, \{a\} \,|\!) \\
\end{array}
$$

- The domain of function *holdValue* is a set of instances. They are direct instances of the entities that have attributes.
- Each attribute of an entity has a value for each instance of the entity.
- The attribute value is within the value range for the attribute.

4.3.4 Associations and Their Links

Schema *OMRTLink* defines

- a function indicating player instances of each role in each link, and
- a relation associating each link with the objects it connects.

OMRTLink

$$
\begin{array}{l}
OMRTInstance \\
roleOfLink : LINK \pfun (ROLE \pfun OBJECT) \\
connection : LINK \rel OBJECT \\
\hline
\dom roleOfLink = link \\
\forall\, l : link \bullet \\
\quad \dom(roleOfLink(l)) = ((directInstanceOf \fatsemi hasFeature) \rhd role)(\!|\, \{l\} \,|\!) \\
\forall\, l : link;\ r : role;\ o : object \mid \\
\quad o = roleOfLink(l)(r) \bullet o \in hasInstance(\!|\, \{playerOfRole(r)\} \,|\!) \\
connection = \{l : link;\ o : object;\ r : role \mid roleOfLink(l)(r) = o \bullet l \mapsto o\}
\end{array}
$$

- The domain of function *roleOfLink* is the set of links in the object model.
- Each role of each link has a role instance as the player.
- These role instances are the instances of the class that plays the role.
- Function *connection* associates each link with the objects it connects.

Schema *OMRTAssociationObject* further formalizes the semantics of function *modeledAs* in Schema *OMAssociationClass* defined in Section 4.1.4. The schema defines a function mapping each link to the association object that is used to model the link.

$$
\begin{array}{|l}
\hline
\textit{OMRTAssociationObject} \\
\textit{OMRTLink} \\
\textit{rtModeledAs} : \textit{LINK} \rightarrowtail\!\!\!\!\!\!\rightarrow \textit{OBJECT} \\
\hline
\operatorname{dom} \textit{rtModeledAs} = \operatorname{dom}(\textit{directInstanceOf} \rhd \operatorname{dom} \textit{modeledAs}) \\
\operatorname{ran} \textit{rtModeledAs} = \textit{hasInstance}(\!|\ \textit{associationClass}\ |\!) \\
\forall\, l : \textit{link};\ o : \textit{object} \mid \textit{rtModeledAs}(l) = o \bullet \\
\qquad o \in \textit{hasInstance}(\!|\ \{\textit{modeledAs}(\textit{directInstanceOf}(l))\}\ |\!) \\
\hline
\end{array}
$$

- The domain of function *rtModeledAs* is a set of links, which are the instances of the associations that are modeled as classes.
- The range of function *rtModeledAs* is a set of objects, which are the instances of the classes that are used to model the associations.
- If an association is modeled as a class, then each link of the association corresponds to an object of the class.

4.3.5 Roles and Their Player Instances

Each object that is connected to others via a link plays a *role* in the link.

$$
\begin{array}{|l}
\hline
\textit{OMRTRole} \\
\textit{OMRTInstance} \\
\textit{OMRTLink} \\
\textit{instanceOfRole} : \textit{ROLE} \leftrightarrow \textit{OBJECT} \\
\hline
\textit{instanceOfRole} = \{l : \textit{link};\ r : \textit{role};\ o : \textit{object} \mid \\
\qquad\qquad o = \textit{roleOfLink}(l)(r) \bullet r \mapsto o\} \\
\hline
\end{array}
$$

The relation defined in the following schema also deals with the relationship between roles and their instances. Each role corresponds to a set of object

sets. Each such object set is defined as a set of instances of the role (in certain association) when the instances that can appear in the remaining roles (of the same association) are fixed. Thus, this gives the semantics of multiplicity of role.

OMRTRoleMultiplicity

$$
\begin{array}{l}
OMRTRole \\
mulOfRole : ROLE \leftrightarrow \mathbb{P}\, OBJECT \\
\hline
\mathrm{dom}\, mulOfRole = \mathrm{dom}\, instanceOfRole \\
\forall\, r : \mathrm{dom}\, mulOfRole \bullet \bigcup(mulOfRole(\!|\, \{r\} \,|\!)) = instanceOfRole(\!|\, \{r\} \,|\!) \\
\forall\, r : \mathrm{dom}\, mulOfRole;\ os : \mathbb{P}\, object \mid \\
\qquad r \mapsto os \in mulOfRole \bullet \# os \in multiplicityOfRole(\!|\, \{r\} \,|\!)
\end{array}
$$

Schema *OMRTOrderedRole* gives semantics of ordered roles (see Section 4.1.4) in terms of run-time model.

OMRTOrderedRole

$$
\begin{array}{l}
OMRTRoleMultiplicity \\
ordering : ROLE \leftrightarrow \mathrm{iseq}\, OBJECT \\
\hline
\mathrm{dom}\, ordering = orderedRole \\
\forall\, r : orderedRole;\ os : \mathbb{P}\, object \mid r \mapsto os \in mulOfRole \bullet \\
\qquad (\exists_1\, os' : \mathrm{iseq}\, object \mid os = \mathrm{ran}\, os' \bullet r \mapsto os' \in ordering)
\end{array}
$$

- Each ordered role corresponds to a sequence of objects, representing certain order among the objects.
- These objects are the instances of the player of the ordered role.

4.3.6 Existence Dependence between Assemblies and Components

Schema *OMRTDependence* gives semantics of dependency between objects in an aggregation association (see Section 4.1.5) in a run-time model.

OMRTDependence

OMRTInstance
OMRTLink
OMRTRole

$$
\begin{array}{l}
\forall\, a : aggregation;\ r1, r2 : role;\ o : object \mid \\
\quad (child^{\sim} \fatsemi collectedIn)(r1) = a \land \\
\quad (parent^{\sim} \fatsemi collectedIn)(r2) = a \land \\
\quad playerOfRole(r1) \mapsto playerOfRole(r2) \in existenceDependOn \land \\
\quad o \in instanceOfRole(\!|\ \{r1\}\ |\!) \bullet \\
\qquad (\exists\, l : directInstanceOf^{\sim}(\!|\ \{parent^{\sim}(r2)\}\ |\!); \\
\qquad o' : instanceOfRole(\!|\ \{r2\}\ |\!) \bullet \\
\qquad\quad o = roleOfLink(l)(r1) \land \\
\qquad\quad o' = roleOfLink(l)(r2))
\end{array}
$$

- $r1$ plays the role of one of the components and $r2$ plays the role of the assembly of aggregation a. The existence of $r1$ depends on the existence of $r2$. If there is an instance o of $r1$ in a run-time object model, then there must exist an instance o' of $r2$, and o and o' are connected by an aggregation link.

4.3.7 Attribute Propagation in Aggregation

If two instances are in an aggregation relationship and certain attributes propagate from the assembly to the component, then the component instance and the assembly instance have the same values for the attributes.

OMRTPropagation

OMRTAttributeValue

$$\forall\, agg : \mathrm{dom}\, featurePropagate \bullet$$
$$(\forall\, a : featurePropagate(\!|\, \{agg\} \,|\!) \cap attribute;$$
$$assei : hasInstance(\!|\, \{single(agg)\} \,|\!);$$
$$compi : hasInstance(\!|\, group(\!|\, \{agg\} \,|\!) \,|\!) \bullet$$
$$holdValue(assei)(a) = holdValue(compi)(a))$$

4.3.8 Disjoint and Overlapping Generalizations

The following schema gives the semantics of disjoint generalizations and overlapping generalizations.

OMRTGeneralizationType

OMDTModel

OMRTInstance

$$\forall\, sub1, sub2 : class;\ dg : disjointGen \mid \{sub1, sub2\} \subseteq group(\!|\, \{dg\} \,|\!) \bullet$$
$$hasInstance(\!|\, \{sub1\} \,|\!) \cap hasInstance(\!|\, \{sub2\} \,|\!) = \emptyset$$

- The subclasses of a disjoint generalization do not have common instances; the subclasses of an overlapping generalization may have common instances.

4.3.9 Candidate Keys

The formal definition and some characteristics of candidate keys have been discussed in Section 4.2.8. The following schema specifies that a candidate key of an entity uniquely identifies an instance of the entity.

OMRTCandidateKey

OMRTAttributeValue

OMRTLink

$$
\begin{array}{l}
\forall\, c : keyedEntity \cap class;\ o1, o2 : object;\ k : key \mid \\
\quad \{o1, o2\} \subseteq hasInstance(\!|\ \{c\}\ |\!) \wedge \\
\quad isKeyOf(k) = c \bullet \\
\qquad hasKeyElement(\!|\ \{k\}\ |\!) \lhd holdValue(o1) = \\
\qquad hasKeyElement(\!|\ \{k\}\ |\!) \lhd holdValue(o2) \Leftrightarrow o1 = o2 \\
\\
\forall\, a : keyedEntity \cap association;\ l1, l2 : link;\ k : key \mid \\
\quad directInstanceOf(l1) = directInstanceOf(l2) = a \wedge \\
\quad isKeyOf(k) = a \bullet \\
\qquad hasKeyElement(\!|\ \{k\}\ |\!) \lhd roleOfLink(l1) = \\
\qquad hasKeyElement(\!|\ \{k\}\ |\!) \lhd roleOfLink(l2) \Leftrightarrow l1 = l2
\end{array}
$$

- Given a candidate key and a set of objects of a class, the values of the attributes, which make up the key, of one object are different from those of other objects.
- Given a candidate key and a set of links of an association, the object instances of the roles, which make up the key, of one link are different from those of other links.

4.3.10 Object Model with Run-Time Concepts

Schema *OMRunTime* defines the object model involving the run-time concepts.

$$
\begin{aligned}
&OMRTModel \mathrel{\widehat=} \\
&\quad OMRTData \land \\
&\quad OMRTInstance \land \\
&\quad OMRTAttributeValue \land \\
&\quad OMRTAssociationObject \land \\
&\quad OMRTLink \land \\
&\quad OMRTRole \land \\
&\quad OMRTOrderedRole \land \\
&\quad OMRTDependence \land \\
&\quad OMRTPropagation \land \\
&\quad OMRTGeneralizationType \land \\
&\quad OMRTCandidateKey
\end{aligned}
$$

4.4 Object Model

So far, we have discussed the object model with the design-time concepts and the run-time concepts. Putting the semantics for all these concepts together, we obtain the theory for the object model.

$$OM \mathrel{\widehat=} OMDTModel \land OMRTModel$$

Chapter 5

Formalization of SMOOA's Information Model

This chapter presents a formalization of the information model of Shlaer-Mellor object-oriented analysis (SMOOA) [SM88, SM92]. The product of the formalization is the design theory of the information model described in Z notation.

SMOOA is described in two books, one on information modeling [SM88] and one on state modeling and process modeling [SM92]. The method provides three models with the corresponding representations to describe entities in a problem domain and how they interact with each other.

> **Information model**. An information model is used to identify the conceptual entities, or *objects*, their characteristics, or *attributes*, and the *relationships* between these entities in the problem domain under analysis. There is one information model for the domain; however, it can be partitioned into multiple smaller information models, each of which is assigned to a separate subsystem. The graphic notation for the information model is based on entity-relationship diagrams. A complete description or definition of each object, attribute, and relationship must be prepared as documen-

tation for the graphic model.

State model. In SMOOA, each object or relationship has a life-cycle — the behavior over time. A state model is concerned with this dynamic behavior. A set of state models, each of which is used to depict the behavior of an object or a relationship, communicate with one another by means of events, and are organized in layers to make the system communication orderly. The representation techniques used in this modeling activity include state transition diagrams (STDs), state transition tables (STTs), STD action description lists, event lists, and object communication models.

Process model. The process models capture the functional components of a system. They specify the processing contained in the actions of the state models in the form of action data flow diagrams (ADFDs).

This chapter focuses on the information modeling aspects of SMOOA, which resemble conventional relational database design extended with a notion of inheritance. A detailed discussion on the information model is presented in [SM88], and is reviewed in Chapter 2 of [SM92]. Information modeling is the most basic among the three kinds of modelings in SMOOA. The information model is used at the beginning of the analysis phase to capture the information of the problem domain under study.

5.1 Design-Time Concepts

Design-time concepts describe the possible patterns of run-time applications. These concepts include object, relationship, attribute, role, and subtype-supertype construct. This section gives the formal definition of the design-time concepts, and formalizes some design rules applied to these concepts.

5.1.1 Fundamental Concepts

Two fundamental concepts in SMOOA, *object* and *relationship*, are discussed in detail in Chapter 3 and 5 of [SM88]. SMOOA uses the word *object* where other methods use *class*. An *object* is an abstraction of a set of real-world things, which have the same characteristics, and subject to and conform to the same rules. A *relationship* is the abstraction of a set of associations that hold systematically between different objects. One object in a system usually has one or more relationships with other objects. We introduce a given set,

$$[ENTITY]$$

and two global sets, which partition the given set, one each for objects and relationships,

$$OBJECT : \mathbb{P}\, ENTITY$$
$$RELATIONSHIP : \mathbb{P}\, ENTITY$$
$$\langle OBJECT, RELATIONSHIP \rangle \text{ partition } ENTITY$$

Another important concept is *attribute*. An *attribute* is the abstraction of a *single* characteristic possessed by all the instances of an object.

Although SMOOA does not include the concept "role" when discussing relationships, it is useful to adopt it in the formalization. Each object that participates in a relationship plays a particular role in the relationship. Each relationship has roles, each of which is played by a participating object. The role name signifies the role that an instance of a participating object plays in each relationship instance. Under some circumstances, the same object participates more than once in a relationship in different roles, so it is essential to distinguish the meaning of each participation by roles.

We introduce a given set,

$$[FEATURE]$$

and two disjoint subsets,

$$
\begin{array}{|l}
ATTRIBUTE : \mathbb{P}\, FEATURE \\
ROLE : \mathbb{P}\, FEATURE \\
\hline
\langle ATTRIBUTE, ROLE \rangle \text{ partition } FEATURE
\end{array}
$$

for attributes and roles respectively.

The global given sets and subsets we introduced so far represent the design categories; they are the sets of all possible elements of all possible information models. We will need to define the sets that represent a particular information model. An information model consists of the following elements that are identified by analyzing the problem domain:

- a set of entities,
- a set of objects,
- a set of relationships between the objects,
- a set of features,
- a set of attributes of the objects, and
- a set of roles involved in the relationships.

IMDTData

$entity : \mathbb{P}\, ENTITY$

$object : \mathbb{P}\, OBJECT$

$relationship : \mathbb{P}\, RELATIONSHIP$

$feature : \mathbb{P}\, FEATURE$

$attribute : \mathbb{P}\, ATTRIBUTE$

$role : \mathbb{P}\, ROLE$

$object \neq \emptyset$

$\langle object, relationship \rangle$ partition $entity$

$\langle attribute, role \rangle$ partition $feature$

- The information model must contain some objects.
- Objects and relationships are subsets of the entity set.
- Attributes and roles are subsets of the feature set.

5.1.2 Features of Entities

A basic property of attributes and roles is that they are always associated with some particular objects or relationships. Thus, given a "feature", we can discover the particular "entity" in which it is defined.

Attributes

In the first stage of the system development, that is in the information model, the major information about an object is recorded in a set of attributes. Chapter 4 of [SM88] gives detailed discussion on attribute.

Schema *IMAttributeBasic* defines

- a relation that relates each attribute to the object in which the attribute is defined, and

- a relation that relates each object to the attributes the object may have.

IMAttributeBasic

$$
\begin{array}{l}
IMDTData \\
attrDefinedIn : ATTRIBUTE \leftrightarrow OBJECT \\
hasAttribute : OBJECT \leftrightarrow ATTRIBUTE \\
\hline
\mathrm{dom}\ attrDefinedIn = attribute \\
\mathrm{ran}\ attrDefinedIn = object \\
hasAttribute = attrDefinedIn^{\sim}
\end{array}
$$

- Every attribute belongs to one or more objects.
- Every object has one or more attributes.
- Function *hasAttribute* is the inverse of function *attrDefinedIn*.

Types of Attributes

The attributes fall into three categories (see Section 4.3 in [SM88]):

1. Descriptive attributes: intrinsic characteristics of the object.
2. Naming attributes: arbitrary names and labels.
3. Referential attributes: facts that tie an instance of one object to an instance of another object.

IMAttributeType

$$
\begin{array}{l}
IMAttributeBasic \\
descriptiveAttr : \mathbb{P}\ ATTRIBUTE \\
namingAttr : \mathbb{P}\ ATTRIBUTE \\
referentialAttr : \mathbb{P}\ ATTRIBUTE \\
\hline
\langle descriptiveAttr, namingAttr \rangle \ \mathsf{partition}\ attribute \\
referentialAttr \subseteq namingAttr
\end{array}
$$

- An attribute can be descriptive or naming attribute.
- A referential attribute is a naming attribute.

Domains

For each attribute, there is a range of legal values that the attribute can take on. This range is called the *domain* of the attribute. SMOOA suggests several ways to define these domains (see [SM88]: pp.37-40). All these domains for the attributes are specified within the scope of objects.

The values an attribute can take on, are of a certain type, such as integers, natural numbers, characters, and strings, which could be expressed as some given set in Z notation. Since it is impossible to enumerate in the theory all real-world types that could be used in a design model, a simpler way to represent all these types is to put them together in a single given set, for example $ATOMIC\_VALUE$, in the design theory. Each type can be viewed as a subset of the given set. We hereinbelow introduce a given set

$$[ATOMIC\_VALUE]$$

which contains elements of all possible types that the attribute values may belong to.

Thereafter, a function *domainSpec* is defined that maps each attribute, of an object, to a set of legal values that the attribute can take on:

IMDomainSpecification

$$IMAttributeBasic$$
$$domainSpec : OBJECT \pfun (ATTRIBUTE \rel ATOMIC\_VALUE)$$

$$\mathrm{dom}\ domainSpec = object$$
$$\forall\ obj : object \bullet \mathrm{dom}(domainSpec(obj)) = hasAttribute(\!|\ \{obj\}\ |\!)$$

- Domain specifications must be provided for the attributes of every object.

- Every attribute of every object must have a set of legal values as its domain.

Identifiers

An *identifier* for an object is a set of one or more attributes whose values uniquely distinguish each instance of the object (see Section 4.4 in [SM88]). In some cases, the same set of attributes could serve as identifiers in more than one object. For instance, in a subtype-supertype construct, which will be discussed in Section 5.1.5, the supertype object and its subtype objects have the same set of attributes as their identifiers. We introduce a unique identity for each identifier specification. Identifiers are formally defined as elements of the given set

$[IDENTIFIER]$

Schema *IMIdentifierBasic* defines

- a set of identifiers, and
- a relation that relates each identifier to the attributes the identifier contains.

IMIdentifierBasic

$$
\begin{array}{l}
IMAttributeBasic \\
identifier : \mathbb{P}\, IDENTIFIER \\
hasIdElement : IDENTIFIER \leftrightarrow ATTRIBUTE \\
\hline
\mathrm{dom}\, hasIdElement = identifier \\
\mathrm{ran}\, hasIdElement \subseteq attribute
\end{array}
$$

- Each identifier is composed of one or more attributes.
- An attribute can participate in constructing one or more identifiers.

Every object can have several identifiers. Only one of them becomes the *preferred identifier* of the object that is used as the identifier of the object in a design model. The following schema defines

- function *isIdOf* mapping from each identifier to the object it belongs to, and
- function *hasPreferredId* mapping from each object to its preferred identifier.

IMIdentifier

$$
\begin{array}{l}
IMIdentifierBasic \\
isIdOf : IDENTIFIER \nrightarrow OBJECT \\
hasPreferredId : OBJECT \rightarrowtail IDENTIFIER \\
\hline
isIdOf \in identifier \twoheadrightarrow object \\
\forall\, id : identifier \bullet hasIdElement(\!|\, \{id\} \,|\!) \subseteq hasAttribute(\!|\, \{isIdOf(id)\} \,|\!) \\
\forall\, id1, id2 : identifier \mid \\
\quad isIdOf(id1) = isIdOf(id2) \land id1 \neq id2 \bullet \\
\quad\quad \neg\, hasIdElement(\!|\, \{id1\} \,|\!) \subseteq hasIdElement(\!|\, \{id2\} \,|\!) \land \\
\quad\quad \neg\, hasIdElement(\!|\, \{id2\} \,|\!) \subseteq hasIdElement(\!|\, \{id1\} \,|\!) \\
hasPreferredId \in object \rightarrowtail identifier \\
\forall\, obj : object \bullet isIdOf(hasPreferredId(obj)) = obj
\end{array}
$$

- Every identifier belongs to one object, and every object must have at least one identifier.
- An identifier of an object contains only the attributes of this object, not the attributes of others.
- If an object has more than one identifier, the attribute combination of one identifier is different from that of other identifiers.

- Each object has a unique preferred identifier.
- An object's preferred identifier is one of its identifiers.

Combining the schemas about attributes, we obtain the following schema that specifies the properties of attributes:

$$\begin{aligned} IMAttribute \mathrel{\widehat=} & \; IMAttributeType \land \\ & IMDomainSpecification \land \\ & IMIdentifier \end{aligned}$$

Roles

The concept *role* is important for the formalization of relationships. Each relationship involves several roles, and each role is played by an object.

Schema *IMRoleBasic* defines two relations indicating the relationship in which each role is involved.

IMRoleBasic

$$IMDTData$$
$$roleDefinedIn : ROLE \pfun RELATIONSHIP$$
$$hasRole : RELATIONSHIP \rel ROLE$$

$$roleDefinedIn \in role \tfun relationship$$
$$hasRole = roleDefinedIn^{\sim}$$
$$\forall rel : relationship \bullet \#(hasRole(\!| \{rel\} |\!)) \geq 2$$

- Every role belongs to exactly one relationship.
- Function *hasRole* is inverse of function *roleDefinedIn*.
- Every relationship must have two or more roles, i.e., a relationship may be binary, ternary, or higher order.

The multiplicity and conditionality of a relationship are called the form of the relationship. SMOOA does not give the definition of multiplicity; however, it classifies binary relationships into three fundamental forms with respect to multiplicity. They are one-to-one, one-to-many, and many-to-many relationships:

- A one-to-one relationship exists when a single instance of an object is associated with a single instance of another.
- A one-to-many relationship exists when a single instance of an object is associated with one or more instances of another, and each instance of the second object is associated with just one instance of the first.
- A many-to-many relationship exists when a single instance of an object is associated with one or more instances of another, and each instance of the second object is associated with one or more instances of the first.

If every instance of both objects is required to participate in the relationship, the relationship is *unconditional*. If there can be instances of the objects that do not participate, the relationship is *conditional*.

Since they are actually concerned with the property of the objects that play roles in relationships, multiplicity and conditionality are associated with roles in the formalization. We define them as multiplicity of roles. The multiplicity of a role at run-time specifies the number of instances of one object that appear in a given role of a relationship when the instances that can appear in the remaining roles are fixed. This number may be "0" if the relationship is conditional on that role.

Schema *IMRoleProperty* defines

- a function that specifies the player of each role, and
- a relation that specifies the multiplicity of each role. In the formalization, we define the multiplicity of a role as a (possibly infinite) subset of the

non-negative integers. For the role r of a relationship, each number n in the set $multiplicityOfRole(\!|\ \{r\}\ |\!)$ indicates that the multiplicity of r can be n.

IMRoleProperty

$$IMDTData$$
$$playerOfRole : ROLE \pfun OBJECT$$
$$multiplicityOfRole : ROLE \rel \mathbb{N}$$

$$playerOfRole \in role \fun object$$
$$\mathrm{dom}\ multiplicityOfRole = role$$

- Each role has a player, which is an object. Each object may play one or more roles.
- Each role has the predefined multiplicity.

According to their multiplicity, roles can be classified into four types. We define them as *single*, *conditional single*, *multiple*, and *conditional multiple*:

IMRoleType

$IMRoleProperty$

$single : \mathbb{P}\, ROLE$

$conditionSingle : \mathbb{P}\, ROLE$

$multiple : \mathbb{P}\, ROLE$

$conditionMultiple : \mathbb{P}\, ROLE$

$\langle single, conditionSingle, multiple, conditionMultiple \rangle$ partition $role$

$\forall\, s : single \bullet multiplicityOfRole(\!|\, \{s\}\, |\!) = \{1\}$

$\forall\, cs : conditionSingle \bullet multiplicityOfRole(\!|\, \{cs\}\, |\!) = \{0, 1\}$

$\forall\, m : multiple \bullet 0 \notin multiplicityOfRole(\!|\, \{m\}\, |\!) \land (\exists\, n : multiplicityOfRole(\!|\, \{m\}\, |\!) \bullet n > 1)$

$\forall\, cm : conditionMultiple \bullet 0 \in multiplicityOfRole(\!|\, \{cm\}\, |\!) \land (\exists\, n : multiplicityOfRole(\!|\, \{cm\}\, |\!) \bullet n > 1)$

- Single, conditional single, multiple, and conditional multiple roles are special types of roles with respect to multiplicity.
- Given a role of a relationship, when the instances of other roles in the same relationship are fixed in a run-time model,
  - if the role is single, then there is exactly one instance playing the role;
  - if the role is conditional single, then there is one instance, or none at all, playing the role;
  - if the role is multiple, then there are one or more instances playing the role;
  - if the role is conditional multiple, then there are one or more instances, or none at all, playing the role.

For a multiple or conditional multiple role, there must be the cases in which two or more instances play the role.

Schema *IMRTRoleType* in Section 5.2.5 explains the semantics of multiplicity of roles in binary relationships, in terms of run-time model.

Combining the schemas about roles, we obtain schema *IMRole*:

$$IMRole \mathrel{\widehat=} IMRoleBasic \land IMRoleProperty \land IMRoleType$$

Features

Relation *hasFeature* in the following schema expresses the relationship between features and entities in the information model.

IMFeature

$IMAttribute$

$IMRole$

$hasFeature : ENTITY \leftrightarrow FEATURE$

$hasFeature = hasAttribute \cup hasRole$

5.1.3 Binary Relationships

SMOOA gives a detailed discussion on coping with binary relationships in an information model, but puts less emphasis on relationships with higher order (see Chapter 5 of [SM88]).

Schema *IMBinaryRelationship* defines a set of binary relationships in the information model.

$$
\begin{array}{|l}
\textit{IMBinaryRelationship} \\
\hline
\textit{IMRole} \\
\textit{binaryRelationship} : \mathbb{P}\, \textit{RELATIONSHIP} \\
\hline
\textit{binaryRelationship} \subseteq \textit{relationship} \\
\forall\, \textit{rel} : \textit{binaryRelationship} \bullet \#(\textit{hasRole}(\!|\, \{\textit{rel}\}\, |\!|)) = 2 \\
\hline
\end{array}
$$

- The binary relationships are a subset of the relationships in the information model.
- A binary relationship has exactly two roles.

Figure 5.1 summarizes all possible forms of the binary relationships described in SMOOA (see Figure on p.59 of [SM88]). Schema *IMBinaryRelationshipType* defines the semantics for these forms of binary relationships.

IMBinaryRelationshipType

$$
\begin{array}{l}
IMBinaryRelationship \\
R\_11, R\_1M, R\_MM : \mathbb{P}\, RELATIONSHIP \\
R\_11c, R\_1cM, R\_1Mc, R\_MMc : \mathbb{P}\, RELATIONSHIP \\
R\_1c1c, R\_1cMc, R\_McMc : \mathbb{P}\, RELATIONSHIP \\
\hline
\langle R\_11, R\_1M, R\_MM, R\_11c, R\_1cM, R\_1Mc, \\
\quad R\_MMc, R\_1c1c, R\_1cMc, R\_McMc \rangle \text{ partition } binaryRelationship \\
\forall\, rel : R\_11 \bullet \\
\quad (\exists\, r1, r2 : single \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_1M \bullet \\
\quad (\exists\, r1 : single;\ r2 : multiple \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_MM \bullet \\
\quad (\exists\, r1, r2 : multiple \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_11c \bullet \\
\quad (\exists\, r1 : single;\ r2 : conditionSingle \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_1cM \bullet \\
\quad (\exists\, r1 : conditionSingle;\ r2 : multiple \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_1Mc \bullet \\
\quad (\exists\, r1 : single;\ r2 : conditionMultiple \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_MMc \bullet \\
\quad (\exists\, r1 : multiple;\ r2 : conditionMultiple \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_1c1c \bullet \\
\quad (\exists\, r1, r2 : conditionSingle \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_1cMc \bullet \\
\quad (\exists\, r1 : conditionSingle;\ r2 : conditionMultiple \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\}) \\
\forall\, rel : R\_McMc \bullet \\
\quad (\exists\, r1, r2 : conditionMultiple \bullet hasRole(\!| \{rel\} |\!) = \{r1, r2\})
\end{array}
$$

- There are ten forms of binary relationships.

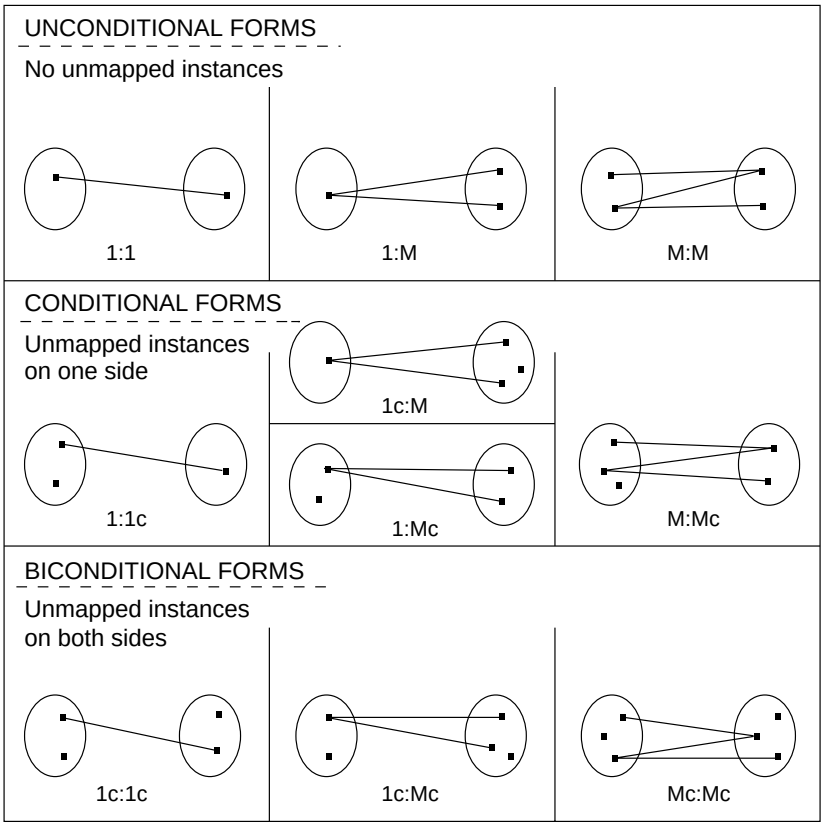


Figure 5.1: Forms of binary relationships

- In a one-to-one unconditional relationship, denoted by R_11, a given instance of role $r1$ is associated with one and only one instance of role $r2$. Furthermore, every instance of role $r2$ must have an instance of role $r1$ so associated.

- In an one-to-many unconditional relationship, denoted by R_1M, a single instance of role $r1$ is associated with one or more instances of role $r2$. Every instance of role $r2$ is associated with exactly one instance of role $r1$.

- In a many-to-many unconditional relationship, denoted by R_MM, every

instance of role $r1$ is associated with one or more instances of role $r2$, and every instance of role $r2$ is associated with one or more instances of role $r1$.

- A one-to-one conditional relationship, denoted by R_11c, is just like a one-to-one unconditional relationship, except that not all instances of role $r1$ need to participate in the relationship.

- A one-to-many conditional relationship (on the "one" side), denoted by R_1cM, has the following properties:

  - Each instance of role $r1$ is associated with one or more of instances of role $r2$. Every instance of role $r1$ participates in the relationship.
  - An instance of role $r2$ is associated with one or zero instance of role $r1$; that is, not all instances of role $r2$ participate.

- A one-to-many conditional relationship (on the "many" side), denoted by R_1Mc, is conditional on the "many" side: each instance of role $r1$ is associated with zero or more instances of role $r2$, while each instance of role $r2$ is associated with exactly one instance of role $r1$.

- A many-to-many conditional relationship, denoted by R_MMc, is just like the ordinary unconditional (R_MM) relationship, except that instances of role $r1$ can exist which do not participate in the relationship.

- In a one-to-one biconditional relationship, denoted by R_1c1c, one instance of role $r1$ is associated with zero or one instance of role $r2$. Furthermore, one instance of role $r2$ is associated with zero or one instance of role $r1$.

- A one-to-many biconditional relationship, denoted by R_1cMc, is a form of the basic one-to-many relationship, but there can be instances of both types of objects which do not participate in the relationship.

- A many-to-many biconditional relationship, denoted by *R_McMc*, resembles the basic many-to-many unconditional form, except there can be instances of both objects which do not participate in the relationship.

5.1.4 Composition of Relationships

Some relationships come about as a necessary consequence of the existence of other relationships (see [SM92]: pp.27-28). Such a relationship is said to have been formed by composition (as in composition of functions in mathematics). Let

$$[FUNCTION]$$

represent the functions that determine how composition relationships are composed from other relationships.

Schema *IMCompositionRelationship* defines

- a set of composition relationships in the information model, and
- a function that associates a composition relationship to the relationships that compose it according to certain function.

IMCompositionRelationship

$$IMDTData$$
$$comRelationship : \mathbb{P}\, RELATIONSHIP$$
$$composedBy : RELATIONSHIP \pfun \mathbb{P}\, RELATIONSHIP \times FUNCTION$$

$$\mathrm{dom}\, composedBy = comRelationship$$
$$\forall\, r : comRelationship \bullet first\,(composedBy(r)) \subset relationship$$

- Composition relationships are a subset of relationships.
- Each composition relationship is formed by a set of relationships.

5.1.5 Generalizations

When several objects in the information model are significantly similar, they can be modeled in a subtype-supertype construct. We call such a construct a *generalization*. Generalization is a distinguished feature of object orientation. It is a special kind of relationships between objects. Section 6.1 in [SM88] discusses the concept.

Generalizations

In a generalization construct, attributes that are common to all the subtype objects are placed in the supertype object. The subtype objects will also have additional attributes to support the more specialized abstractions represented by each subtype. Let

$$[GENERALIZATION]$$

represent all possible generalization constructs.

Each such construct can be obtained under a certain specialization principle. It is also possible to apply several different principles to one supertype object. Hence, a supertype object may be involved in more than one generalization.

Schema *IMGeneralizationBasic* defines

- a set of generalization constructs in the information model,
- a function from each generalization construct to the supertype object in the generalization.
- a relation from each generalization construct to the subtype objects in the generalization.

IMGeneralizationBasic

$$
\begin{array}{l}
IMDTData \\
generalization : \mathbb{P}\, GENERALIZATION \\
supertype : GENERALIZATION \nrightarrow OBJECT \\
subtype : GENERALIZATION \leftrightarrow OBJECT \\
\hline
supertype \in generalization \rightarrow object \\
\\
\mathrm{dom}\, subtype = generalization \land \mathrm{ran}\, subtype \subset object
\end{array}
$$

- Each generalization has a supertype object. An object may be involved as a supertype object in one or more generalizations.
- Each generalization has a set of subtype objects. An object may be involved as a subtype object in one or more generalizations.

Similar to formalizing OMT's object model (discussed in Chapter 4), we view a generalization as a set of binary relationships. Each such relationship associates the supertype object with one of the subtype objects:

IMGeneralizationRelationship

$$
\begin{array}{l}
IMBinaryRelationshipType \\
IMGeneralizationBasic \\
genRelationship : \mathbb{P}\, RELATIONSHIP \\
superRole : RELATIONSHIP \rightarrowtail ROLE \\
subRole : RELATIONSHIP \rightarrowtail ROLE \\
\hline
genRelationship \subseteq binaryRelationship \\
\\
superRole \in genRelationship \rightarrowtail single \\
\\
subRole \in genRelationship \rightarrowtail conditionSingle \\
\\
\forall\, rel : genRelationship \bullet hasRole(\!|\, \{rel\}\, |\!) = \{superRole(rel), subRole(rel)\}
\end{array}
$$

- A generalization relationship is a one-to-one conditional binary relationship. SMOOA requires that, in a generalization construct, if an instance for a subtype object is created, an instance for the supertype object must be created as well, and vice versa. Therefore, in a generalization relationship, each instance of the subtype object, *sub*, corresponds to an instance of the supertype object *super*; but an instance of *super* may correspond to an instance of *sub* or may correspond to an instance of other objects that are also subtype objects of *super* in the same generalization construct. Schema *IMRTGenInstance* in Section 5.2.4 formalizes this property in terms of run-time concepts.

- The role whose player is a supertype object is a single role.

- The role whose player is a subtype object is a conditional single role.

- Each generalization relationship contains a supertype object and a subtype object.

Schema *IMGeneralization* defines the mappings between the generalization relationships and the generalization constructs in which these relationships are involved.

$$
\begin{array}{|l}
\textit{IMGeneralization} \\
\hline
\textit{IMGeneralizationRelationship} \\
\textit{collectedIn} : \textit{RELATIONSHIP} \pfun \textit{GENERALIZATION} \\
\hline
\textit{collectedIn} \in \textit{genRelationship} \rightarrow\!\!\!\!\!\rightarrow \textit{generalization} \\
\forall\, \textit{rel} : \textit{genRelationship};\ \textit{gen} : \textit{generalization} \mid \\
\quad \textit{collectedIn}(\textit{rel}) = \textit{gen} \bullet \\
\qquad \textit{playerOfRole}(\textit{superRole}(\textit{rel})) = \textit{supertype}(\textit{gen}) \land \\
\qquad \textit{playerOfRole}(\textit{subRole}(\textit{rel})) \in \textit{subtype}(\!|\ \{\textit{gen}\}\ |\!) \\
\forall\, \textit{gen} : \textit{generalization} \bullet \\
\quad \#(\textit{subtype}(\!|\ \{\textit{gen}\}\ |\!)) = \#(\textit{collectedIn}^{\sim}(\!|\ \{\textit{gen}\}\ |\!)) \\
\hline
\end{array}
$$

- Each generalization relationship belongs to a generalization construct. A generalization construct may contain a set of generalization relationships.
- For a generalization relationship, the player of the super-role is the supertype object, and the player of the sub-role is one of the subtype objects of the generalization construct, in which the generalization relationship is involved.
- The number of generalization relationships in one generalization construct is the same as the number of subtype objects involved in the construct.

Inheritance and the Characteristics

A subtype object and the supertype object of a generalization construct are in a "is-kind-of" relationship, that is, the subtype object is a refined version of the supertype object. An object can be the subtype object of several supertype objects, each of which is in a different generalization construct. This is called "multiple inheritance". Each generalization construct can be

drawn on repeatedly in the same problem. All the objects that participate in subtype-supertype relationships in the information model make up an inheritance hierarchy. These concepts are formalized in schema *IMIsKindOf*, which defines

- a relation that associates each subtype object with all of its direct supertype object(s), no matter which generalization construct it is in, and
- a relation that associates each subtype object with all of its supertype objects, possibly through repeated generalization constructs.

IMInheritanceBasic

$$
\begin{array}{l}
IMFeature \\
IMGeneralization \\
isKindOf : OBJECT \leftrightarrow OBJECT \\
inherit : OBJECT \leftrightarrow OBJECT \\
\hline
isKindOf = \{gen : generalization;\ sub : object \mid \\
\qquad sub \in subtype(\!| \{gen\} |\!) \bullet sub \mapsto supertype(gen)\} \\
inherit = isKindOf^{+} \\
inherit \cap \mathrm{id}\, object = \emptyset \\
\forall\, gen : generalization;\ sub1, sub2 : object \mid \\
\qquad \{sub1, sub2\} \subseteq subtype(\!| \{gen\} |\!) \bullet \\
\qquad\quad \mathrm{dom}(inherit \rhd \{sub1\}) \cap \mathrm{dom}(inherit \rhd \{sub2\}) = \emptyset
\end{array}
$$

- *isKindOf* associates each subtype object with all of its supertype objects.
- *inherit* associates each subtype object with its ancestors.
- *inherit* should be a directed acyclic graph, which means that any object is not a subtype (or supertype) object of itself.

- In a generalization construct, any two subtype objects of a supertype object must not have common descendants. The allowance of such inheritance is incorrect in semantics. SMOOA requires that the properties of the subtype objects should be a partition of that of the supertype object, so any descendant of one subtype object could not have property of another subtype object.

As is explained in schema *IMGeneralizationRelationship*, there is a one-to-one mapping between the instances of the supertype object and the instances of the subtype objects in a given generalization. It is also known that, given an object, the values of its identifier uniquely identify each instance of the object. We therefore draw the conclusion that there is a one-to-one mapping between the values of the supertype preferred identifier and the values of the subtype preferred identifiers. Assuming the domains of the subtype preferred identifiers are disjoint[1], we define the domain of the supertype preferred identifier to be the union of the domains of the subtype preferred identifiers.

[1]When the domains of the subtype preferred identifiers are not disjoint, SMOOA provides two ways of modification (see [SM88]: p.67) in order to ensure the one-to-one mapping between the instances of the supertype object and the subtype objects. Here we formalize the first one. In this way, the supertype and subtype preferred identifiers consist of the same set of attributes, and the domains of subtype preferred identifiers are guaranteed disjoint.

IMAttributeInheritance

IMInheritanceBasic

$$\forall\, super, sub : object \mid$$
$$sub \mapsto super \in isKindOf \bullet$$
$$hasIdElement(\!| \{hasPreferredId(sub)\} |\!) =$$
$$hasIdElement(\!| \{hasPreferredId(super)\} |\!)$$

$$\forall\, gen : generalization;\ super : object;\ a : attribute \mid$$
$$supertype(gen) = super \wedge$$
$$a \in hasIdElement(\!| \{hasPreferredId(super)\} |\!) \bullet$$
$$(domainSpec(super))(\!| \{a\} |\!) =$$
$$\bigcup \{sub : subtype(\!| \{gen\} |\!) \bullet (domainSpec(sub))(\!| \{a\} |\!)\}$$

- The supertype object and its subtype objects have the same preferred identifier.
- Given a generalization construct, if an attribute is a part of the preferred identifier, then its domain in the subtype object is a subset of its domain in the supertype object. Further, the union of the domains of the attribute in all the subtype objects is the domain of the attribute in the supertype object.

The properties of the subtype-supertype constructs and the rules and constraints on these constructs are summarized as follows.

$$IMInheritance \mathrel{\widehat{=}} IMInheritanceBasic \wedge IMAttributeInheritance$$

5.1.6 Relationship Formalization

SMOOA indicates that the purpose of a relationship is to allow us to state which instances of one object are associated with instances of others. This is

accomplished by placing referential attributes in appropriate objects. When this has been done, the relationship is said to have been formalized in data. The Z schemas in this section specify the semantics for formalizing relationships by means of referential attributes.

Certain referential attribute(s) in an object always captures some relationships in the information model. The referential attributes in an object may capture several relationships; thus, an object can be used to formalize one or more relationships. We call the object that has referential attributes a *formalizer*. The formalizer of a relationship functions as a connector that connects itself to each participating object via referential attributes.

Schema *IMRelationshipFormalizingBasic* defines

- a set of formalizers in the information model, and
- a function from each relationship to its formalizer.

IMRelationshipFormalizingBasic

$$IMCompositionRelationship$$
$$IMInheritance$$
$$formalizer : \mathbb{P}\, OBJECT$$
$$formalizedBy : RELATIONSHIP \twoheadrightarrow OBJECT$$

$$formalizer = \mathrm{ran}(referentialAttr \lhd attrDefinedIn)$$
$$formalizedBy \in relationship \setminus comRelationship \twoheadrightarrow formalizer$$

- The formalizers in the information model are a set of objects that have referential attributes.
- Each relationship, except for composition relationships, is formalized by an object.

Within the scope of a formalizer, function *referencing* in the following schema defines a mapping from each relationship that is formalized by the formalizer to the referential attributes of the formalizer that capture the relationship. These referential attributes, called foreign keys, are actually attributes of the identifiers of other objects that participate in the relationship.

IMReferencing

$$IMRelationshipFormalizingBasic$$
$$referencing : OBJECT \pfun (RELATIONSHIP \rel ATTRIBUTE)$$

$$\dom referencing = formalizer$$

$$\forall obj : formalizer;\ rel : relationship \mid formalizedBy(rel) = obj \bullet$$
$$rel \in \dom(referencing(obj)) \land$$
$$(referencing(obj))(\!| \{rel\} |\!) \subseteq hasAttribute(\!| \{obj\} |\!) \cap referentialAttr$$

- The domain of function *referencing* is the formalizers in the information model.
- One or more referential attributes of a formalizer capture a relationship.

Players as Formalizers

The information model provides rules to formalize relationships. The assignments of referential attributes to objects in conditional and unconditional one-to-one and one-to-many binary relationships are formalized in schema *IMPlayerFormalizing*. The common characteristic of these relationships is that their formalizers are also their players. Generalization relationships are formalized differently from other binary relationships. We discuss this later in the section (see p.144).

IMPlayerFormalizing

IMReferencing

IMBinaryRelationshipType

$$
\begin{aligned}
&\forall\, rel : binaryRelationship \setminus (genRelationship \cup comRelationship);\ r1, r2 : role \mid \\
&\quad hasRole(\!|\, \{rel\} \,|\!) = \{r1, r2\} \land \\
&\quad formalizedBy(rel) = playerOfRole(r1) \bullet \\
&\qquad (referencing(playerOfRole(r1)))(\!|\, \{rel\} \,|\!) = \\
&\qquad\quad hasIdElement(\!|\, \{hasPreferredId(playerOfRole(r2))\} \,|\!) \land \\
&\qquad (rel \in R\_11 \Rightarrow r1 \in single \land r2 \in single) \land \\
&\qquad (rel \in R\_1M \Rightarrow r1 \in multiple \land r2 \in single) \land \\
&\qquad (rel \in R\_11c \Rightarrow r1 \in conditionSingle \land r2 \in single) \land \\
&\qquad (rel \in R\_1c1c \Rightarrow r1 \in conditionSingle \land r2 \in conditionSingle) \land \\
&\qquad (rel \in R\_1cM \Rightarrow r1 \in multiple \land r2 \in conditionSingle) \land \\
&\qquad (rel \in R\_1Mc \Rightarrow r1 \in conditionMultiple \land r2 \in single) \land \\
&\qquad (rel \in R\_1cMc \Rightarrow r1 \in conditionMultiple \land r2 \in conditionSingle)
\end{aligned}
$$

- To model an unconditional one-to-one relationship (R_11), we place the foreign key in either of the objects.
- To model an unconditional one-to-many relationship (R_1M), we take the identifier from the "one" side and make it as a foreign key in the "many" side.
- A one-to-one conditional relationship (R_11c) is modeled by adding the referential attributes to the object which always participates in the relationship.
- Modeling a one-to-one biconditional relationship (R_1c1c) is similar to modeling a one-to-one unconditional relationship.
- A one-to-many conditional (on the "one" side) relationship (R_1cM) can be modeled by adding the foreign key to the object on the "many" side

of the relationship.

- A one-to-many conditional (on the "many" side) relationship (R_1Mc) can be modeled by adding the foreign key to the "many" side.
- A one-to-many biconditional relationship (R_1cMc) can be modeled by adding a foreign key to the object on the "many" side.

Associative Objects

Relationships R_1c1c, R_MM, R_MMc, R_McMc, and higher-order relationships can be formalized in associative objects. SMOOA uses both associative objects and correlation tables for this purpose in [SM88]; however, it omits the concept of correlation table in [SM92]. In fact, a correlation table can be treated as a special form of associative objects. Here we only formalize the concept of associative object. Generally speaking, any form of relationships can be formalized in associative objects.

IMAssociativeObjectFormalizing

$$
\begin{array}{l}
IMReferencing \\
associativeObject : \mathbb{P}\ OBJECT \\
\hline
associativeObject \subseteq formalizer \\
\forall\ rel : \mathrm{dom}\ formalizedBy \setminus genRelationship;\ obj : associativeObject \mid \\
\quad formalizedBy(rel) = obj \bullet obj \notin playerOfRole(\!|\ hasRole(\!|\ \{rel\}\ |\!)\ |\!) \\
\forall\ rel : \mathrm{dom}\ formalizedBy \setminus genRelationship;\ obj : associativeObject \mid \\
\quad formalizedBy(rel) = obj \bullet \\
\qquad (referencing(obj))(\!|\ \{rel\}\ |\!) = \\
\qquad hasIdElement(\!|\ (hasRole \fatsemi playerOfRole \fatsemi hasPreferredId)(\!|\ \{rel\}\ |\!)\ |\!) \subseteq \\
\qquad hasIdElement(\!|\ \{hasPreferredId(obj)\}\ |\!)
\end{array}
$$

- Associative objects are formalizers.

- The associative object that formalizes a relationship is not the player of the relationship.
- The identifier of an associative object contains the combination of the identifiers of all players of the relationship that is formalized in the associative object.

The formalizing can be in one of the two forms when the formalizers are associative objects. The two forms are single occurrence and multiple occurrence:

$$
\begin{array}{l}
\textit{IMAssociativeObjectFormalizingType} \\
\hline
\textit{IMAssociativeObjectFormalizing} \\
\textit{singleOccurrence} : \textit{RELATIONSHIP} \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\;\; \mathrel{+\!\!\!\!\!\rightarrow} \textit{OBJECT} \\
\textit{multipleOccurrence} : \textit{RELATIONSHIP} \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;\;\; \mathrel{+\!\!\!\!\!\rightarrow} \textit{OBJECT} \\
\hline
\langle \textit{singleOccurrence}, \textit{multipleOccurrence} \rangle \ \mathsf{partition} \\
\qquad\qquad\qquad \textit{formalizedBy} \rhd \textit{associativeObject} \\
\hline
\end{array}
$$

We give further semantics in Section 5.2.6.

Formalizing Generalization Relationships

The schemas already discussed in this sections do not take generalization relationships into consideration. The generalization relationships are modeled by the subtype objects involved in the relationships:

IMGenRelationshipFormalizing

IMPlayerFormalizing

$$\forall\, obj : \mathrm{dom}\, isKindOf \bullet obj \in formalizer$$

$$\forall\, rel : genRelationship \bullet formalizedBy(rel) = playerOfRole(subRole(rel))$$

$$\forall\, rel : genRelationship;\ obj : object \mid formalizedBy(rel) = obj \bullet (referencing(obj))(\!|\ \{rel\}\ |\!) = hasIdElement(\!|\ \{hasPreferredId(obj)\}\ |\!)$$

- Every subtype object is a formalizer.
- Every generalization relationship is modeled by the subtype object in the relationship.
- The referential attributes of a formalizer is its preferred identifier.

Domain Restriction on Referential Attributes

A referential attribute has the same domain as the corresponding identifier attribute does. Given a relationship *rel*, if *a* is a referential attribute of *rel*'s formalizer *obj* and is also an identifier attribute of one of *rel*'s players, *obj'*, then the domain of *a* in *obj* is the same as the domain of *a* in *obj'*:

IMReferentialAttributeDomain

IMPlayerFormalizing

IMAssociativeObjectFormalizing

$$\forall\, rel : relationship \setminus genRelationship;\ obj : formalizer \mid formalizedBy(rel) = obj \bullet (\forall\, a : (referencing(obj))(\!|\ \{rel\}\ |\!);\ obj' : (hasRole \fatsemi playerOfRole)(\!|\ \{rel\}\ |\!) \bullet domainSpec(obj)(a) = domainSpec(obj')(a))$$

Combining the schemas about relationship formalizing, we obtain the following schema.

$$
\begin{aligned}
&IMRelationshipFormalizing \mathrel{\widehat=} \\
&\quad IMPlayerFormalizing \land \\
&\quad IMAssociativeObjectFormalizing \land \\
&\quad IMAssociativeObjectFormalizingType \land \\
&\quad IMGenRelationshipFormalizing \land \\
&\quad IMReferentialAttributeDomain
\end{aligned}
$$

5.1.7 Names and Naming

Objects, relationships, and attributes have names. Each object also has a unique key letter, which is an abbreviation of the object's name; and each relationship has a unique label as its short-form representation. We introduce given sets

$$[NAME, KEY\_LETTER, LABEL]$$

to represent all possible names, key letters, and labels.

Entity Names

The following two schemas formalize the naming of objects and relationships.

Schema *IMObjectName* defines

- a set of names for the objects in the information model,
- a function that maps each object to a unique object name, and
- a function that maps each object to a unique key letter.

$$
\begin{array}{l}
\textit{IMObjectName} \\
\hline
IMDTData \\
objectName : \mathbb{P}\, NAME \\
nameOfObject : OBJECT \rightarrowtail\!\!\!\!\!\!+\;\; NAME \\
keyLetterOfObject : OBJECT \rightarrowtail\!\!\!\!\!\!+\;\; KEY\_LETTER \\
\hline
nameOfObject \in object \rightarrowtail\!\!\!\!\!\rightarrow objectName \\
keyLetterOfObject \in object \rightarrowtail KEY\_LETTER
\end{array}
$$

- Each object is given a unique name.
- Each object is given a unique key letter.

Schema *IMRelationshipName* defines

- a set of names for the relationships in the information model,
- a function that maps each relationship to a relationship name, and
- a function that maps each relationship to a label.

IMRelationshipName

$$
\begin{array}{l}
IMDTData \\
IMGeneralization \\
relationshipName : \mathbb{P}\, NAME \\
nameOfRelationship : RELATIONSHIP \twoheadrightarrow NAME \\
labelOfRelationship : RELATIONSHIP \nrightarrow LABEL \\
\hline
nameOfRelationship \in relationship \setminus genRelationship \twoheadrightarrow relationshipName \\
labelOfRelationship \in (relationship \setminus genRelationship \rightarrowtail LABEL) \cup \\
\qquad\qquad (genRelationship \rightarrow LABEL) \\
\forall\, rel1, rel2 : genRelationship \bullet \\
\quad collectedIn(rel1) = collectedIn(rel2) \Leftrightarrow \\
\quad labelOfRelationship(rel1) = labelOfRelationship(rel2) \\
labelOfRelationship(\!|\, relationship \setminus genRelationship \,|\!) \cap \\
labelOfRelationship(\!|\, genRelationship \,|\!) = \emptyset
\end{array}
$$

- Each relationship, except for generalization relationships, has a name. Sometimes two or more relationships have the same name.
- Each relationship, except for generalization relationships, has a unique label.
- The generalization relationships in a generalization construct have a unique label.
- The labels for generalization relationships and the labels for other relationships are disjoint sets.

Feature Names

The naming of features is different from the naming of entities, because the scope of feature names is not global. The scope for features is the entity with

which they are associated. When formalizing the naming in the local scope, we parameterize the sets and functions by the scope. The following two schemas formalize the naming of attributes and roles.

Schema *IMAttributeName* defines, within a given object, a function from attributes to their names.

IMAttributeName

$$IMFeature$$
$$IMInheritance$$
$$nameOfAttribute : OBJECT \pfun (ATTRIBUTE \rightarrowtail NAME)$$

$$\mathrm{dom}\, nameOfAttribute = object$$
$$\forall obj : object \bullet nameOfAttribute(obj) \in hasAttribute(\!| \{obj\} |\!) \rightarrowtail NAME$$

- Each attribute of an object has a unique name.

In the information model, each end of a relationship, except for generalization relationships, is always attached a name to. This name specifies the role that end plays in the relationship. We regard this name as the role name, and call the combination of these role names as the name of the relationship. Schema *IMRoleName* defines, within a given relationship, a function from roles to their names.

IMRoleName

$$IMDTData$$
$$IMGeneralization$$
$$nameOfRole : RELATIONSHIP \pfun (ROLE \rightarrowtail NAME)$$

$$\mathrm{dom}\, nameOfRole = relationship \setminus genRelationship$$
$$\forall rel : \mathrm{dom}\, nameOfRole \bullet nameOfRole(rel) \in hasRole(\!| \{rel\} |\!) \rightarrowtail NAME$$

- Each role of a relationship, except for generalization relationships, has a unique name.

The following schema combines the properties of the above name-related schemas.

$$
\begin{aligned}
&IMName \mathrel{\widehat=} \\
&\quad IMObjectName \land \\
&\quad IMRelationshipName \land \\
&\quad IMAttributeName \land \\
&\quad IMRoleName
\end{aligned}
$$

5.1.8 Descriptions

An *object description* is a short, informative statement which allows one to tell, with certainty, whether or not a particular real-world thing is an instance of the object as conceptualized in the information model. An *attribute description* is a short, informative description that tells how the formal attribute reflects the real-world characteristic of interest. A *relationship description* is a short, informative description that provides statements about the relationship between the objects. Let

$$[TEXT]$$

represent such descriptions.

Schema *IMDescription* defines

- a function from objects to their text descriptions,
- a function from relationships to their text descriptions, and
- a function, within the scope of the objects, from attributes to their text descriptions.

IMDescription

$$
\begin{array}{l}
IMFeature \\
objectDescription : OBJECT \pfun TEXT \\
relationshipDescription : RELATIONSHIP \pfun TEXT \\
attributeDescription : OBJECT \pfun (ATTRIBUTE \pfun TEXT)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{dom}\, objectDescription = object \\
\mathrm{dom}\, relationshipDescription = relationship \\
\mathrm{dom}\, attributeDescription = object \\
\forall\, obj : object \bullet \\
\quad attributeDescription(obj) \in hasFeature(\!|\, \{obj\} \,|\!) \rightarrow TEXT
\end{array}
$$

- An object description must be provided for each object in the information model.
- A relationship description must be provided for each relationship in the information model.
- An attribute description must be provided for each attribute of each object.

5.1.9 Information Model with Design-Time Concepts

Schema *IMDTModel* defines an information model involving the concepts discussed in the previous sections.

$$
\begin{array}{l}
IMDTModel \mathrel{\widehat=} \\
\quad IMDTData \land IMFeature \land \\
\quad IMBinaryRelationshipType \land IMCompositionRelationship \land \\
\quad IMGeneralization \land IMInheritance \land \\
\quad IMRelationshipFormalizing \land \\
\quad IMName \land IMDescription
\end{array}
$$

5.2 Run-Time Concepts

A run-time model represents the internal operation of a running application. It is described with a set of run-time concepts, which represent the instances of the design-time concepts of the system. In choosing how to formalize the run-time concepts, we focus on choosing features required to explain the semantics of an information model. Since an information model describes the static view of the system, we do not discuss the concepts related to timing and order of operations.

In this section, we formally define the run-time concepts, including *instance*, *real-world instance*, and *attribute value*. Moreover, we further formalize the semantics of the design-time concepts by specifying the relationships between the design-time and run-time concepts.

5.2.1 Fundamental Concepts

At run-time, an object can be instantiated as a set of *instance*s. These object instances are real-world things that the object represents. They have the same characteristic and behavior. A relationship also has a set of relationship instances, each of which connects instances of objects that participate in the relationship. We introduce a given set

$$[INSTANCE]$$

and two subsets, one each for object instances and relationship instances,

$$OBJ\_INSTANCE : \mathbb{P}\, INSTANCE$$
$$REL\_INSTANCE : \mathbb{P}\, INSTANCE$$
$$\langle OBJ\_INSTANCE, REL\_INSTANCE \rangle \text{ partition } INSTANCE$$

In an inheritance hierarchy, a real-world thing is not represented by a single subtype or supertype object, but by the combination of a set of objects in the

hierarchy (see [SM92]: pp.28-29). Let

$$[RW\_INSTANCE]$$

represent all possible real-world instances.

A basic run-time system consists of

- a set of instances,
- a set of object instances,
- a set of relationship instances, and
- a set of real-world instances.

$$\begin{array}{|l}
\textit{IMRTData} \\
\hline
instance : \mathbb{P}\, INSTANCE \\
objInstance : \mathbb{P}\, INSTANCE \\
relInstance : \mathbb{P}\, INSTANCE \\
rwInstance : \mathbb{P}\, RW_INSTANCE \\
\hline
\langle objInstance, relInstance \rangle \ \mathsf{partition}\ instance \\
\hline
\end{array}$$

- Object instances and relationship instances are instances in the run-time system.

5.2.2 Entities and Their Instances

An object may have one or more instances in the run-time system being modeled. The following schema defines the relationship between objects and their instances.

IMRTInstance

$$
\begin{array}{l}
IMDTModel \\
IMRTData \\
isInstanceOf : INSTANCE \pfun ENTITY \\
hasInstance : ENTITY \rel INSTANCE \\
\hline
isInstanceOf \in (objInstance \fun object) \cup (relInstance \fun relationship) \\
hasInstance = isInstanceOf^{\sim}
\end{array}
$$

- Every instance belongs to an object or a relationship.
- An object or a relationship may have one or more instances.

5.2.3 Attributes and Their Values

An *attribute value* is the data value held by an object. Schema *IMRTAttributeValue* defines a function specifying the value of each attribute in the object instances.

IMRTAttributeValue

$$
\begin{array}{l}
IMRTInstance \\
holdValue : OBJ\_INSTANCE \pfun (ATTRIBUTE \pfun ATOMIC\_VALUE) \\
\hline
\mathrm{dom}\, holdValue = objInstance \\
\forall\, obji : objInstance \bullet \\
\quad \mathrm{dom}(holdValue(obji)) = hasAttribute(\!| \{isInstanceOf(obji)\} |\!) \\
\forall\, obji : objInstance;\ a : attribute;\ v : ATOMIC\_VALUE \mid \\
\quad v = holdValue(obji)(a) \bullet \\
\quad\quad v \in (domainSpec(isInstanceOf(obji)))(\!| \{a\} |\!)
\end{array}
$$

- The domain of function *holdValue* is the set of object instances in the information model.

- Each attribute of an object instance has a value.
- The attribute value is within the value range specified for the attribute.

The following schema specifies the semantics of attribute identifiers in terms of run-time concepts.

IMRTIdentifier

IMRTAttributeValue

$$\begin{aligned}
&\forall\, obj : object;\ obji1, obji2 : objInstance;\ id : identifier \mid \\
&\quad isInstanceOf(obji1) = isInstanceOf(obji2) = obj \land \\
&\quad isIdOf(id) = obj \bullet \\
&\qquad hasIdElement(\!|\, \{id\} \,|\!) \lhd holdValue(obji1) = \\
&\qquad hasIdElement(\!|\, \{id\} \,|\!) \lhd holdValue(obji2) \Leftrightarrow obji1 = obji2
\end{aligned}$$

- Two instances of an object must have different identifier values.

5.2.4 Generalizations and Real World Instances

SMOOA does not allow "abstract" object within the inheritance hierarchy. That is, if an object in an inheritance hierarchy has instances, its ancestors and descendants have instances as well.

IMRTGenInstance

IMRTInstance

$$\begin{aligned}
&\forall\, super, sub : object \mid sub \mapsto super \in isKindOf \bullet \\
&\quad hasInstance(\!|\, \{super\} \,|\!) \neq \emptyset \Leftrightarrow hasInstance(\!|\, \{sub\} \,|\!) \neq \emptyset
\end{aligned}$$

As discussed in Section 5.1.5, there is a bijection from the instances of the subtype objects to the instances of the supertype object.

IMRTGeneralization

$$
\begin{array}{l}
IMRTAttributeValue \\
insGen : GENERALIZATION \pfun (OBJ\_INSTANCE \rightarrowtail OBJ\_INSTANCE) \\
\hline
\dom insGen = \dom(supertype \fatsemi hasInstance) \\
\forall g : \dom insGen \bullet \\
\quad insGen(g) \in hasInstance(\!| \{subtype(\!| \{g\} |\!)\} |\!) \rightarrowtail \\
\qquad\qquad hasInstance(\!| \{supertype(g)\} |\!) \\
\forall g : generalization;\ subi, superi : objInstance \mid subi \mapsto superi \in insGen(g) \bullet \\
\quad hasIdElement(\!| \{hasPreferredId(isInstanceOf(subi))\} |\!) \lhd holdValue(subi) = \\
\quad hasIdElement(\!| \{hasPreferredId(isInstanceOf(superi))\} |\!) \lhd holdValue(superi)
\end{array}
$$

- The domain of *insGen* is the set of generalizations whose supertype (and also subtype) objects have instances in a run-time model.
- Given a generalization, the mapping from the subtype instances to the supertype instances is a bijection.
- For each instance pair in the bijection, the value of the identifier of one instance is the same as the value of the identifier of the other.

Based on the function defined in the above schema, we define a mapping from each subtype instance to all its corresponding supertype instances regardless of the generalization constructs.

IMRTInheritance

$$
\begin{array}{l}
IMRTGeneralization \\
insIsKindOf : OBJ\_INSTANCE \leftrightarrow OBJ\_INSTANCE \\
\hline
insIsKindOf = \{g : generalization;\ i1, i2 : objInstance \mid \\
\qquad\qquad i1 \mapsto i2 \in insGen(g) \bullet i1 \mapsto i2\}
\end{array}
$$

In a subtype-supertype construct, one real-world instance is composed of an instance of the supertype object and an instance of exactly one subtype object (see [SM92]: pp.28-29). An inheritance hierarchy consists of repeated subtype-supertype constructs and multiple inheritance. In such a hierarchy, a real-world instance is composed of a set of instances:

IMRTrwInstanceBasic

$$
\begin{array}{l}
IMRTInheritance \\
represent : OBJ\_INSTANCE \twoheadrightarrow RW\_INSTANCE \\
\hline
represent \in objInstance \twoheadrightarrow rwInstance \\
\\
\forall\, rw : rwInstance;\ i : objInstance \mid represent(i) = rw \bullet \\
\quad i \notin \operatorname{dom} insIsKindOf \cup \operatorname{ran} insIsKindOf \Rightarrow \\
\qquad \operatorname{dom}(represent \rhd \{rw\}) = \{i\} \land \\
\quad i \in \operatorname{dom} insIsKindOf \cup \operatorname{ran} insIsKindOf \Rightarrow \\
\qquad (\forall\, i' : insIsKindOf(\!|\, \{i\} \,|\!) \cup insIsKindOf^{\sim}(\!|\, \{i\} \,|\!) \bullet represent(i') = rw)
\end{array}
$$

- Every real-world instance can be represented by a set of object instances.
- The second predicate indicates that,
  - if an object is not in any inheritance hierarchy, then each of its instance represents a real-world instance;
  - if an object is in an inheritance hierarchy, then its instance i can only partially represent a real-world instance of the object. The instances that are in *insIsKindOf* relation with i also represent the real-world instance.

The following function defines a mapping from each object to the set of real-world instances it represents. If a real-world instance contains an instance of an object, we may say that the object has the real-world instance.

IMRTrwInstance

$$IMRTrwInstanceBasic$$
$$hasrwInstance : OBJECT \leftrightarrow RW\_INSTANCE$$

$$hasrwInstance = hasInstance \fatsemi represent$$

- Relation *hasrwInstance* is derived from the composition of *hasInstance* and *represent*.

We define that the attributes that a real-world instance has are the attributes that its representing instances have:

IMRTrwAttributeValue

$$IMRTAttributeValue$$
$$IMRTrwInstance$$
$$rwHoldValue : RW\_INSTANCE \pfun (ATTRIBUTE \pfun ATOMIC\_VALUE)$$

$$\dom rwHoldValue = rwInstance$$

$$\forall rwi : rwInstance \bullet$$
$$\quad \dom(rwHoldValue(rwi)) = hasAttribute(\!| hasrwInstance^{\sim}(\!| \{rwi\} |\!) |\!)$$

$$\forall rwi : rwInstance;\ obji : objInstance;\ a : attribute \mid$$
$$\quad represent(obji) = rwi \land$$
$$\quad a \in \dom(holdValue(obji)) \bullet$$
$$\quad\quad rwHoldValue(rwi)(a) = holdValue(obji)(a)$$

- The domain of *rwHoldValue* is the set of real-world instances in the information model.
- Each attribute of a real-world instance has a value.
- If a real-world instance consists of object instance *obji*, and *obji* has attribute a, then the value of a in the real-world instance is the same as the value of a in *obji*.

5.2.5 Relationships and Roles

In a run-time application, it is real-world instances, but not instances, of objects that participate in the relationships. Schema *IMRTRelationship* defines

- a function specifying player instances of each role in each relationship instance, and
- a relation mapping from each relationship instance to the real-world instances that participate in the relationship.

IMRTRelationship

$$
\begin{array}{l}
IMRTrwInstance \\
relConnect : REL\_INSTANCE \pfun (ROLE \pfun RW\_INSTANCE) \\
connection : REL\_INSTANCE \rel RW\_INSTANCE \\
\hline
\dom relConnect = relInstance \\
\forall reli : relInstance \bullet \\
\quad \dom(relConnect(reli)) = hasRole(\!| \{isInstanceOf(reli)\} |\!) \\
\forall reli : relInstance;\ r : role;\ rwi : rwInstance \mid \\
\quad rwi = relConnect(reli)(r) \bullet rwi \in hasrwInstance(\!| \{playerOfRole(r)\} |\!) \\
connection = \{reli : relInstance;\ rwi : rwInstance \mid \\
\quad (\forall r : role \bullet relConnect(reli)(r) = rwi)\}
\end{array}
$$

- The first three predicates state that each role of each relationship instance has a real-world instance as its player. This real-world instance is a real-world instance of the object that plays the role.
- Relation *connection* maps each relationship instance to the real-world instances that play the roles in the relationship.

The following schema gives the semantics of multiplicity of roles involved in binary relationships.

IMRTRoleType

$$
\begin{array}{l}
IMDTModel \\
IMRTInstance \\
IMRTRelationship \\
\hline
\forall\, r, r' : role;\ rel : binaryRelationship \mid \\
\quad hasRole(\!|\, \{rel\} \,|\!) = \{r, r'\} \bullet \\
\quad\quad (\forall\, rwi : hasrwInstance(\!|\, \{playerOfRole(r)\} \,|\!);\ n : \mathbb{N} \mid \\
\quad\quad\quad n = \#\{reli : hasInstance(\!|\, \{rel\} \,|\!) \mid \\
\quad\quad\quad\quad reli \mapsto rwi \in connection\} \bullet \\
\quad\quad\quad\quad n \in multiplicityOfRole(\!|\, \{r'\} \,|\!))
\end{array}
$$

- Given a binary relationship *rel* with its two roles r and r', and a real-world instance *rwi* of r, the number of the relationship instances of *rel* that the *rwi* participates in is the number of the real-world instances of r' that have relations with *rwi*. This number is one of the numbers in $multiplicityOfRole(\!|\, \{r'\} \,|\!)$.

5.2.6 Relationship Formalization

Function *insFormalizedBy* in the following schema gives the semantics of the associative objects in terms of run-time model. It associates each relationship and its formalizer with a function that maps from the instances of the relationship to the instances of the formalizer.

IMRTRelationshipFormalizingBasic

$$IMRTRelationship$$

$$insFormalizedBy : RELATIONSHIP \times OBJECT \pfun (RW\_INSTANCE \rel REL\_INSTANCE)$$

$$\forall rel : \dom formalizedBy \bullet (rel, formalizedBy(rel)) \in \dom insFormalizedBy$$

$$\forall rel : relationship;\ obj : object \mid (rel, obj) \in \dom insFormalizedBy \land obj \in playerOfRole(\!| hasRole(\!| \{rel\} |\!) |\!) \bullet (insFormalizedBy(rel, obj))^{\sim} \in hasInstance(\!| \{rel\} |\!) \rightarrowtail hasrwInstance(\!| \{obj\} |\!)$$

$$\forall rel : relationship;\ obj : object \mid (rel, obj) \in \dom insFormalizedBy \land rel \mapsto obj \in singleOccurrence \bullet insFormalizedBy(rel, obj) \in hasrwInstance(\!| \{obj\} |\!) \rightarrowtail hasInstance(\!| \{rel\} |\!)$$

$$\forall rel : relationship;\ obj : object \mid (rel, obj) \in \dom insFormalizedBy \land rel \mapsto obj \in multipleOccurrence \bullet insFormalizedBy(rel, obj) \in hasrwInstance(\!| \{obj\} |\!) \rightarrow\!\!\!\!\!\rightarrow hasInstance(\!| \{rel\} |\!)$$

- An element in the domain of *insFormalizedBy* is the pair of a relationship and its formalizer.
- If the formalizer of a relationship is one of its players, then the mapping from the formalizer instances to the relationship instances is an injection with the entire set of relationship instances as range.

- If the formalizer of a relationship is an associative object, and it is in a single occurrence form, then the mapping from the formalizer instances to the relationship instances is a bijection.

- If the formalizer of a relationship is an associative object, and it is in a multiple occurrence form, then the mapping from the formalizer instances to the relationship instances is a total surjection.

The following schema specifies the relationship between the values of formalizers' referential attributes and the values of players' identifiers.

IMRTRelationshipFormalizing

$$
\begin{array}{l}
IMRTrwAttributeValue \\
IMRTRelationshipFormalizingBasic \\
\hline
\forall\, rel : relationship;\ obj : object;\ reli : relInstance;\ rwi, rwi' : rwInstance \mid \\
\quad \{rwi, rwi'\} = connection(\!|\, \{reli\} \,|\!) \land \\
\quad rwi \mapsto reli \in insFormalizedBy(rel, obj) \bullet \\
\quad (referencing(obj))(\!|\, \{rel\} \,|\!) \lhd rwHoldValue(rwi) = \\
\quad hasIdElement(\!|\, \{hasPreferredId(hasrwInstance^{\sim}(rwi'))\} \,|\!) \lhd rwHoldValue(rwi') \\
\\
\forall\, rel : relationship;\ obj : object;\ reli : relInstance;\ rwi : rwInstance \mid \\
\quad rwi \notin connection(\!|\, \{reli\} \,|\!) \land \\
\quad rwi \mapsto reli \in insFormalizedBy(rel, obj) \bullet \\
\qquad (referencing(obj))(\!|\, \{rel\} \,|\!) \lhd rwHoldValue(rwi) = \\
\qquad \bigcup \{rwi' : connection(\!|\, \{reli\} \,|\!) \bullet \\
\quad hasIdElement(\!|\, \{hasPreferredId(hasrwInstance^{\sim}(rwi'))\} \,|\!) \lhd rwHoldValue(rwi')\}
\end{array}
$$

- If the formalizer of a binary relationship is one of its players, then the values of the referential attributes of the formalizer instance should be the same as the values of the identifier attributes of the other player instance that participates in the same relationship instance.

- If the formalizer of a relationship is an associative object, the values of the referential attributes of the formalizer instance should be the same as the values of the identifier attributes of all the player instances that participate in the same relationship instance.

5.2.7 Information Model with Run-Time Concepts

Schema *IMRTModel* defines the information model involving the run-time concepts.

$$
\begin{aligned}
&IMRTMOdel \mathrel{\widehat=} \\
&\quad IMRTData \land \\
&\quad IMRTInstance \land IMRTrwInstance \land \\
&\quad IMRTAttributeValue \land IMRTrwAttributeValue \land \\
&\quad IMRTIdentifier \land \\
&\quad IMRTRelationship \land IMRTRoleType \land \\
&\quad IMRTRelationshipFormalizing
\end{aligned}
$$

5.3 Information Model

In summary, the theory for the information model is represented as follows.

$$IM \mathrel{\widehat=} IMDTModel \land IMRTMOdel$$

Chapter 6

Approach to Comparing the OOAD Methods

This chapter discusses the approach to comparing the representational properties of the OOAD methods. We propose a mechanism to systematically reason about their similarities and differences. The formal descriptions of the methods serve as the theoretical basis for the comparison. The basic idea of the comparison approach is to extract from these abstract theories a core which contains the features common to all the methods being compared, and characterize each of the methods by an extension to the core.

In this chapter, we fulfill the following tasks: (1) formally define the concepts and notations that are used to describe the comparison approach, (2) identify some patterns of similarities and differences between the methods, (3) define the core, the extensions to the core, and the characterizing sets in the core, (4) describe the procedure that a comparison process should follow, and (5) give decision rules to guide the formation of the core and the extensions.

We make use of mathematical notations of sets, relations, and functions to define the concepts and notations discussed in this chapter. We also use Z symbols when appropriate, to reduce the number of new mathematical notations

that have to be introduced[1].

6.1 Formal Definitions of Concepts

This section gives the definitions of the concepts and notations that the comparison is based on.

6.1.1 Type

Z is based on ***typed set theory***. Each variable declared in Z is given a type. As the design methods are formalized using the Z notation, we need to present the formal definition of types. The simplest types are given sets. More complex types can be built up with type-constructors. The type-constructors available in Z are power set, Cartesian product, and schema [Spi88]. The formal definition of *types* is given in Definition 6.1, which is simplified from Spivey's definition[2].

Definition 6.1 *Types* are defined as follows.

1. Given sets are types.
2. If t is a type, so is $\mathbb{P}\, t$.
3. If t_1, t_2, ..., t_n are types, so is their Cartesian product $t_1 \times t_2 \times \ldots \times t_n$.
4. If the declaration part of a schema has n variables x_1, x_2, ..., x_n, with types t_1, t_2, ..., t_n, respectively, then the schema is of type $\langle\!|\ x_1 : t_1;\ x_2 : t_2;\ \ldots;\ x_n : t_n\ |\!\rangle$.

[1]Alternatively, we can employ exclusively Z notation to serve the purpose. In this way, there will be a uniform style throughout the dissertation. This can be done in the future research by translating the mathematical notations used in this chapter to Z. For now, it is not a critical issue.

[2]If the definition of a concept is adapted from Spivey, we give explicit citation; otherwise, the definition is given by us.

Note that $X \times Y \times Z$, $X \times (Y \times Z)$, and $(X \times Y) \times Z$ are different types. Also, note that $\langle\!\!| \ x : X; \ y : Y \ |\!\!\rangle$ and $\langle\!\!| \ y : Y; \ x : X \ |\!\!\rangle$ are the same type: order of components does not matter.

More elaborate structures like functions and sequences which are also used in declarations are not part of the Z type system. These structures are merely shorthands for the types described above, e.g., function $A \rightarrowtail B$ has type $\mathbb{P}(A \times B)$, with certain restrictions on it.

A type determines a set of values which are its elements. This set is called *carrier* of the type [Spi88]. Here we employ the concept "world of sets" introduced in [Spi88]. Let W stand for the world of sets. The carrier of a type is an element of W.

6.1.2 Signature and Predicate

A *signature* defines a set of types, a set of carriers for the types, a set of variables, and a declaration that assigns a type to each variable.

A declaration is *normalized* if it introduces variables by specifying their names and types without imposing further restrictions on them. The declaration $x : \mathbb{Z}$ is normalized because $\mathbb{Z}$ is a type. On the other hand, $x : 0..10$ is *non-normalized* because, besides defining the type of x (i.e., $\mathbb{Z}$), it specifies that $x \in 0..10$.

Definition 6.2 We call $Sig = \langle Type, Carrier, Var, Decl \rangle$ a *normalized signature*, where

1. *Type* is a set of types in Z.
2. $Carrier : Type \rightarrow W$ is a total function, which defines a carrier for each type.
3. *Var* is a set of variables.

4. $Decl : Var \rightarrow Type$ is a total function, which relates each variable to its corresponding type.

As is mentioned in Section 3.3.3, the variables in a theory are divided into two classes: category variables and relation variables. Thus, given a signature Sig, the variable set Var can be further divided into two subsets $catVar$ and $relVar$, that is,

$$\langle catVar, relVar \rangle \text{ partition } Var$$

We can always formalize category variables in any design theory to be set-valued.

Let Sig and Sig' be normalized signatures. Sig' is a *subsignature* of Sig, denoted as $Sig' \subseteq Sig$, if

$$Type' \subseteq Type$$
$$Carrier' = Type' \lhd Carrier$$
$$Var' \subseteq Var$$
$$Decl' = Var' \lhd Decl$$

Definition 6.3 Given a normalized signature Sig, a predicate p of Sig is a first-order formula containing the usual connectives and quantifiers. Free variables and the types of bound variables of p must be members of Var and $Type$ respectively.

It can easily be proven that, if Sig' is a subsignature of Sig and p is a predicate of Sig', then p is a predicate of Sig.

Definition 6.4 We call $Sig^* = \langle Type^*, Carrier^*, Var^*, Decl^*, Var'^*, Pred'^*, Cons^* \rangle$ an *extended signature*, where

1. $Type^*$, $Carrier^*$, Var^*, $Decl^*$ have the same definitions as those defined in a normalized signature. That is, $\langle Type^*, Carrier^*, Var^*, Decl^* \rangle$ is a normalized signature.

2. Var'^* is a subset of Var^*, i.e., $Var'^* \subseteq Var^*$. The declarations of the variables in Var'^* impose further restrictions on the variables besides the assignment of types.

3. $Pred'^*$ is a set of predicates.

4. $Cons^* : Var'^* \rightarrowtail\!\!\!\!\!\rightarrow Pred'^*$ is a bijection. For each variable $v \in Var'^*$ whose declaration imposes certain restriction on the values that v can take, $Cons^*(v)$ represents the restriction, and it is a predicate of $\langle Type^*, Carrier^*, Var^*, Decl^* \rangle$.

A predicate of the extended signature Sig^* is defined as a predicate of $\langle Type^*, Carrier^*, Var^*, Decl^* \rangle$.

6.1.3 Theory

A formalization of a design method under the Theory-Model paradigm is called a *theory* of the method. The theory is represented in terms of a set of schemas in Z. As stated in Chapter 3, a theory is composed of a set of categories of concepts and rules that the elements of these categories must satisfy. In a Z schema, the categories of the concepts for a design are formalized as variables, and the rules as predicates.

If the declaration of a schema is normalized, we say that the schema is normalized itself. If all the schemas of a theory are normalized, the theory is normalized.

Definition 6.5 A *normalized theory* is a tuple $T = \langle Sig, Pred \rangle$, where

1. Sig is a normalized signature.

2. $Pred$ is a set of predicates of Sig.

3. T is closed. That is, if p is a predicate of Sig and $Pred \models p$, then $p \in Pred$. According to the Completeness Theorem [CK73], this is the same as requiring that T be closed under $\vdash$.

For a normalized theory, *Sig* corresponds to the declaration parts of the Z schemas, and *Pred* corresponds to the predicate parts. It should be noted that a design theory is closed. The standard way of giving the properties that a design theory T possesses is by listing a finite set of predicates for T. The predicates in the Z schemas for a theory are such a set. Any other predicate which is not given in the schemas but is consequence of the predicates in the schemas belongs to the theory. This property of a design theory is important. During the comparison, it is sometimes necessary to add new predicates to facilitate comparing a certain part of one theory with another. These auxiliary predicates are obtained by applying rules of inference to, or by changing the forms of, the existing predicates of the theory. With the property that a theory is closed, we can assure that the new predicates belong to the theory.

For human readability, not all the schemas are normalized in our formalization. If a schema declares that a relation variable is a function, then the schema is non-normalized. We call a theory containing non-normalized schemas an *extended theory*.

Definition 6.6 An *extended theory* is a tuple $T^* = \langle Sig^*, Pred^* \rangle$, where

1. Sig^* is an extended signature.
2. $Pred^*$ is a set of predicates of Sig^*.
3. T^* is closed. That is, if p^* is a predicate of Sig^* and $Pred^* \cup Pred'^* \models p^*$, then $p^* \in Pred^* \cup Pred'^*$.

We will analyze the formalization in terms of normalized theories.

Proposition 6.1 Every extended theory can be normalized.

Proof. A non-normalized schema can always be transformed into an equivalent normalized schema by simply moving the restrictions introduced in its declaration about the variables to its predicate part.

After every schema of a theory is normalized, the theory is normalized.

Let T^* be an extended theory and T be the corresponding normalized theory after the transformation, we have

$$Sig = \langle Type^*, Carrier^*, Var^*, Decl^* \rangle$$
$$Pred = Pred^* \cup Pred'^* \quad \blacksquare$$

As an extended theory can be transformed to an equivalent normalized theory, the discussion about properties of normalized theories also applies to extended theories. We will use "theory" when the distinction is not relevant.

6.1.4 Subtheory

Definition 6.7 Given theories $T = \langle Sig, Pred \rangle$ and $T' = \langle Sig', Pred' \rangle$, T' is a *subtheory* of T, denoted as $T' \subseteq T$, if

1. $Sig' \subseteq Sig$
2. $Pred'$ is a set of predicates of Sig', and $Pred' \subseteq Pred$.

A schema or a combination of several schemas of the entire schema set for a design theory is in fact a subtheory of the design theory.

6.1.5 Model

Definition 6.8 An *interpretation* for a theory T is defined as $I = \langle Val, Int \rangle$, where

1. Val is a set of values, and $Val \subset W$.
2. $Int : Var \rightarrow Val$ is a total function, which assigns a value in Val to every variable in Var with correct type, that is, it must satisfy the following constraint:

$$\forall\, v : Var \bullet Int(v) \in Carrier(Decl(v))$$

Any concrete design developed under a design method is an interpretation of the corresponding design theory.

Definition 6.9 An interpretation I of theory T is a *model* of T iff, under I, every predicate in *Pred* is true.

Formally, a model of T is denoted as m^T, and the set of all models of T is denoted as M^T.

Definition 6.10 Let m^T be a model of theory T and Sig' be a subsignature of T's signature. A *submodel* of m^T restricted on Sig' is defined as $m^T \uparrow Sig' = \langle Val', Int' \rangle$, where,

1. $Val' \subseteq Val$
2. $Int' = Var' \rightarrow Val' = Var' \lhd Int$

A submodel corresponds to a partial concrete design.

Proposition 6.2 A submodel of a model of theory T restricted on the signature of T's subtheory T' is a model of T'.

Proof. Let T be a theory, T' be a subtheory of T, m^T be a model of T, and m' be a submodel of m^T restricted on Sig'.

Firstly, we prove that m' is an interpretation of T'. According to the definition for submodel, the following exist

(1). $Val' \subseteq Val \subset W$
(2). $Int' = Var' \rightarrow Val'$

From $Int' = Var' \lhd Int$ and $Decl' = Var' \lhd Decl$, we have

(3). $\forall\, v : Var' \bullet Int'(v) = Int(v) \wedge Int(v) \in Val'$
(4). $\forall\, v : Var' \bullet Decl'(v) = Decl(v)$

From $Carrier' = Type' \lhd Carrier$, we have

$$(5). \quad \forall\, t : Type' \bullet Carrier'(t) = Carrier(t)$$

From (3), (4), (5), and $Var' \subseteq Var$, we have

$$(6). \quad \forall\, v : Var' \bullet Int(v) \in Carrier(Decl(v)) \\ \Rightarrow \forall\, v : Var' \bullet Int'(v) \in Carrier'(Decl'(v))$$

(1), (2), and (6) show that m' is an interpretation of T'.

Secondly, we prove that, under m', every predicate p in $Pred'$ is true. According to the definition of subtheory, p is a predicate in $Pred$ and is a predicate of Sig', so its free variables are in Var'. Since each v in Var' takes the same value in both m^T and m', if p is true under m^T, p is also true under m'.

According to the definition of model, m' is a model of T'. ■

On the contrary, not every model of T' can be extended to a model of T, because a predicate in $(Pred \setminus Pred')$ may impose additional constraints on variables in Var'.

6.1.6 Specification Space and State Space

As discussed in Chapter 3, a design theory involves both design-time concepts and run-time concepts. The signature part of the theory contains variables for both kinds of concepts. Thus, a theory can be divided into two parts: design-time and run-time.

Definition 6.11 Assume that T is the theory for a design method. The *design-time signature*, *dtSig*, of T describes variables only concerning design-time concepts and their relationships. The *run-time signature*, *rtSig*, of T describes variables involving run-time concepts. It includes category variables

for run-time concepts and relation variables that specify relationships between run-time concepts and design-time concepts.

Both design-time and run-time signatures are subsignatures of $T.Sig$, and the following holds,

$$\langle dtSig.Var, rtSig.Var \rangle \text{ partition } Sig.Var$$

Definition 6.12 Assume that T is the theory for a design method. The *design-time theory*, T_D, of T is a subtheory of T, for which:

1. $T_D.Sig = T.dtSig$
2. $T_D.Pred \subseteq T.Pred$, and $T_D.Pred$ is the maximum set of predicates of $T_D.Sig$.

The *run-time theory*, T_R, of T is a subtheory of T, for which:

1. $T_R.Sig = T.rtSig$
2. $T_R.Pred \subseteq T.Pred$, and $T_R.Pred$ is the maximum set of predicates of $T_R.Sig$.

The combination of T_D and T_R is T.

The signature of a theory T corresponds to a *theory space*, which is multi-dimensional. Each axis represents a variable in $T.Sig.Var$ with the corresponding type defined in $T.Sig.Decl$, namely a pair of (v, t). The theory space of T's subtheory T' is the subspace of T's theory space projected on $T'.Sig$. Each axis of such a subspace represents a pair of (v, t) in $T'.Sig.Decl$. The design-time theory T_D and run-time theory T_R of T correspond to a *specification space* and a *state space* respectively. The specification space can be viewed as the theory space projected on $dtSig$. Similarly, the state space is the theory space projected on $rtSig$. The two spaces are subspaces of the theory space.

A concrete design is represented by a point in the specification space, and a run-time application is represented by a point in the state space. Each point in the specification space corresponds to a set of points in the state space, meaning that these state points represent the run-time applications of the concrete design. All the models of the theory constitute a subset of the theory space. We call the points in this subset the *legal points* in the theory space. Accordingly, the legal points in a theory space projected on the specification space and the state space of the theory space are the legal points of these subspaces. Figure 6.1 illustrates two theory spaces, their specification spaces and state spaces, and the equivalence mapping between the legal points of the spaces.

According to Proposition 6.2, given a subtheory T' of theory T, the legal points of T's theory space projected on T''s theory space are a subset of the legal points of T''s theory space.

6.1.7 Semantic Equivalence

One design method having the same expressive power as another one, means every application developed under the former can also be developed under the latter. We say that the two methods are *semantically equivalent*. By this definition, there should be one-to-one mapping between the legal points in the state spaces of two semantically equivalent design theories.

A run-time theory cannot stand on its own, because its semantics by itself is not complete. A run-time instance relies heavily on the design-time theory in the sense that the instance conforms to the constraints on its design-time counterpart, as is already discussed in Section 3.3.1. To be more precise, semantic equivalence with respect to two design theories means that given a model for one theory, there should be a corresponding model for the other theory, and vice versa.

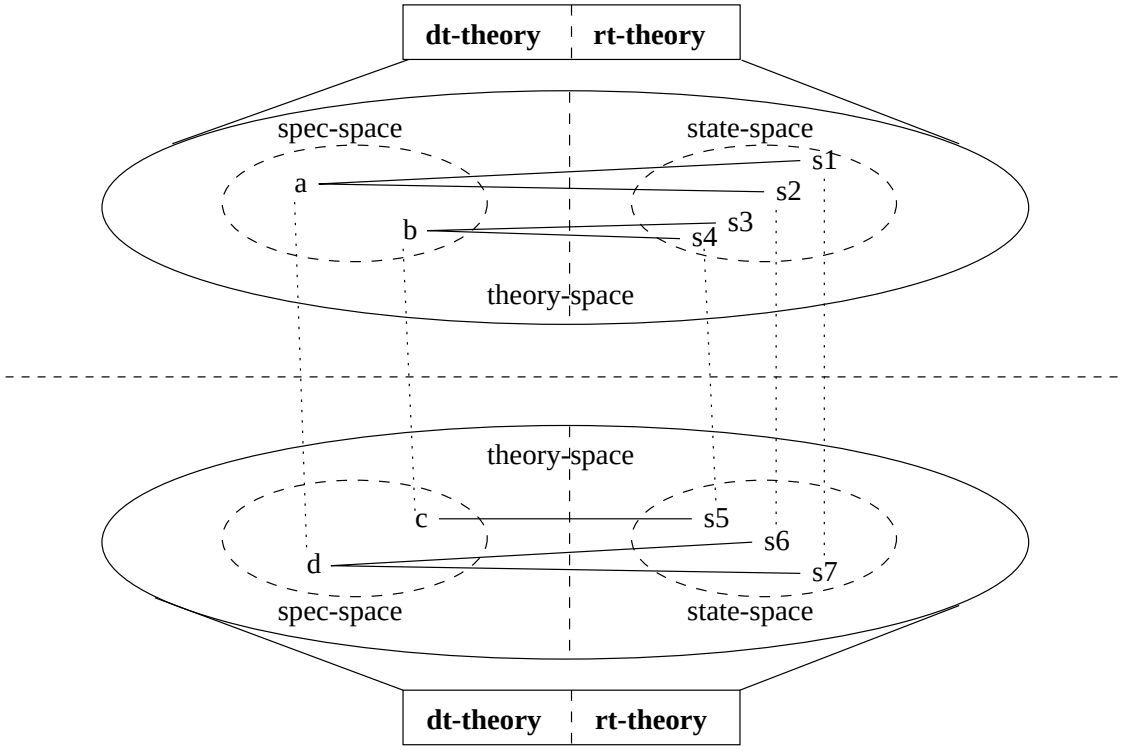


Figure 6.1: Two design theories and their theory spaces.

a, b, c, and d are legal points in the specification spaces; s1 .. s7 are legal points in the state spaces. A line connecting a point in the specification spaces with a point in the state spaces associates a concrete design with its run-time application. A pair of lines drawn from a pair of points in one theory space to a pair of points in the other means that the two models represent the same real-world application. s3 does not have a corresponding point in the other theory, meaning that the application represented by (b, s3) cannot be modeled by the other design method.

Definition 6.13 Theory T_1 *semantically includes* theory T_2 ($T_2 \sqsubseteq_{sem} T_1$) iff, for every legal point m^{T_2} in the theory space of T_2, there is a legal point m^{T_1} in the theory space of T_1 so that m^{T_1} and m^{T_2} model the same application in the real-world.

Such a pair of m^{T_1} and m^{T_2} is defined to have equivalence relation "$\equiv_{sem}$", that is,

$$m^{T_1} \equiv_{sem} m^{T_2}$$

Definition 6.14 Theory T_1 is *semantically equivalent* to theory T_2 iff

$$T_1 \sqsubseteq_{sem} T_2 \wedge T_2 \sqsubseteq_{sem} T_1$$

Definition 6.15 Theory T_1 is *not semantically equivalent* to theory T_2 iff

$$\neg (T_1 \sqsubseteq_{sem} T_2) \wedge \neg (T_2 \sqsubseteq_{sem} T_1)$$

Vertical lines between the two theory spaces in Figure 6.1 illustrate the equivalence mapping between the models of the two theories. In reality, because of the semantic differences between the methods, usually the theories are not semantically equivalent. That is, only a subset of the models of one theory has counterparts in the other theory, and vice versa. What the comparison will do is to determine which subset of the models of one theory is semantically equivalent to which subset of the models of the other theory.

6.2 Mappings between Components of the Theories

The fundamental components of a theory are types, variables, and predicates. A comparison is performed by comparing these components of two theories. First of all, we need to determine which components are comparable and which are not. This section discusses mappings between comparable components.

6.2.1 Mapping between Types

Before defining the comparable mapping between types of two theories in general, we must define the comparable mapping between given sets, i.e., basic types. The mapping between basic types is established according to (1) the original definitions for the design categories in the two methods, and (2) the semantics of the categories described in the theories. If two concepts address similar issues, have similar definition, comply with similar constraints, and have similar relationships with other comparable concepts, then the two basic types that represent the two concepts are defined to be comparable. For example, we define "class" in OMT to be comparable to "object" in SMOOA, because they have similar definition and similar relationships to other concepts, and constraints on them are similar.

Based on the mapping between basic types, we are able to define the comparable mapping between any other types.

Definition 6.16 The two types $T_1.t$ and $T_2.t$, one each defined in theories T_1 and T_2 respectively, are comparable

$$T_1.t \approx_{ty} T_2.t$$

iff one of the following situations is satisfied:

1. $T_1.t$ and $T_2.t$ are given sets, and we have defined the two given sets to be comparable;

2. $T_1.t = \mathbb{P}\, T_1.t'$ and $T_2.t = \mathbb{P}\, T_2.t'$ and

   $$T_1.t' \approx_{ty} T_2.t'$$

3. $T_1.t = T_1.t_1 \times T_1.t_2 \times \ldots \times T_1.t_n$ and $T_2.t = T_2.t_1 \times T_2.t_2 \times \ldots \times T_2.t_n$, and for every $i \in 1 \,..\, n$,

   $$T.t_i \approx_{ty} T_2.t_i$$

4. $T_1.t = \langle\!|\ T_1.x_1 : T_1.t_1;\ T_1.x_2 : T_1.t_2;\ \ldots;\ T_1.x_n : T_1.t_n\ |\!\rangle$ and $T_2.t = \langle\!|\ T_2.x_1 : T_2.t_1;\ T_2.x_2 : T_2.t_2;\ \ldots;\ T_2.x_n : T_2.t_n\ |\!\rangle$, and there is an isomorphism f between $T_1.t$ and $T_2.t$, which satisfies that for every $i \in 1..n$, there is a unique $j \in 1..n$ so that $f((T_1.x_i : T_1.t_i)) = (T_2.x_j : T_2.t_j)$ and

$$T_1.t_i \approx_{ty} T_2.t_j$$

If two types are comparable, then we assume that some values in the two carrier sets are the same. That is, if $T_1.t \approx_{ty} T_2.t$, then

$$T_1.Carrier(T_1.t) \cap T_2.Carrier(T_2.t) \neq \emptyset$$

6.2.2 Mapping between Variables

After defining the comparable mapping between types of the two theories, we are able to define the comparable mapping between signatures, and hence the comparable mapping between variables of the theories.

Definition 6.17 Let T_1 and T_2 be two theories.

1. Signature mapping from T_1 to T_2 is defined as an injection:

   $$f_{sig|(T_1,T_2)} : T_1.Sig.Decl \rightarrowtail T_2.Sig.Decl$$

   It satisfies that, for every $(T_1.v, T_1.t) \in T_1.Sig.Decl$ and every $(T_2.v, T_2.t) \in T_2.Sig.Decl$, if $f_{sig|(T_1,T_2)}((T_1.v, T_1.t)) = (T_2.v, T_2.t)$, then $T_1.t \approx_{ty} T_2.t$. The reverse may not be true.

   The signature mapping from theory T_2 to theory T_1 is the inverse of $f_{sig|(T_1,T_2)}$, i.e., $f_{sig|(T_2,T_1)} = f^{\sim}_{sig|(T_1,T_2)}$.

2. From a signature mapping $f_{sig|(T_1,T_2)}$, we can define a variable mapping $f_{v|(T_1,T_2)}$:

   $$f_{v|(T_1,T_2)} : T_1.Sig.Var \rightarrowtail T_2.Sig.Var$$

It satisfies that

$$\text{if} \quad f_{sig|(T_1,T_2)}((T_1.v, T_1.t)) = (T_2.v, T_2.t), \quad \text{then} \quad f_{v|(T_1,T_2)}(T_1.v) = T_2.v$$

$T_1.v$ and $T_2.v$ are said to be comparable, and are called *comparable variables*.

Determining the comparability of two variables is similar to determining the comparability of two types, as discussed at the beginning of Section 6.2.1. The comparable variables must have comparable types. For example, we define variables *singularRole* in *OM* and *single* in *IM* to be comparable, because they have comparable types (i.e., *ROLE* in both theories), and the second predicate in schema *OMRoleType* (on p.57) and the second predicate in schema *IMRoleType* (on p.126) show that they have similar meaning. However, two variables with comparable types may not be comparable variables. For example, *association* in *OM* and *binaryRelationship* in *IM* have comparable types, but these two variables are not comparable because association is a more general concept than binary relationship.

Sometimes, a set-valued variable $T_1.v$ in theory T_1 corresponds to two or more set-valued variables in theory T_2. In this case, we introduce two or more new variables, in T_1, the values of which partition the values of $T_1.v$. In this way, each of the newly introduced variable is related to one of the variables in T_2. The new variables are used as auxiliary variables during the comparison.

6.2.3 Non-comparable Variables

If $(T_1.Sig.Var \backslash \mathrm{dom}\, f_{v|(T_1,T_2)}) \neq \emptyset$, then there must be some variables defined in theory T_1 having no counterparts in theory T_2. After analyzing *OM* and *IM*, we find that if this is the case, a variable $T_1.v \in (T_1.Sig.Var \setminus \mathrm{dom}\, f_{v|(T_1,T_2)})$ is in one of the following situations.

Redundant. Variable $T_1.v$ is redundant. When formalizing a method, we sometimes use auxiliary variables to make the theory more readable.

Without these variables, the theory does not change the semantics of the method. In the following formula,

$$a : X \leftrightarrow Y, b : Y \leftrightarrow X \mid b = a^{\sim}$$

either a or b is redundant. Whenever the redundant variable $T_1.v$ appears in a predicate, it can always be replaced by a non-redundant variable in T_1. Therefore, $T_1.v$ can be ignored during the comparison.

Explicitly defined. The concept in T_2 exists, but is not explicitly mentioned. It may be made explicit by adding an auxiliary variable. For example, SMOOA has naming, descriptive, and referential attributes, but OMT does not classify attributes into such detail. Nevertheless, attributes in OMT can also be naming and descriptive, although not referential. Thus, we may add two auxiliary variables, *naming* and *descriptive*, in the theory of OMT, to represent the two subsets of the attribute set. The added auxiliary variable in T_2 then can be compared to $T_1.v$.

Mistyped. T_2 has the variable $T_2.w$ that specifies the same aspect of a design as $T_1.v$ does, but the type of $T_2.w$ is not comparable to the type of $T_1.v$. We may redefine either $T_1.v$ or $T_2.w$, such as changing a schema to a relation, to make the two variables comparable.

Underlying. T_2 does not support the concept, but supports its underlying meaning. For example, a design in OMT may use one or more *rules* to restrict the legal values that an attribute of certain class can take on; but SMOOA has no such concept as rules. Instead, a design in SMOOA gives direct description of an attribute's value range once the attribute is defined in an object. To formalize this aspect of OMT, we need to introduce type *RULE* and several variables such as *generalRestrict*, *constraint*, *valueRange*, etc. By contrast, for SMOOA, we only need to

introduce one variable, i.e. *domainSpec*. Despite their different appearances, both theories have the same capability to specify a value range for an attribute. Therefore, we can regard the description of each attribute in an information model as adding rules about it.

Subsumed. $T_1.v$ can be "subsumed" in another concept. In other words, the concept specified by $T_1.v$ is used to explain another concept. *objInstance* in *IM* is such a variable. It is used to explain *rwInstance* (on p.153).

Technically different. The two methods use different techniques to model the same aspect of an application. The two sets of concepts involved in the modeling techniques thus may not have one-to-one correspondence, although they actually result in the same real-world applications. They may employ different notions, which become non-comparable category variables; and among the category concepts there may exist different relationships, which yield non-comparable relation variables. For example, SMOOA "formalizes" relationships by means of referential attributes but OMT does not have similar modeling technique, so SMOOA has extra concepts in this aspect. However, the two methods have similar expressive power.

In this case, we need to reformulate some definitions so that the two methods in this aspect are as similar as possible and then we can define the mapping between variables.

Pure non-comparable. T_2 does not provide the concept, also does not support the underlying meaning. For example, *operation* in OMT is not supported in SMOOA. In this case, $T_1.v$ is a "pure" non-comparable variable.

6.2.4 Mapping between Predicates

Definition 6.18 Let $T_1.p$ be a predicate formula of theory T_1, and $T_1.v_1$, $T_1.v_2$, ..., $T_1.v_n$ be all free variables in $T_1.p$. If each of these variables has a comparable variable in theory T_2, then $T_1.p$ is *transformable* from T_1 to T_2.

Let $T_1.v_i$ and $T_2.v_i$ be comparable variables, i.e., $T_2.v_i = f_{v|(T_1,T_2)}(T_1.v_i)$, where $i \in 1..n$. $T_1.p[T_2.v_1/T_1.v_1,\ T_2.v_2/T_1.v_2,\ \ldots,\ T_2.v_n/T_1.v_n]$ (simply denoted as $T_1.p[T_2/T_1]$) is a substitution of $T_2.v_1, T_2.v_2, \ldots, T_2.v_n$ for $T_1.v_1, T_1.v_2, \ldots, T_1.v_n$. $T_1.p[T_2/T_1]$ is a predicate of $T_2.Sig$.

Let $T_1.P$ be a set of predicates, $T_1.P = \{T_1.p_1, T_1.p_2, \ldots, T_1.p_n\}$, where every predicate in $T_1.P$ is transformable from T_1 to T_2. We define $T_1.P[T_2/T_1] = \{T_1.p_1[T_2/T_1],\ T_1.p_2[T_2/T_1],\ \ldots,\ T_1.p_n[T_1/T_2]\}$

Similarly, if $T_2.p$ is a transformable predicate formula of theory T_2, $T_2.p[T_1/T_2]$ denotes $T_2.p\ [f^{\sim}_{v|(T_1,T_2)}\ (T_2.v_1)/T_2.v_1, f^{\sim}_{v|(T_1,T_2)}\ (T_2.v_2)/T_2.v_2, \ldots, f^{\sim}_{v|(T_1,T_2)}\ (T_2.v_n)/T_2.v_n]$.

Definition 6.19 A predicate p in T_1 is *non-transformable* to T_2 if not all free variables in $T_1.p$ have comparable variables in T_2. $T_1.p$ is *conditionally transformable* from T_1 to T_2, if $T_1.p$ becomes transformable after each non-comparable variable in it is assigned a "null" value to. For the set-valued variables, the "null" value is the empty set $\emptyset$.

One example of conditionally transformable predicates is the predicate "$definedIn = attrDefinedIn \cup opDefinedIn \cup roleDefinedIn$" in schema *OMFeatureBasic* (on p.58). Since variable *opDefinedIn* does not have a counterpart in *IM*, this predicate is not transformable to *IM*. If let $opDefinedIn = \emptyset$, the predicate becomes "$definedIn = attrDefinedIn \cup roleDefinedIn$", and hence, becomes transformable.

If a predicate of T_1 is transformable or conditionally transformable to T_2, then we may be able to compare this predicate to the predicates in T_2. If two predicate sets, each from one theory, impose similar constraints on comparable variables, then we compare them. These predicates are called *comparable*

*predicate*s. A *non-comparable predicate* describes properties specific to one theory and does not have a counterpart in the other theory.

We may compare a conditionally transformable predicate p in one theory with the predicates in the other theory under the condition that the non-comparable variables in the predicate all have a "null" value (e.g., $\emptyset$). Under this condition, the variable substitution can take place by ignoring the non-comparable variables.

Definition 6.20 Given two sets of comparable predicates, $T_1.P$ and $T_2.Q$, if

$$\begin{aligned} &\text{for every } p \in T_1.P, \quad T_2.Pred \models T_1.p[T_2/T_1] \quad \text{and} \\ &\text{for every } q \in T_2.Q, \quad T_1.Pred \models T_2.q[T_1/T_2] \end{aligned}$$

then we say the two sets of predicates are *equivalent*, denoted as $T_1.P \simeq_{sem} T_2.Q$.

Sometimes, when comparing $T_1.p$ and $T_2.q$, we may not directly obtain $T_1.Pred \models T_2.q[T_1/T_2]$ or $T_2.Pred \models T_1.p[T_2/T_1]$. However, if putting further constraints on free variables in $T_1.p$ or $T_2.q$, we may get the conclusion that $T_1.Pred \models T_2.q[T_1/T_2]$ or $T_2.Pred \models T_1.p[T_2/T_1]$ or both.

Definition 6.21 Given two sets of comparable predicates, $T_1.P$ and $T_2.Q$. If $T_1.P$ and $T_2.Q$ are equivalent under the condition that the restricted values are assigned to certain free variables in the predicate(s), then we say the two sets of predicates are *conditionally equivalent*, denoted as $T_1.P \sim_{sem} T_2.Q$.

Given two comparable predicates $T_1.p$ and $T_2.q$. It can be easily proven that

$$(T_1.p[T_2/T_1] = T_2.q) \Leftrightarrow (T_2.q[T_1/T_2] = T_1.p)$$

Even under the condition that certain variables in $T_1.p$ and $T_2.q$ take restricted values, the above formula still holds. Based on Definition 6.20 and

Definition 6.21, we have

$$(T_1.p[T_2/T_1] = T_2.q) \vee (T_2.q[T_1/T_2] = T_1.p) \Rightarrow \begin{cases} T_1.p \sim_{sem} T_2.q & \text{some variables in } T_1.p \text{ and } T_2.q \\ & \text{take restricted values} \\ T_1.p \simeq_{sem} T_2.q & \text{otherwise} \end{cases}$$

6.3 Characterizing the Similarities and Differences between the Methods

After comparing two theories T_1 and T_2 under the guidelines to be given in Section 6.4 and 6.5, as the result, we obtain a core and two extensions to the core.

6.3.1 Core

The *core* has properties common to both T_1 and T_2. It contains a subtheory of T_1, denoted as $Core_{T_1}$, and a subtheory of T_2, denoted as $Core_{T_2}$. $Core_{T_1}.Sig.Var$ and $Core_{T_2}.Sig.Var$ contain the variables except "pure" non-comparable variables of T_1 and T_2. The predicates in $Core_{T_1}.Pred$ and $Core_{T_2}.Pred$ place the same constraints on these variables. There is an isomorphic mapping between the models of $Core_{T_1}$ and $Core_{T_2}$:

$$\forall\, m^{Core_{T_1}} : M^{Core_{T_1}} \exists\, m^{Core_{T_2}} : M^{Core_{T_2}} \bullet m^{Core_{T_1}} \equiv_{sem} m^{Core_{T_2}}$$
$$\forall\, m^{Core_{T_2}} : M^{Core_{T_2}} \exists\, m^{Core_{T_1}} : M^{Core_{T_1}} \bullet m^{Core_{T_2}} \equiv_{sem} m^{Core_{T_1}}$$

According to Proposition 6.2, each model of T_1 restricted on $Core_{T_1}.Sig$ is a model of $Core_{T_1}$. Similarly, each model of T_2 restricted on $Core_{T_2}.Sig$ is a model of $Core_{T_2}$. However, some models of the core may not be able to be extended to a model of T_1 or T_2, because there may be additional restrictions

on T_1 or T_2 that prevent the extension. That is,

$$\{m^{T_1} : M^{T_1} \bullet m^{T_1} \uparrow Core_{T_1}.Sig\} \subseteq M^{Core_{T_1}}$$
$$\{m^{T_2} : M^{T_2} \bullet m^{T_2} \uparrow Core_{T_2}.Sig\} \subseteq M^{Core_{T_2}}$$

6.3.2 Extensions

The *extension* of each theory to the core contains properties specific to the theory. The extension of T_1 is denoted as Ext_{T_1}. It contains a subsignature and a set of predicates from T_1, denoted as $Ext_{T_1}.Sig$ and $Ext_{T_1}.Pred$ respectively. The variables in the extension are "pure" non-comparable variables. The predicates in Ext_{T_1} can be divided into two parts:

1. the predicates about the variables in Ext_{T_1}, denoted as $Ext_{T_1}.ecPred$, and

2. the predicates imposing further constraints on the variables in $Core_{T_1}$, denoted as $Ext_{T_1}.ccPred$.

A similar discussion applies to T_2.

The predicates in $Ext_{T_1}.ccPred$ (or $Ext_{T_2}.ccPred$) can be divided into three parts: $ccPred_{T_1}$, $ccPred_{T_2}$, and $ccPred_{T_1 \bullet T_2}$. $Ext_{T_1}.ccPred_{T_1}$ imposes more constraints on the comparable variables in the core than $Ext_{T_2}.ccPred_{T_1}$ does. Similarly, $Ext_{T_2}.ccPred_{T_2}$ imposes more constraints on the comparable variables in the core than $Ext_{T_1}.ccPred_{T_2}$ does. $Ext_{T_1}.ccPred_{T_1 \bullet T_2}$ and $Ext_{T_2}.ccPred_{T_1 \bullet T_2}$ contain comparable predicates that impose different constraints on comparable variables in the core.

A complete theory is a combination of the core and an extension. That is,

$$T_1 = Core_{T_1} \cup Ext_{T_1}$$
$$T_2 = Core_{T_2} \cup Ext_{T_2}$$

The Venn diagram in Figure 6.2 (a) demonstrates the relationships between the core and extensions of the two theories.

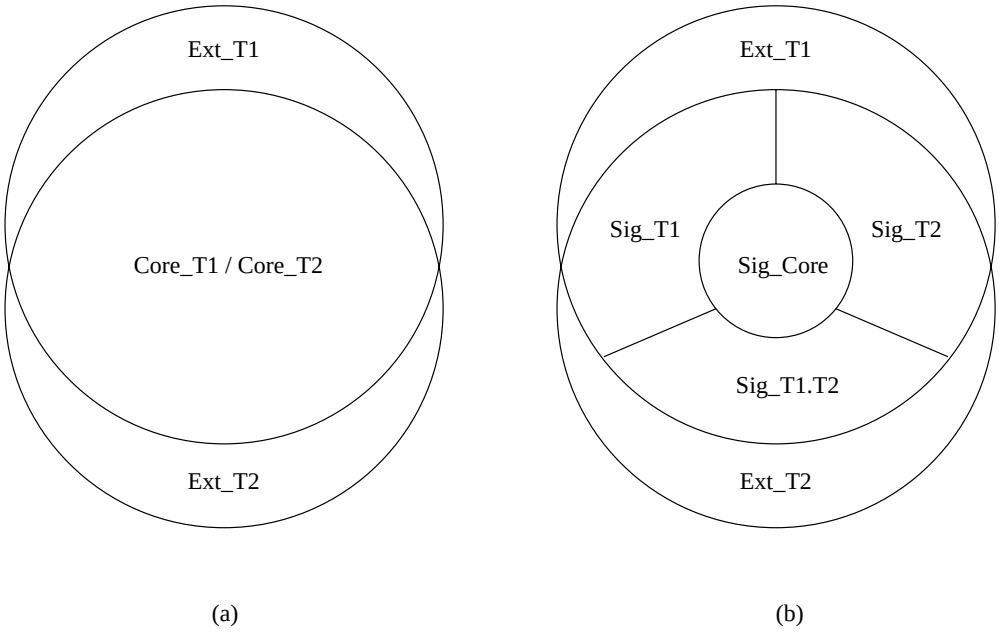


Figure 6.2: Venn Diagrams for relationships between core and extensions.

6.3.3 Characterizing Sets

The signature of $Core_{T_1}$ (or $Core_{T_2}$) can be divided into four parts: Sig_{Core}, Sig_{T_1}, Sig_{T_2}, and $Sig_{T_1 \bullet T_2}$. We call each of them a *characterizing set*. They are a partition of $Core_{T_1}.Sig$ (or $Core_{T_2}.Sig$). In the core, there are therefore four pairs of characterizing sets: ($Core_{T_1}.Sig_{Core}$, $Core_{T_2}.Sig_{Core}$), ($Core_{T_1}.Sig_{T_1}$, $Core_{T_2}.Sig_{T_1}$), ($Core_{T_1}.Sig_{T_2}$, $Core_{T_2}.Sig_{T_2}$), and ($Core_{T_1}.Sig_{T_1 \bullet T_2}$, $Core_{T_2}.Sig_{T_1 \bullet T_2}$). The variables defined in each pair of the signatures are comparable. The Venn diagram in Figure 6.2 (b) demonstrates that the signature in the core is partitioned by the characterizing sets. In practice, some of the characterizing sets may be empty depending on how similar the two theories are.

The meaning of the characterizing sets and their relationships to the predicates in $Ext_{T_1}.ccPred$ and $Ext_{T_2}.ccPred$ are described as follows.

$Core_{T_1}.Sig_{Core}$ and $Core_{T_2}.Sig_{Core}$:

Ext_{T_1} and Ext_{T_2} do not impose constraints on the variables in the two sets. Thus, given a model m^{T_1}, there is a model m^{T_2} so that $m^{T_1} \uparrow Core_{T_1}.Sig_{Core}$ and $m^{T_2} \uparrow Core_{T_2}.Sig_{Core}$ represent the same portion of a concrete design. The converse is also true. That is,

$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid m^{T_1} \uparrow Core_{T_1}.Sig_{Core} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{Core} \bullet m^{T_1}\} = M^{T_1}$$
$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid m^{T_1} \uparrow Core_{T_1}.Sig_{Core} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{Core} \bullet m^{T_2}\} = M^{T_2}$$

$Core_{T_1}.Sig_{T_1}$ and $Core_{T_2}.Sig_{T_1}$:

Ext_{T_1} imposes more constraints on the variables in $Core_{T_1}.Sig_{T_1}$ than Ext_{T_2} does on the corresponding variables in $Core_{T_2}.Sig_{T_1}$. The constraints are recorded in $Ext_{T_1}.ccPred_{T_1}$ and $Ext_{T_2}.ccPred_{T_1}$ respectively. Theory T_2 is said to be more general than theory T_1 in these aspects. Thus, a model m^{T_1} can always find a model m^{T_2} so that $m^{T_1} \uparrow Core_{T_1}.Sig_{T_1}$ and $m^{T_2} \uparrow Core_{T_2}.Sig_{T_1}$ represent the same portion of a concrete design. But the converse is not true. That is,

$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid m^{T_1} \uparrow Core_{T_1}.Sig_{T_1} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_1} \bullet m^{T_1}\} = M^{T_1}$$
$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid m^{T_1} \uparrow Core_{T_1}.Sig_{T_1} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_1} \bullet m^{T_2}\} \subset M^{T_2}$$

$Core_{T_1}.Sig_{T_2}$ and $Core_{T_2}.Sig_{T_2}$:

Ext_{T_2} imposes more constraints on the variables in $Core_{T_2}.Sig_{T_2}$ than Ext_{T_1} does on the corresponding variables in $Core_{T_1}.Sig_{T_2}$. The constraints are recorded in $Ext_{T_2}.ccPred_{T_2}$ and $Ext_{T_1}.ccPred_{T_2}$ respectively. Theory T_1 is said to be more general than theory T_2 in these aspects. Thus, a model m^{T_2} can always find a model m^{T_1} so that $m^{T_2} \uparrow Core_{T_2}.Sig_{T_2}$

and $m^{T_1} \uparrow Core_{T_1}.Sig_{T_2}$ represent the same portion of a concrete design. But the converse is not true. That is,

$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_2} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_2} \bullet m^{T_1}\} \subset M^{T_1}$$
$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_2} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_2} \bullet m^{T_2}\} = M^{T_2}$$

$Core_{T_1}.Sig_{T_1 \bullet T_2}$ and $Core_{T_2}.Sig_{T_1 \bullet T_2}$:

Ext_{T_1} imposes constraints, recorded in $Ext_{T_1}.ccPred_{T_1}$ and $Ext_{T_1}.ccPred_{T_1 \bullet T_2}$, on the variables in $Core_{T_1}.Sig_{T_1 \bullet T_2}$; and Ext_{T_2} imposes constraints, recorded in $Ext_{T_2}.ccPred_{T_2}$ and $Ext_{T_2}.ccPred_{T_1 \bullet T_2}$, on the variables in $Core_{T_2}.Sig_{T_1 \bullet T_2}$. The constraints imposed by Ext_{T_1} are different from those imposed by Ext_{T_2}. Thus, there must be some model m^{T_1}, but not every model, in M^{T_1}, that cannot find a model m^{T_2} so that $m^{T_1} \uparrow Core_{T_1}.Sig_{T_1 \bullet T_2}$ and $m^{T_2} \uparrow Core_{T_2}.Sig_{T_1 \bullet T_2}$ represent the same portion of a concrete design. The converse is also true. That is,

$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_1 \bullet T_2} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_1 \bullet T_2} \bullet m^{T_1}\} \subset M^{T_1}$$
$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_1 \bullet T_2} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_1 \bullet T_2} \bullet m^{T_2}\} \subset M^{T_2}$$

6.3.4 Semantically Equivalent Models

Given a model of T_1, if any of the variables in Ext_{T_1} is of value other than the empty set $\emptyset$, then the model does not have a semantically equivalent model of T_2.

The semantically equivalent models of the two theories are the models of the core:

$$\forall\, m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \bullet$$
$$m^{T_1} \equiv_{sem} m^{T_2} \Rightarrow m^{T_1} \in M^{Core_{T_1}} \wedge m^{T_2} \in M^{Core_{T_2}}$$

The semantically equivalent models in T_1 are

$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_1} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_1} \land$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_1 \bullet T_2} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_1 \bullet T_2} \bullet m^{T_1}\}$$

Similarly, the semantically equivalent models in T_2 are

$$\{m^{T_1} : M^{T_1};\ m^{T_2} : M^{T_2} \mid$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_2} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_2} \land$$
$$m^{T_1} \uparrow Core_{T_1}.Sig_{T_1 \bullet T_2} \equiv_{sem} m^{T_2} \uparrow Core_{T_2}.Sig_{T_1 \bullet T_2} \bullet m^{T_2}\}$$

6.4 Comparison Process

The concepts and categories of a theory are usually closely related to each other. If comparing the related concepts all at once, we have to consider almost the entire theory, which makes the comparison difficult to manage.

The way we organize a theory into schemas provides a basis for the comparison to proceed incrementally. In this way, we can make the comparison better organized and more understandable.

A theory is composed of a series of subtheories, which are presented in a series of Z schemas. When formalizing the method, we begin with the schemas with some basic concepts and categories. Based on the established schemas, we then introduce new schemas that may include some new concepts, relations between the concepts, and constraints on the concepts and the relations. Each newly introduced schema usually includes one or more established schemas. Each schema is regarded as a subtheory.

The comparison may be accomplished under a stepwise strategy. Initially, the core and extensions are empty. They are extended as the comparison continues.

At each step, we choose one or more schemas from each theory to compare. We do not take arbitrary schemas; they are intentionally constructed. The

chosen schemas should describe the same or similar aspect of the theories. In accordance with the order in which the concepts are formalized in a theory, a comparison begins with the design-time concepts and predicates, then moves on to run-time ones.

At each step, we need to determine the comparability between newly introduced variables and predicates. We may add auxiliary variables or change the way some variables are formalized when necessary, and accordingly, add or change the related predicates. If a variable or a predicate in the chosen schema of one theory does not have a comparable counterpart in the chosen schema of the other theory, but has a comparable counterpart in other schemas of the other theory, we split the schema into two or more parts. The comparison of these schemas then proceeds in several steps.

We do not consider the schemas that are just the combinations of other schemas, e.g. $OMName \mathrel{\widehat{=}} OMEntityName \wedge OMFeatureName$, because they do not introduce new variables and predicates, and thus do not give any new information.

At each step, guided by the decision rules given in Section 6.5, we compare the newly introduced variables and predicates, and classify them into the core or extensions.

The comparison repeats the above steps until all the schemas are compared.

6.5 Decision Rules

The decision rules serve as criteria during the comparison in deciding whether a part of a theory should belong to the core or an extension.

The rules are generalized from the analyses of the theories for OMT's object model and SMOOA's information model. They are general enough to be applied to comparing other design methods that embody similar modeling techniques. Should we encounter any circumstances that none of the rules could be applied to, we make decision based on the definitions of core and

extension.

The characterizing sets reveal the more detailed differences between the two theories. The rules for dividing the signatures in the core into the characterizing sets are presented in Appendix B.

6.5.1 Rules for Variables

1. The comparable variables belong to the core.

2. A "pure" non-comparable variable of T discussed in Section 6.2.3 belongs to Ext_T.

3. Let $T.v$ be a "redundant" non-comparable variable which can be replaced by a non-redundant variable $T.w$. $T.v$ belongs to the core or to the extension as $T.w$ does.

4. Let $T_1.v$ be an "explicitly defined" or "mistyped" non-comparable variable. We can create auxiliary variables in T_2 or change the declaration of $T_1.v$ to make $T_1.v$ comparable. The comparison is then guided by rule 1 and rule 2.

5. An "underlying" non-comparable variable belongs to the core.

6. A "subsumed" non-comparable variable belongs to the core or to the extension as the variable that "subsumes" it does.

7. A set of "technically different" non-comparable variables needs reformulating during the comparison. Whether these variables should belong to the core or to the extensions is determined by the new variables.

6.5.2 Rules for Comparable Predicates

8. If two sets of comparable predicates are transformable and are equivalent, then they belong to the core.

9. Given two sets of comparable predicates $T_1.P$ and $T_2.Q$. If they are transformable, and if

   - $T_1.Pred \models T_2.Q[T_1/T_2]$ but $T_2.Pred \not\models T_1.P[T_2/T_1]$, then

     $$T_1.P \subseteq Ext_{T_1}.ccPred_{T_1} \quad \text{and} \quad T_2.Q[T_1/T_2] \subset Core_{T_1}$$
     $$T_2.Q \subset Core_{T_2}$$

   - $T_2.Pred \models T_1.P[T_2/T_1]$ but $T_1.Pred \not\models T_2.Q[T_1/T_2]$, then

     $$T_1.P \subset Core_{T_1}$$
     $$T_2.Q \subseteq Ext_{T_2}.ccPred_{T_2} \quad \text{and} \quad T_1.P[T_2/T_1] \subset Core_{T_2}$$

10. Given two sets of comparable predicates $T_1.P$ and $T_2.Q$.

    - If they are conditionally transformable with $T_1.P$ having non-comparable variables, and $T_1.P \sim_{sem} T_2.Q$ or $T_2.Pred \models T_1.P[T_2/T_1]$, then

      $$T_1.P \subseteq Ext_{T_1}.ccPred_{T_2}$$
      $$T_2.Q \subseteq Ext_{T_2}.ccPred_{T_2}$$

    - If they are conditionally transformable with $T_2.Q$ having non-comparable variables, and $T_1.P \sim_{sem} T_2.Q$ or $T_1.Pred \models T_2.Q[T_1/T_2]$, then

      $$T_1.P \subseteq Ext_{T_1}.ccPred_{T_1}$$
      $$T_2.Q \subseteq Ext_{T_2}.ccPred_{T_1}$$

11. Given two sets of comparable predicates $T_1.P$ and $T_2.Q$. If they do not satisfy the conditions for rule 9 and rule 10, then

    $$T_1.P \subseteq Ext_{T_1}.ccPred_{T_2 \bullet T_2}$$
    $$T_2.Q \subseteq Ext_{T_2}.ccPred_{T_1 \bullet T_2}$$

6.5.3 Rules for Non-comparable Predicates

12. If a non-comparable predicate $T_1.p$ (or $T_2.q$) is about the properties of the variables in $Core_{T_1}$ (or $Core_{T_2}$), then it belongs to $Ext_{T_1}.ccPred_{T_1}$ (or $Ext_{T_2}.ccPred_{T_2}$).

13. If a non-comparable predicate $T_1.p$ (or $T_2.q$) is about the properties of a "pure" non-comparable variable, then the predicate belongs to $Ext_{T_1}.ecPred$ (or $Ext_{T_2}.ecPred$).

14. If a non-comparable predicate is about how a "redundant" non-comparable variable can be replaced by a non-redundant variable, then the predicate belongs to the core or to the extension as the redundant variable does.

15. If a non-comparable predicate $T_1.p$ is about the properties of an "explicitly defined" or "mistyped" non-comparable variable, then the predicate may become comparable after adding auxiliary variables and predicates in T_2 or changing the definition of the variable. In this case, the comparison follows rule 8 to rule 13.

16. If a non-comparable predicate $T_1.p$ is about the properties of an "underlying" non-comparable variable, we do not consider it non-comparable. In this case, $T_1.p$ belongs to the core.

17. If a non-comparable predicate $T_1.p$ is about the properties of a "subsumed" non-comparable variable, whether it should belong to the core or to the extension is determined by the non-comparable variable.

18. If a non-comparable predicate $T_1.p$ is about properties of a "technically different" non-comparable variable, we do not consider it non-comparable. In this case, the comparison proceeds after we reformulate the variables and predicates.

Chapter 7

Comparison of OMT and SMOOA

In this chapter, we discuss the comparison of the design-time theories of *OM* and *IM*. We extract the core from the two theories, and identify the extension to the core for each theory. Both *OM* and *IM* are subtheories with respect to the methods as a whole[1].

Section 7.1 defines the mapping between types. Section 7.2 presents the stepwise comparison of the two theories. Section 7.3 summarizes the comparison result.

7.1 Mapping between Types

Table 7.1 illustrates the mapping between the basic types and subsets of the basic types of *OM* and *IM*. Some of the types in one theory do not have comparable counterparts in the other theory. The comparability of other complex types that are constructed from the basic types is determined according to Definition 6.16 in Chapter 6.

[1]The complete theory for OMT also includes the theories for dynamic model, functional model, and other representational aspects of the method. It is the same with SMOOA.

7.2 Comparing the Two Theories

For convention, we define some simplified notations to be used in the comparison.

1. Each schema is regarded as a subtheory. We take the schema name as the subtheory name.

2. For a variable v of theory OM, we use $OM.v$ to denote it, or simply use v if the context is clear. It is the same with a variable in IM. Furthermore, we use f_v as shorthand for $f_{v|(OM,IM)}$.

3. Given two schemas $subOM$ and $subIM$ from the two theories, $f_v \uparrow (subOM, subIM)$ denotes a set of comparable pairs, each of which is composed of a variable defined in $subOM$ and a variable in $subIM$. The union of all such sets is f_v. These variables belong to the core.

   The set $(subOM.Sig.Var \setminus \mathrm{dom}\, f_v)$ denotes the "pure" non-comparable variables defined in $subOM$. These variables belong to Ext_{OM}. Similarly, $(subIM.Sig.Var \setminus \mathrm{ran}\, f_v)$ denotes the "pure" non-comparable variables defined in $subIM$. These variables belong to Ext_{IM}.

4. To each predicate in a schema, we give a numerical subscript according to the order it appears in the schema. A predicate p of schema $subOM$ of OM is denoted as $subOM.p$ or simply $OM.p$ if the context is clear. It is the same with a predicate in IM.

5. We define a transformation function from the predicates of OM to the predicates of IM. Given two subtheories $subOM$ and $subIM$, and a predicate $subOM.p$ which is transformable from OM to IM, the transformation function τ is defined as follows.

   $$\tau(subOM.p) \mathrel{\widehat{=}} p[subIM/subOM]$$

$\tau(subOM.p)$ is a predicate of $subIM.Sig$. If the scope of discussion is in certain subtheory, we can simply use $\tau(OM.p)$.

Similarly, $\tau^{\sim}$ is defined as a transformation function from the predicates of IM to the predicates of OM. For a transformable predicate $subIM.p$,

$$\tau^{\sim}(subIM.p) \mathrel{\widehat{=}} p[subOM/subIM]$$

The transformation function can be applied to a conditionally transformable predicate after the non-comparable variables in the predicate take "null" values.

Each of the following subsections compares one or more schemas from each theory. Every schema to be compared is accompanied with the page number which refers to where the schema is defined in the dissertation. At each comparison step, two parts may be presented: variable and predicate. The variable part lists the pairs of the comparable variables as well as the non-comparable variables. The predicate part discusses the predicates that impose constraints on the variables.

7.2.1 Basic Design Elements

OMDTData (p.51) **vs.** *IMDTData* (p.118) **:**

Variable:

$$f_v \uparrow (OMDTData, IMDTData) = \{(entity, entity), (class, object), (association, relationship), (feature, feature), (attribute, attribute), (role, role)\}$$

$$OMDTData.Sig.Var \setminus \mathrm{dom}\, f_v = \{operation\}$$

$$IMDTData.Sig.Var \setminus \mathrm{ran}\, f_v = \emptyset$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

$$\begin{aligned}\tau(OM.p_1) &= \tau(OM.class \neq \emptyset) \\ &= (IM.object \neq \emptyset) \\ &= IM.p_1\end{aligned}$$

According to rule 8,

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_2}$:

$$\begin{aligned}\tau(OM.p_2) &= \tau(\langle OM.class, OM.association\rangle \text{ partition } OM.entity) \\ &= (\langle IM.object, IM.relationship\rangle \text{ partition } IM.entity) \\ &= IM.p_2\end{aligned}$$

According to rule 8,

$$OM.p_2 \in Core_{OM} \quad \text{and} \quad IM.p_2 \in Core_{IM}$$

$\underline{OM.p_3 \text{ vs. } IM.p_3}$:

$$\tau(OM.p_3) = \tau(\langle OM.attribute, OM.operation, OM.role\rangle \text{ partition } OM.feature)$$

When $OM.operation = \emptyset$, we have

$$\begin{aligned}\tau(OM.p_3) &= \tau(\langle OM.attribute, OM.role\rangle \text{ partition } OM.feature) \\ &= (\langle IM.attribute, IM.role\rangle \text{ partition } IM.feature) \\ &= IM.p_3\end{aligned}$$

Otherwise, $\tau(OM.p_3) \neq IM.p_3$. According to rule 10,

$$OM.p_3 \in Ext_{OM}.ccPred_{IM} \quad \text{and} \quad IM.p_3 \in Ext_{IM}.ccPred_{IM}$$

7.2.2 Attributes

OMAttribute (p.52) **vs.** *IMAttributeBasic* (p.119) **:**

Variable:

In *IMAttributeBaisc*, variable *hasAttribute* is redundant because $hasAttribute = attrDefinedIn^{\sim}$. Therefore, we only consider $OM.attrDefinedIn$ and $IM.attrDefinedIn$.

$$f_v \uparrow (OMAttribute, IMAttributeBasic) = \{(attrDefinedIn, attrDefinedIn)\}$$

According to rule 3,

$$IM.hasAttribute \in Core_{IM}$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

$$\begin{aligned}\tau(OM.p_1) &= \tau(\text{dom } OM.attrDefinedIn = OM.attribute) \\ &= (\text{dom } IM.attrDefinedIn = IM.attribute) \\ &= IM.p_1\end{aligned}$$

According to rule 8,

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_2}$:

$$\begin{aligned}\tau(OM.p_2) &= \tau(\text{ran } OM.attrDefinedIn \subseteq OM.entity) \\ &= (\text{ran } IM.attrDefinedIn \subseteq IM.entity) \\ IM.p_2 &= (\text{ran } IM.attrDefinedIn = IM.object)\end{aligned}$$

We can prove that

$$IM.Pred \models \tau(OM.p_2)$$

Proof. From $IMDTData.p_2$, we have

$$IM.object \subseteq IM.entity$$

so

$$\begin{aligned} IM.p_2 &= (\text{ran}\, IM.attrDefinedIn = IM.object) \\ &\Rightarrow (\text{ran}\, IM.attrDefinedIn \subseteq IM.entity) \\ &= \tau(OM.p_2) \end{aligned}$$

Therefore,

$$IMDTData.p_2, IM.p_2 \models \tau(OM.p_2) \qquad \blacksquare$$

On the other hand, $IM.p_2$ does not always hold when $\tau(OM.p_2)$ is true. That is, $OM.Pred \not\models \tau^{\sim}(IM.p_2)$. According to rule 9,

$$OM.p_2 \in Core_{OM} \quad \text{and} \quad IM.p_2 \in Ext_{IM}.ccPred_{IM} \quad \text{and} \quad \tau(OM.p_2) \in Core_{IM}$$

$\underline{IM.p_3}$:

$IM.p_3$ is the predicate about the "redundant" non-comparable variable $IM.hasAttribute$. Since the variable is in $Core_{IM}$, according to rule 14, the predicate belongs to $Core_{IM}$.

IMAttributeType (p.119) :

SMOOA introduces three types of attributes in the information model. In contrast, OMT does not discuss specific types of attributes. However, attributes in the object model can also be naming or descriptive. We create an auxiliary schema for OM that describes the attribute types:

auxOMAttributeType

$OMAttribute$

$descriptiveAttr : \mathbb{P}\, ATTRIBUTE$

$namingAttr : \mathbb{P}\, ATTRIBUTE$

$\langle descriptiveAttr, namingAttr \rangle$ partition $attribute$

Now we can compare *IMAttributeType* and *auxOMAttributeType*.

Variable:

$$f_v \uparrow (auxOMAttributeType, IMAttributeType) = \{(descriptiveAttr, descriptiveAttr), (namingAttr, namingAttr)\}$$
$$auxOMAttributeType.Sig.Var \setminus \mathrm{dom}\, f_v = \emptyset$$
$$IMAttributeType.Sig.Var \setminus \mathrm{ran}\, f_v = \{referentialAttr\}$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

$$\begin{aligned}\tau(OM.p_1) &= \tau(\langle OM.descriptiveAttr, OM.namingAttr\rangle \text{ partition } OM.attribute) \\ &= (\langle IM.descriptiveAttr, IM.namingAttr\rangle \text{ partition } IM.attribute) \\ &= IM.p_1\end{aligned}$$

According to rule 8,

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{IM.p_2}$:

$IM.p_2$ specifies the characteristic of $IM.referentialAttr$. According to rule 13,

$$IM.p_2 \in Ext_{IM}.ecPred$$

7.2.3 Operations and Methods

$OM.operation$ belongs to Ext_{OM}, so do the variables defined in *OMOperationProperty* (p.53), *OMOperation* (p.54), and *OMOperationType*1 (p.54). According to rule 2,

$$\{OM.argumentOfOp, OM.resultOfOp, OM.opDefinedIn, OM.queryOp, OM.updateOp\} \subset Ext_{OM}$$

According to rule 13, the predicates defined in these schemas belong to $Ext_{OM}.ecPred$.

Method is a concept specific to OM, which is formalized in schemas *OMMethodProperty* (p.60), and *OMMethod* (p.60). The variables defined in the schemas belong to Ext_{OM}:

$$\{OM.method, OM.argumentOfMethod, OM.resultOfMethod,$$
$$OM.implement, OM.methodLookup\} \subset Ext_{OM}$$

The predicates defined in the schemas belong to $Ext_{OM}.ecPred$.

7.2.4 Roles

OMRoleProperty (p.55) **vs.** *IMRoleProperty* (p.125) **:**
Variable:

$$f_v \uparrow (OMRoleProperty, IMRoleProperty) =$$
$$\{(playerOfRole, playerOfRole), (multiplicityOfRole, multiplicityOfRole)\}$$

Predicate:
$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

$$\begin{aligned}\tau(OM.p_1) &= \tau(OM.playerOfRole \in OM.role \rightarrow OM.class)\\ &= (IM.playerOfRole \in IM.role \rightarrow IM.object)\\ &= IM.p_1\end{aligned}$$

According to rule 8,

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_2}$:

$$\begin{aligned}\tau(OM.p_2) &= \tau(\text{dom } OM.multiplicityOfRole = OM.role)\\ &= (\text{dom } IM.multiplicityOfRole = IM.role)\\ &= IM.p_2\end{aligned}$$

According to rule 8,

$$OM.p_2 \in Core_{OM} \quad \text{and} \quad IM.p_2 \in Core_{IM}$$

OMRole (p.56) **vs.** *IMRoleBasic* (p.123) **:**

Variable:

$$f_v \uparrow (OMRole, IMRoleBasic) = \{(roleDefinedIn, roleDefinedIn), (hasRole, hasRole)\}$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

$$\begin{aligned}\tau(OM.p_1) &= \tau(OM.roleDefinedIn \in OM.role \twoheadrightarrow OM.association) \\ &= (IM.roleDefinedIn \in IM.role \twoheadrightarrow IM.relationship) \\ &= IM.p_1\end{aligned}$$

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_2}$:

$$\begin{aligned}\tau(OM.p_2) &= \tau(OM.hasRole = OM.roleDefinedIn^{\sim}) \\ &= (IM.hasRole = IM.roleDefinedIn^{\sim}) \\ &= IM.p_2\end{aligned}$$

$$OM.p_2 \in Core_{OM} \quad \text{and} \quad IM.p_2 \in Core_{IM}$$

$\underline{OM.p_3 \text{ vs. } IM.p_3}$:

$$\begin{aligned}\tau(OM.p_3) &= \tau(\forall\, a : OM.association \bullet \#(OM.hasRole(\!| \{a\} |\!)) \geq 2) \\ &= (\forall\, rel : IM.relationship \bullet \#(IM.hasRole(\!| \{rel\} |\!)) \geq 2) \\ &= IM.p_3\end{aligned}$$

$$OM.p_3 \in Core_{OM} \quad \text{and} \quad IM.p_3 \in Core_{IM}$$

OMRoleType (p.57) **vs.** *IMRoleType* (p.126) **:**

Roles are classified into three types in OM and four types in IM. In fact, the members of $OM.multipleRole$ can be further divided into two sets. We add two auxiliary variables and two predicates in $OMRoleType$:

$$
\begin{array}{|l}
onemulRole : \mathbb{P}\, ROLE \\
optmulRole : \mathbb{P}\, ROLE \\
\hline
\forall\, m : onemulRole \bullet 0 \notin multiplicityOfRole(\!|\, \{m\} \,|\!) \wedge \\
\qquad\qquad (\exists\, n : multiplicityOfRole(\!|\, \{m\} \,|\!) \bullet n > 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (OMRoleType.p_{add1}) \\
\forall\, om : optmulRole \bullet 0 \in multiplicityOfRole(\!|\, \{om\} \,|\!) \wedge \\
\qquad\qquad (\exists\, n : multiplicityOfRole(\!|\, \{om\} \,|\!) \bullet n > 1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (OMRoleType.p_{add2}) \\
\end{array}
$$

We need to prove that an element of $multipleRole$ is either an element of $onemulRole$ or an element of $optmulRole$, that is,

$$\langle onemulRole, optmulRole \rangle \ \mathsf{partition}\ multipleRole$$

This is equal to the following two formulas:

(1). $onemulRole \cup optmulRole = multipleRole$
(2). $onemulRole \cap optmulRole = \emptyset$

We now prove the above two formulas hold.

Proof.

(1).

$$
\begin{array}{rl}
OM.p_{add1} = & (\forall\, m : onemulRole \bullet 0 \notin multiplicityOfRole(\!|\, \{m\} \,|\!) \wedge \\
& \qquad (\exists\, n : multiplicityOfRole(\!|\, \{m\} \,|\!) \bullet n > 1)) \\
\Rightarrow & (\forall\, m : onemulRole \bullet (\exists\, n : multiplicityOfRole(\!|\, \{m\} \,|\!) \bullet n > 1)) \\
OM.p_{add2} = & (\forall\, om : optmulRole \bullet 0 \in multiplicityOfRole(\!|\, \{om\} \,|\!) \wedge \\
& \qquad (\exists\, n : multiplicityOfRole(\!|\, \{om\} \,|\!) \bullet n > 1)) \\
\Rightarrow & (\forall\, om : optmulRole \bullet (\exists\, n : multiplicityOfRole(\!|\, \{om\} \,|\!) \bullet n > 1))
\end{array}
$$

Comparing the above two formulas to $OM.p_4$, we see that any element of *onemulRole* or *optmulRole* is an element of *multipleRole*.

(2). The two sets *onemulRole* and *optmulRole* are defined to be different according to the difference of the multiplicity of their roles, described in $OM.p_{add1}$ and $OM.p_{add2}$. The multiplicity of any role in *optmulRole* contains "0", while the multiplicity of any role in *onemulRole* does not. Therefore, an element of one set cannot be an element of another set, and vice versa. ■

Variable:

We define

$$f_v \uparrow (OMRoleType, IMRoleType) = \{(singularRole, single), (optionalRole, conditionSingle), (onemulRole, multiple), (optmulRole, conditionMultiple)\}$$

$OM.onemulRole$ and $OM.optmulRole$ represent $OM.multipleRole$, and the two auxiliary variables are in the core. Therefore,

$$OM.multipleRole \in Core_{OM}$$

Predicate:

$$\begin{aligned}\tau(OM.p_1) &= \tau(\langle OM.singularRole, OM.optionalRole, \\ &\qquad OM.multipleRole\rangle \text{ partition } OM.role) \\ &= \tau(\langle OM.singularRole, OM.optionalRole, \\ &\qquad OM.onemulRole, OM.optmulRole\rangle \text{ partition } OM.role) \\ &= (\langle IM.single, IM.conditionSingle, \\ &\qquad IM.multiple, IM.conditionMultiple\rangle \text{ partition } IM.role) \\ &= IM.p_1\end{aligned}$$

Therefore,

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

Also, we can prove that

$$OM.p_2 \simeq_{sem} IM.p_2$$
$$OM.p_3 \simeq_{sem} IM.p_3$$
$$OM.p_{add1} \simeq_{sem} IM.p_4$$
$$OM.p_{add2} \simeq_{sem} IM.p_5$$

Therefore,

$$\{OM.p_2, OM.p_3, OM.p_{add1}, OM.p_{add2}\} \subset Core_{OM}$$
$$\{IM.p_2, IM.p_3, IM.p_4, IM.p_5\} \subset Core_{IM}$$

$OM.p_4$ is a combination of $OM.p_{add1}$ and $OM.p_{add2}$. Therefore,

$$OM.p_4 \in Core_{OM}$$

OMOrderedRole (p.64) **:**

The notation for ordered roles is the modeling technique specific to OMT. However, an ordered role is merely an element of *multipleRole*. If the two theories have a common model, there must be instances that correspond to both *orderedRole* in *OM* and *multiple* in *IM*. Therefore, *orderedRole* is an "explicitly defined" non-comparable variable. We can add to *IM* an auxiliary variable and an auxiliary predicate similar to the ones in *OMOrderedRole*.

The variable *OM.orderedRole* and the predicate defined in *OMOrderedRole* belong to $Core_{OM}$.

7.2.5 Features

OMFeatureBasic (p.58) **vs.** *IMFeature* (p.127) **:**

OMFeatureBasic has two variables but *IMFeature* has one, where *OM.definedIn* is comparable to $IM.hasFeature^{\sim}$, and *OM.hasFeature* has different semantics from *IM.hasFeature*. We add one variable to *IMFeature*,

$$definedIn : FEATURE \leftrightarrow ENTITY$$

$$definedIn = hasFeature^{\sim} \qquad (IMFeature.p_{add})$$

Even though $OM.hasFeature$ and $IM.hasFeature$ do not have the same meaning, they are still defined to be comparable because a model of IM restricted to $IM.hasFeature$ is a subset of the corresponding model of OM restricted to $OM.hasFeature$.

Variable:

$$f_v \uparrow (OMFeatureBasic, IMFeature) = \{(definedIn, definedIn), (hasFeature, hasFeature)\}$$

Predicate:

$\underline{OM.p_1 \text{ and } OM.p_3 \text{ vs. } IM.p_{add}}$:

$$\begin{aligned} IM.p_{add} &= (IM.definedIn = IM.hasFeature^{\sim}) \\ &= (IM.definedIn = IM.hasAttribure^{\sim} \cup IM.hasRole^{\sim}) \\ &= (IM.definedIn = IM.attrDefinedIn \cup IM.roleDefinedIn) \\ OM.p_1 &= (OM.definedIn = OM.attrDefinedIn \cup \\ &\qquad OM.opDefinedIn \cup OM.roleDefinedIn) \end{aligned}$$

$OM.opDefinedIn$ is a non-comparable variable. When $OM.opDefinedIn = \emptyset$, we have

$$IM.p_{add} \sim_{sem} OM.p_1$$

According to rule 10,

$$OM.p_1 \in Ext_{OM}.ccPred_{IM} \quad \text{and} \quad IM.p_{add} \in Ext_{IM}.ccPred_{IM}$$

$$\begin{aligned}
\tau(OM.p_3) &= \tau(OM.definedIn^{\sim} \subseteq OM.hasFeature) \\
&= (IM.definedIn^{\sim} \subseteq IM.hasFeature) \\
IM.p_{add} &= (IM.definedIn = IM.hasFeature^{\sim}) \\
&= (IM.definedIn^{\sim} = IM.hasFeature) \\
&\Rightarrow (IM.definedIn^{\sim} \subseteq IM.hasFeature) \\
&= \tau(OM.p_3)
\end{aligned}$$

On the other hand, we have $OM.Pred \not\models \tau^{\sim}(IM.p_{add})$. According to rule 9,

$$OM.p_3 \in Core_{OM} \quad \text{and} \quad IM.p_{add} \in Ext_{IM}.ccPred_{IM} \quad \text{and} \quad \tau(OM.p_3) \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_1}$:

$$\begin{aligned}
\tau(OM.p_2) &= \tau(OM.hasFeature \in OM.entity \leftrightarrow OM.feature) \\
&= (IM.hasFeature \in IM.entity \leftrightarrow IM.feature) \\
IM.p_1 &= (IM.hasFeature = IM.hasAttribute \cup IM.hasRole) \\
&= (IM.hasFeature = IM.attrDefinedIn^{\sim} \cup IM.roleDefinedIn^{\sim}) \\
&\Rightarrow (IM.hasFeature \in IM.object \leftrightarrow IM.attribute \cup \\
&\qquad\qquad IM.relationship \leftrightarrow IM.role) \\
&\Rightarrow (IM.hasFeature \in IM.entity \leftrightarrow IM.feature) \\
&= \tau(OM.p_2)
\end{aligned}$$

On the other hand, we have $OM.Pred \not\models \tau^{\sim}(IM.p_1)$. Therefore,

$$OM.p_2 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Ext_{IM}.ccPred_{IM} \quad \text{and} \quad \tau(OM.p_2) \in Core_{IM}$$

7.2.6 More Schemas about Attributes

OMAttributeRange (p.91) **vs.** *IMDomainSpecification* (p.120) **:**
Variable:

$$f_v \uparrow (OMAttributeRange, IMDomainSpecification) = \{(valueRange, domainSpec)\}$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

From $IMFeature.p_1$:

$$\begin{aligned} IM.hasFeature &= IM.hasAttribute \cup IM.hasRole \\ &= IM.attrDefinedIn^{\sim} \cup IM.roleDefinedIn^{\sim} \\ &= (IM.attribute \leftrightarrow IM.object)^{\sim} \cup (IM.role \twoheadrightarrow IM.relationship)^{\sim} \end{aligned}$$

we have

$$IM.hasFeature \rhd IM.attribute = IM.hasAttribute$$

and

$$\begin{aligned} \mathrm{dom}(IM.hasFeature \rhd IM.attribute) &= \mathrm{dom}\, IM.hasAttribute \\ &= \mathrm{ran}\, IM.attrDefinedIn \\ &= IM.object \end{aligned}$$

Therefore,

$$\begin{aligned} \tau^{\sim}(IM.p_1) &= \tau^{\sim}(\mathrm{dom}\, IM.domainSpec = IM.object) \\ &= \tau^{\sim}(\mathrm{dom}\, IM.domainSpec = \mathrm{dom}(IM.hasFeature \rhd IM.attribute)) \\ &= (\mathrm{dom}\, OM.valueRange = \mathrm{dom}(OM.hasFeature \rhd OM.attribute)) \\ &= OM.p_1 \end{aligned}$$

According to rule 8,

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_2}$:

$$\begin{aligned} \tau^{\sim}(IM.p_2) &= \tau^{\sim}(\forall\, obj : IM.object \bullet \\ &\qquad \mathrm{dom}(IM.domainSpec(obj)) = IM.hasAttribute(\!|\, \{obj\} \,|\!)) \\ &= \tau^{\sim}(\forall\, obj : \mathrm{dom}\, IM.domainSpec \bullet \\ &\qquad \mathrm{dom}(IM.domainSpec(obj)) = IM.hasFeature(\!|\, \{obj\} \,|\!) \cap IM.attribute) \\ &= (\forall\, e : \mathrm{dom}\, OM.valueRange \bullet \\ &\qquad \mathrm{dom}(OM.valueRange(e)) = OM.hasFeature(\!|\, \{e\} \,|\!) \cap OM.attribute) \\ &= OM.p_2 \end{aligned}$$

Therefore,

$$OM.p_2 \in Core_{OM} \quad \text{and} \quad IM.p_2 \in Core_{IM}$$

$\underline{OM.p_3}$:

The last predicate in schema *OMAttributeRange* is about the values that each attribute may take on under the constraint rules. They involve global variable *OM.valueOfType* (p.52), and schemas *OMAttributeProperty* (p.52), *OMGenAttrRestriction* (p.88), *OMAttributeConstraintBasic* (p.89), and *OMAttributeConstraint* (p.90). These schemas are used to describe which rules are applied to which attributes. Although SMOOA does not have "rules" on attributes, we assume that the similar rules implicitly apply when the atomic values are assigned to each attribute described in relation variable *IM.domainSpec*. Therefore, in this aspect, the two methods are equivalent. The variables defined in these schemas are "underlying" non-comparable variables. According to rule 5,

$$\{OM.valueOfType, OM.typeOfAttr, OM.restrict, \\ OM.generalRestrict, OM.constraint\} \subset Core_{OM}$$

According to rule 16, $OMAttributeRange.p_3$, the global predicate for *OM.valueOfType*, and the predicates defined in the above four schemas belong to $Core_{OM}$.

Default Values

Attribute default value is a concept specific to *OM*, which is formalized in schemas *OMDefaultValueBasic* (p.59) and *OMDefaultValue* (p.91). The variable
OM.defaultValOfAttr defined in the schemas belongs to Ext_{OM}; the predicates defined in the schemas belong to $Ext_{OM}.ecPred$.

7.2.7 Keys and Identifiers

OMCandidateKeyBasic (p.96) **vs.** *IMIdentifierBasic* (p.121) **:**

Variable:

$$f_v \uparrow (OMCandidateKeyBasic, IMIdentifierBasic) = \{(key, identifier), (hasKeyElement, hasIdElement)\}$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

$$\begin{aligned}\tau(OM.p_1) &= \tau(\text{dom } OM.hasKeyElement = OM.key) \\ &= (\text{dom } IM.hasIdElement = IM.identifier) \\ &= IM.p_1\end{aligned}$$

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_2}$:

$$\begin{aligned}\tau(OM.p_2) &= \tau(\text{ran } OM.hasKeyElement \subseteq OM.feature) \\ &= (\text{ran } IM.hasIdElement \subseteq IM.feature)\end{aligned}$$

Since $IM.attribute \subseteq IM.feature$, which is derived from $IMDTData.p_2$,

$$\begin{aligned}IM.p_2 &= (\text{ran } IM.hasIdElement \subseteq IM.attribute) \\ &\Rightarrow (\text{ran } IM.hasIdElement \subseteq IM.feature)\end{aligned}$$

We have

$$IMDTData.p_2, IM.p_2 \models \tau(OM.p_2)$$

but $OM.Pred \not\models \tau^{\sim}(IM.p_2)$. According to rule 9,

$$OM.p_2 \in Core_{OM} \quad \text{and} \quad IM.p_2 \in Ext_{IM}.ccPred_{IM} \quad \text{and} \quad \tau(OM.p_2) \in Core_{IM}$$

OMCandidateKey (p.97) **vs.** *IMIdentifier* (p.122) **:**

Variable $OM.keyedEntity$ is an auxiliary variable. It is introduced to denote the entities that have keys. Similarly in IM, we can also introduce such

a variable, say $IM.idedEntity$, which obviously equals to $IM.object$. We add the following variable to $IMIdentifier$:

$$idedEntity : \mathbb{P}\, ENTITY$$
$$idedEntity = object \qquad (IMIdentifier.p_{add})$$

A preferred identifier is merely an identifier in IM. We assume each class that has keys in OM has a "preferred key", which is one of the keys of the class. We add the following variable to $OMCandidateKey$:

$$hasPreferredKey : CLASS \rightarrowtail KEY$$
$$hasPreferredId \in keyedEntity \rightarrowtail key \qquad (OMCandidateKey.p_{add1})$$
$$\forall\, c : class \bullet isKeyOf(hasPreferredKey(c)) = c \qquad (OMCandidateKey.p_{add2})$$

Variable:

$$f_v \uparrow (OMCandidateKey, IMIdentifier) = \{(isKeyOf, isIdOf), (keyedEntity, idedEntity), (hasPreferredKey, hasPreferredId)\}$$

$OM.theKey$ is redundant because $theKey = isKeyOf^{\sim}$. According to rule 3,

$$OM.theKey \in Core_{OM}$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_{add}}$:

$$\tau(OM.p_1) = \tau(OM.keyedEntity \subseteq OM.entity) = (IM.idedEntity \subseteq IM.entity)$$
$$IM.p_{add} = (IM.idedEntity = IM.object)$$

The result of the comparison of these two predicates is the same with that of the comparison of $OMAttribute.p_2$ and $IMAttributeBasic.p_2$:

$$IMDTData.p_2, IM.p_{add} \models \tau(OM.p_1)$$
$$OM.Pred \not\models \tau^{\sim}(IM.p_{add})$$

Therefore,

$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_{add} \in Ext_{IM}.ccPred_{IM} \quad \text{and} \quad \tau(OM.p_1) \in Core_{IM}$$

$\underline{OM.p_2 \text{ vs. } IM.p_1}$:

$$\begin{aligned}\tau(OM.p_2) &= \tau(OM.isKeyOf \in OM.key \twoheadrightarrow OM.keyedEntity)\\ &= (IM.isIdOf \in IM.identifier \twoheadrightarrow IM.idedEntity)\\ &= (IM.isIdOf \in IM.identifier \twoheadrightarrow IM.object)\\ &= IM.p_1\end{aligned}$$
$$OM.p_1 \in Core_{OM} \quad \text{and} \quad IM.p_1 \in Core_{IM}$$

$\underline{OM.p_3}$:

$OM.p_3 : OM.theKey = OM.isKeyOf^{\sim}$ belongs to $Core_{OM}$.

$\underline{OM.p_4 \text{ and } IM.p_2}$:

These two predicates further specify the properties of $OM.hasKeyElement$ and
$IM.hadIdElement$.

From $IM.p_1 : isIdOf \in identifier \twoheadrightarrow object$, we can prove

$$isIdOf^{\sim}(\!| \ object \ |\!) = identifier$$

and therefore, given a predicate P,

$$\forall\, o : object \ \forall\, id : isIdOf^{\sim}(\!| \ \{o\} \ |\!) \bullet P(id) \Leftrightarrow \forall\, id : identifier \bullet P(id)$$

$IM.p_2$ can be changed to the following form:

$$IM.p_2 \quad = \quad (\forall\, id : identifier \bullet$$

$$
\begin{aligned}
& \mathit{hasIdElement}(\!|\ \{\mathit{id}\}\ |\!) \subseteq \mathit{hasAttribute}(\!|\ \{\mathit{isIdOf}(\mathit{id})\}\ |\!)))\\
= \quad & (\forall\, o : \mathit{object} \bullet\\
& \quad (\forall\, \mathit{id} : \mathit{isIdOf}^{\sim}(\!|\ \{o\}\ |\!) \bullet\\
& \quad\quad \mathit{hasIdElement}(\!|\ \{\mathit{id}\}\ |\!) \subseteq \mathit{hasAttribute}(\!|\ \{\mathit{isIdOf}(\mathit{id})\}\ |\!)))\\
= \quad & (\forall\, o : \mathit{idedEntity} \cap \mathit{object} \bullet\\
& \quad (\forall\, \mathit{id} : \mathit{isIdOf}^{\sim}(\!|\ \{o\}\ |\!) \bullet\\
& \quad\quad \mathit{hasIdElement}(\!|\ \{\mathit{id}\}\ |\!) \subseteq \mathit{hasFeature}(\!|\ \{o\}\ |\!)) \cap \mathit{attribute})
\end{aligned}
$$

$$
\begin{aligned}
OM.p_4 \quad = \quad & (\forall\, c : \mathit{keyedEntity} \cap \mathit{class} \bullet\\
& \quad (\forall\, k : \mathit{theKey}(\!|\ \{c\}\ |\!) \bullet\\
& \quad\quad \mathit{hasKeyElement}(\!|\ \{k\}\ |\!) \subseteq \mathit{attribute} \cap \mathit{hasFeature}(\!|\ \{c\}\ |\!)))\\
= \quad & (\forall\, c : \mathit{keyedEntity} \cap \mathit{class} \bullet\\
& \quad (\forall\, k : \mathit{isKeyOf}^{\sim}(\!|\ \{c\}\ |\!) \bullet\\
& \quad\quad \mathit{hasKeyElement}(\!|\ \{k\}\ |\!) \subseteq \mathit{attribute} \cap \mathit{hasFeature}(\!|\ \{c\}\ |\!)))
\end{aligned}
$$

It is obvious that $OM.p_4 \simeq_{sem} IM.p_2$. Therefore,

$$OM.p_4 \in Core_{OM} \quad \text{and} \quad IM.p_2 \in Core_{IM}$$

$\underline{OM.p_6 \text{ vs. } IM.p_3}$:

$$\tau(OM.p_6) = IM.p_3$$

Therefore,

$$OM.p_6 \in Core_{OM} \quad \text{and} \quad IM.p_3 \in Core_{IM}$$

$\underline{OM.p_5}$:

$OM.p_5$ is about properties of keys for associations and cannot be compared to any predicate in IM. It imposes the constraint on $OM.hasKeyElement$. According to rule 12,

$$OM.p_5 \in Ext_{OM}.ccPred_{OM}$$

$\underline{OM.p_{add1} \text{ vs. } IM.p_4 \text{ and } OM.p_{add2} \text{ vs. } IM.p_5}$:

We can prove that

$$OM.p_{add1} \simeq_{sem} IM.p_4$$
$$OM.p_{add2} \simeq_{sem} IM.p_5$$

Therefore,

$$\{OM.p_{add1}, OM.p_{add2}\} \subset Core_{OM}$$
$$\{IM.p_4, IM.p_5\} \subset Core_{IM}$$

Candidate Keys for Relationships

The predicates in schemas *OMm_mKey* (p.98), *OMone_mKey* (p.98), and *OMopt_oneKey* (p.99) are about candidate keys for relationships. They are specific to OM, and impose constraints on $OM.isKeyOf$ and $OM.hasKeyElement$, so they belong to $Ext_{OM}.ccPred_{OM}$.

7.2.8 Binary Associations and Relationships

OMBinaryAssociationBasic (p.66) **vs.** *IMBinaryRelationship* (p.128)**:**
Variable:

$$f_v \upharpoonright (OMBinaryAssociationBasic, IMBinaryRelationship) = \{(binaryAssociation, binaryRelationship)\}$$

Predicate:

$$\begin{aligned}\tau(OM.p_1) &= \tau(OM.binaryAssociation \subseteq OM.association)\\ &= IM.binaryRelationship \subseteq IM.relationship\\ &= IM.p_1\end{aligned}$$

$$\begin{aligned}\tau(OM.p_2) &= \tau(\forall\, a : OM.binaryAssociation \bullet \#(OM.hasRole(\!| \{a\} |\!)) = 2)\\ &= (\forall\, rel : IM.binaryRelationship \bullet \#(IM.hasRole(\!| \{rel\} |\!)) = 2)\\ &= IM.p_2\end{aligned}$$

The result is

$\{OM.p_1, OM.p_2\} \subset Core_{OM}$
$\{IM.p_1, IM.p_2\} \subset Core_{IM}$

OMBinaryAssociationType(p.68) **vs.** *IMBinaryRelationshipType*(p.129)**:**

Similar to "roles", "binary associations" in *OM* can be further divided into ten types, based on the types of their roles. We change *OMBinaryAssociationType* to the following schema:

auxOMBinaryAssociationType

$$
\begin{array}{l}
OMBinaryAssociationBasic \\
one\_oneAssoc, one\_onemulAssoc, one\_optmulAssoc : \mathbb{P}\, ASSOCIATION \\
onemul\_onemulAssoc, onemul\_optmulAssoc : \mathbb{P}\, ASSOCIATION \\
optmul\_optmulAssoc, one\_optAssoc, opt\_onemulAssoc : \mathbb{P}\, ASSOCIATION \\
opt\_optmulAssoc, opt\_optAssoc : \mathbb{P}\, ASSOCIATION \\
\hline
\langle one\_oneAssoc, one\_onemulAssoc, one\_optmulAssoc, onemul\_onemulAssoc, \\
\quad onemul\_optmulAssoc, optmul\_optmulAssoc, one\_optAssoc, opt\_onemulAssoc, \\
\quad opt\_optmulAssoc, opt\_optAssoc\rangle \ \mathsf{partition}\ binaryAssociation \\
\forall\, a : one\_oneAssoc \bullet (\exists\, r1, r2 : singularRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : one\_onemulAssoc \bullet \\
\quad (\exists\, r1 : singularRole;\ r2 : onemulRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : one\_optmulAssoc \bullet \\
\quad (\exists\, r1 : singularRole;\ r2 : optmulRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : onemul\_onemulAssoc \bullet (\exists\, r1, r2 : onemulRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : onemul\_optmulAssoc \bullet \\
\quad (\exists\, r1 : onemulRole;\ r2 : optmulRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : optmul\_optmulAssoc \bullet (\exists\, r1, r2 : optmulRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : one\_optAssoc \bullet \\
\quad (\exists\, r1 : singularRole;\ r2 : optionalRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : opt\_onemulAssoc \bullet \\
\quad (\exists\, r1 : optionalRole;\ r2 : onemulRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : opt\_optmulAssoc \bullet \\
\quad (\exists\, r1 : optionalRole;\ r2 : optmulRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\}) \\
\forall\, a : opt\_optAssoc \bullet (\exists\, r1, r2 : optionalRole \bullet hasRole(\!| \{a\} |\!) = \{r1, r2\})
\end{array}
$$

Schema *auxOMBinaryAssociationType* has the same semantics as schema *OMBinaryAssociationType*.

Variable:

$$f_v \uparrow (auxOMBinaryAssociationType, IMBinaryRelationshipType) =$$
$$\{(one\_oneAssoc, R_{11}), (one\_onemulAssoc, R_{1M}),$$
$$(one\_optmulAssoc, R_{1Mc}), (onemul\_onemulAssoc, R_{MM}),$$
$$(onemul\_optmulAssoc, R_{MMc}), (optmul\_optmulAssoc, R_{McMc}),$$
$$(one\_optAssoc, R_{11c}), (opt\_onemulAssoc, R_{1cM}),$$
$$(opt\_optmulAssoc, R_{1cMc}), (opt\_optAssoc, R_{1c1c})\}$$

We have,

$$OM.one\_mulAssoc = OM.one\_onemulAssoc \cup OM.one\_optmulAssoc$$
$$OM.mul\_mulAssoc = OM.onemul\_onemulAssoc \cup$$
$$OM.onemul\_optmulAssoc \cup OM.optmul\_optmulAssoc$$
$$OM.opt\_mulAssoc = OM.opt\_onemulAssoc \cup OM.opt\_optmulAssoc$$

Therefore,

$$\{OM.one\_mulAssoc, OM.mul\_mulAssoc, OM.opt\_mulAssoc\} \subset Core_{OM}$$

Predicate:

It can be proven that the predicates in *auxOMBinaryAssociationType* are equivalent to those in *IMBinaryRelationshipType*. Therefore, they belong to $Core_{OM}$ and $Core_{IM}$. The predicates in *OMBinaryAssociationType* also belong to $Core_{OM}$.

OMQualifiedAssociation (p.69) :

Schema *OMQualifiedAssociation* is about properties of qualified associations, which are merely binary associations. If a model for *OM* is also a model for *IM*, then the qualified associations in *OM* must have counterparts in *IM*. These binary relationships in *IM* must satisfy predicate $\tau(OMQualifiedAssociation.p_3)$. Although *IM* does not have this modeling concept, it still have the same expressive power as *OM* in this aspect. The variables in *OMQualifiedAssociation* are "explicitly defined" non-comparable

variables. Therefore, the variables as well as the predicates defined in this schema should be in $Core_{OM}$.

7.2.9 Derived Associations, Classes, and Attributes

OMAssociationType (p.101) **vs.** *IMCompositionRelationship* (p.132)**:**

A relationship in SMOOA can be a composition relationship. Thus, the relationships in *IM* can be divided into two sets: base relationships and composition relationships. We modify schema *IMCompositionRelationship* to be the following:

auxIMCompositionRelationship

$$
\begin{array}{l}
IMDTData \\
baseRelationship : \mathbb{P}\, RELATIONSHIP \\
comRelationship : \mathbb{P}\, RELATIONSHIP \\
composedBy : RELATIONSHIP \leftrightarrow \mathbb{P}\, RELATIONSHIP \times FUNCTION \\
\hline
\langle baseRelationship, comRelationship \rangle \ \mathsf{partition}\ relationship \\
\mathrm{dom}\, composedBy = comRelationship \\
\forall\, r : comRelationship \bullet first\,(composedBy(r)) \subset relationship
\end{array}
$$

As to *OMAssociationType*, we need to reformulate $OM.assocDeriving$, which is a "mistyped" non-comparable variable:

$$ASSOCIATION \nrightarrow DerivedElement[ASSOCIATION]$$

It can also be expressed as

$$ASSOCIATION \nrightarrow ASSOCIATION \times \mathbb{P}\, ASSOCIATION \times FUNCTION$$

The domain and the first item of the range of $OM.assocDeriving$ are identical, because $OM.p_3$ states that

$$\forall\, da : derivedAssoc \bullet (assocDeriving(da)).derived = da$$

Therefore, we change $OM.assocDeriving$ to $OM.auxassocDeriving$ without altering its semantics:

$$auxassocDeriving : ASSOCIATION \leftrightarrow \mathbb{P}\, ASSOCIATION \times FUNCTION$$

and change $OM.p_3$ to the following:

$$\forall\, da : derivedAssoc \bullet first\,(auxassocDeriving(da)) \subset association$$

Variable:

$$f_v \uparrow (OMAssociationType, auxIMCompositionRelationship) = \{(baseAssoc, baseRelationship), (derivedAssoc, comRelationship), (auxassocDeriving, composedBy)\}$$

Since $assocDeriving$ is equivalent to $auxassocDeriving$, $assocDeriving$ belongs to $Core_{OM}$.

Predicate:

We can prove that

$$OM.p_1 \simeq_{sem} IM.p_1$$
$$OM.p_2 \simeq_{sem} IM.p_2$$
$$OM.p_3 \simeq_{sem} IM.p_3$$

The predicates defined in the two schemas belong to $Core_{OM}$ and $Core_{IM}$.

OMClassType2 (p.100) :

Schema *OMClassType2* is about derived classes. *IM* does not support the meaning of "derived object", so all the objects in *IM* are base objects. We add a schema to *IM*:

auxIMObjectType

$$IMDTData$$
$$baseObject : \mathbb{P}\, OBJECT$$

$$baseObject = object$$

Variable:

$$f_v \uparrow (OMClassType2, auxIMObjectType) = \{(baseClass, baseObject)\}$$
$$OMClassType2.Sig.Var \setminus \mathrm{dom}\, f_v = \{derivedClass, clsDeriving\}$$
$$auxIMObjectType.Sig.Var \setminus \mathrm{ran}\, f_v = \emptyset$$

Predicate:

$\underline{OM.p_1 \text{ vs. } IM.p_1}$:

$OM.p_1$ and $IM.p_1$ describe the values that $OM.class$ and $IM.object$ can take. When $OM.derivedClass = \emptyset$,

$$OM.p_1 \simeq_{sem} IM.p_1$$

According to rule 10

$$OM.p_1 \in Ext_{OM}.ccPred_{IM} \quad \text{and} \quad IM.p_1 \in Ext_{IM}.ccPred_{IM}$$

$\underline{OM.p_2 \text{ and } OM.p_3}$:

$OM.p_2$ and $OM.p_3$ are specific to OM. They belong to $Ext_{OM}.ecPred$.

OMAttributeType (p.102) :

IM does not support the meaning of "derived attribute"; therefore, all the attributes in IM are base attributes. The discussion about $OMAttributeType$ is similar to that about $OMClassType2$.

We add a schema to IM:

auxIMAttributeType

$$IMDTData$$
$$baseAttr : \mathbb{P}\, ATTRIBUTE$$
$$baseAttr = attribute$$

Variable:

$f_v \uparrow (OMAttributeType, auxIMAttributeType) = \{(baseAttr, baseAttr)\}$
$OMAttributeType.Sig.Var \setminus \mathrm{dom}\, f_v = \{derivedAttr, attrDeriving\}$
$auxIMAttributeType.Sig.Var \setminus \mathrm{ran}\, f_v = \emptyset$

Predicate:

$OM.p_1 \in Ext_{OM}.ccPred_{IM}$ and $IM.p_1 \in Ext_{IM}.ccPred_{IM}$
$\{OM.p_2, OM.p_3\} \subset Ext_{OM}.ecPred$

7.2.10 Generalizations

Both *OM* and *IM* support the concept "generalization". *OM* also supports "aggregation", which has a similar construct to generalization; *IM*, however, has no such concept. While combining the commonality of the two concepts together in theory *OM*, we now need to separate them and focus on each individual.

There are four schemas about both generalization and aggregation in *OM*. They are *OMGroupingBasic* (p.71), *OMGroupedClass* (p.72), *OMGroupedAssociation* (p.73), and *OMGrouping* (p.74). According to the following two predicates in *OM*,

$\langle aggregation, generalization \rangle$ partition $grouping$
$\langle aggAssociation, genAssociation \rangle$ partition $groupedAssociation$

we can assign the empty set $\emptyset$ to $OM.aggregation$ and to $OM.aggAssociation$ while comparing generalization constructs of the two theories. By doing so, we obtain four schemas for generalization. They are *genOMGroupingBasic*, *genOMGroupedClass*, *genOMGroupedAssociation*, and *genOMGrouping*.

genOMGroupingBasic

$OMDTData$
$generalization : \mathbb{P}\, GENERALIZATION$

genOMGroupedClass

$$genOMGroupingBasic$$
$$gen\_single : GENERALIZATION \pfun CLASS$$
$$gen\_group : GENERALIZATION \rel CLASS$$

$$gen\_single \in generalization \fun class$$
$$\dom gen\_group = generalization \land \ran gen\_group \subset class$$

genOMGroupedAssociation

$$OMAssociation$$
$$genAssociation : \power ASSOCIATION$$
$$gen\_parent : ASSOCIATION \pinj ROLE$$
$$gen\_child : ASSOCIATION \pinj ROLE$$

$$genAssociation \subseteq binaryAssociation$$
$$gen\_parent \in genAssociation \inj singularRole$$
$$gen\_child \in groupedAssociation \inj optionalRole$$
$$\forall a : genAssociation \bullet hasRole(\!| \{a\} |\!) = \{gen\_parent(a), gen\_child(a)\}$$

It should be noted that the third predicate of schema *genOMGroupedAssociation* is obtained from the following two predicates, which are defined in *OMGroupedAssociation* and *OMGeneralizationBasic* (p.78) respectively:

$$child \in groupedAssociation \inj role$$
$$\forall g : generalization \bullet (collectedIn^{\sim} \fatsemi child)(\!| \{g\} |\!) \subseteq optionalRole$$

genOMGrouping

$genOMGroupedClass$

$genOMGroupedAssociation$

$gen\_collectedIn : ASSOCIATION \pfun GENERALIZATION$

$gen\_collectedIn \in genAssociation \fpfun generalization$

$\forall\, a : genAssociation;\ g : generalization \mid gen\_collectedIn(a) = g \bullet$
$\quad playerOfRole(gen\_parent(a)) = gen\_single(g) \land$
$\quad playerOfRole(gen\_child(a)) \in gen\_group(\!|\ \{g\}\ |\!)$

$\forall\, g : generalization \bullet$
$\quad \#(gen\_group(\!|\ \{g\}\ |\!)) = \#(gen\_collectedIn^{\sim}(\!|\ \{g\}\ |\!))$

We now compare the four schemas with *IMGeneralizationBasic* (p.134), *IMGeneralizationRelationship* (p.134), and *IMGeneralization* (p.136).

Variable:

$$f_v \uparrow (genOMGroupingBasic \cup genOMGroupedClass \cup genOMGroupedAssociation \cup genOMGrouping, IMGeneralizationBasic \cup IMGeneralizationRelationship \cup IMGeneralization) = \{(generalization, generalization), (gen\_single, supertype), (gen\_group, subtype), (genAssociation, genRelationship), (gen\_parent, superRole), (gen\_child, subRole), (gen\_collectedIn, collectedIn)\}$$

Predicate:

We can prove that the following pairs of predicates are equivalent.

$$genOMGroupedClass.p_1 \simeq_{sem} IMGeneralizationBasic.p_1$$
$$genOMGroupedClass.p_2 \simeq_{sem} IMGeneralizationBasic.p_2$$
$$genOMGroupedAssociation.p_1 \simeq_{sem} IMGeneralizationRelationship.p_1$$
$$genOMGroupedAssociation.p_2 \simeq_{sem} IMGeneralizationRelationship.p_2$$
$$genOMGroupedAssociation.p_3 \simeq_{sem} IMGeneralizationRelationship.p_3$$
$$genOMGroupedAssociation.p_4 \simeq_{sem} IMGeneralizationRelationship.p_4$$
$$genOMGrouping.p_1 \simeq_{sem} IMGeneralization.p_1$$
$$genOMGrouping.p_2 \simeq_{sem} IMGeneralization.p_2$$
$$genOMGrouping.p_3 \simeq_{sem} IMGeneralization.p_3$$

All of the above predicates belong to the core.

OMIsKindOf (p.78) **vs.** *IMInheritanceBasic* (p.137) **:**
Variable:

$$f_v \uparrow (OMIsKindOf, IMInheritanceBasic) = \{(isKindOf, isKindOf), (isDescendentOf, inherit)\}$$

Variable $OM.isAncestorOf$ is redundant because

$$OM.isAncestorOf = OM.isDescendentOf^{\sim}$$

It belongs to $Core_{OM}$.
Predicate:

In *IM*, although not explicitly stated, generalization is transitive and asymmetric. In other words, in an object model, if $o1$ is a subtype object of $o2$ and $o2$ is a subtype object of $o3$, then $o1$ is a subtype object of $o3$; and if $o1$ is a subtype object of $o2$, then $o2$ cannot be a subtype object of $o1$. We add two predicates to specify these properties:

$$\forall\, o1, o2, o3 : object \mid o1 \mapsto o2 \in inherit \wedge o2 \mapsto o3 \in inherit \bullet o1 \mapsto o3 \in inherit \qquad (IMIsKindOf.p_{add_1})$$

$$\forall\, o1, o2 : object \mid o1 \mapsto o2 \in inherit \bullet o2 \mapsto o1 \notin inherit \qquad (IMIsKindOf.p_{add_2})$$

We can prove that

$$OM.p_1 \simeq_{sem} IM.p_1$$
$$OM.p_2 \simeq_{sem} IM.p_2$$
$$OM.p_3 \simeq_{sem} IM.p_3$$

Therefore, these predicates belong to the core.

$\underline{OM.p_4}$ vs. $\underline{IM.p_{add_1}}$ and $\underline{IM.p_{add_2}}$:

$OM.p_4$ can be expanded as follows.

$$\forall\, a, b, c : CLASS \mid a \mapsto b \in isDescendentOf \wedge b \mapsto c \in isDescendentOf \bullet a \mapsto c \in isDescendentOf$$
$$\forall\, a, b : CLASS \mid a \mapsto b \in isDescendentOf \bullet b \mapsto a \notin isDescendentOf$$

Every element in $CLASS$ satisfies the above two predicates, so does every element in *class*. We have

$$OM.p_4 \simeq_{sem} \{IM.p_{add_1}, IM.p_{add_2}\}$$

Therefore, the predicates belong to the core.

$\underline{OM.p_5}$:

According to rule 14, it belongs to $Core_{OM}$.

$\underline{IM.p_4}$: It will be discussed when we discuss disjoint and overlapping generalization (on p.227).

OMInheritance (p.79) **:**

The predicate in this schema is specific to OM, so it belongs to $Ext_{OM}.ccPred_{OM}$. It specifies the meaning of variable $OM.hasFeature$.

IMAttributeInheritance (p.139) **:**

This schema is about properties related to variables $IM.hasIdElement$ and $IM.domainSpec$. The properties are not possessed by any variable in OM. Therefore, the predicates defined in the schema belong to $Ext_{IM}.ccPred_{IM}$.

OMSubclassAttributeRange (p.92) **:**

The predicate in this schema imposes the constraint on value ranges of an inherited attribute in a subclass. It is related to variable $OM.valueRange$. Since IM does not specify the similar properties on inherited attributes of an subtype object, this predicate is specific to OM. Therefore, it belongs to $Ext_{OM}.ccPred_{OM}$.

Overriding

There are three schemas about overriding: *OMOverride* (p.80), *OMDefaultValueOverride* (p.81), and *OMMethodOverride* (p.82). They are about the properties pertinent to default attribute values and methods. The variables, $OM.defaultValOverride$ and $OM.methodOverride$, belong to Ext_{OM}; the predicates defined in the schemas belong to $Ext_{OM}.ecPred$.

Multiple Inheritance

Both OM and IM support the meaning of multiple inheritance: a class or an object can have more than one superclass or supertype object. Such a class in OM is called "join class", which is formalized in schema *OMMultipleInheritance* (p.85). IM does not give a name to such an object. We here give an auxiliary schema for IM to define a set of "join objects", which are objects with multiple inheritance.

auxIMMultipleInheritance

$IMInheritance$

$joinObject : \mathbb{P}\ OBJECT$

$joinObject \subset \mathrm{dom}\ isKindOf$

$\forall jo : joinObject \bullet \#(isKindOf(\!|\ \{jo\}\ |\!)) > 1$

Variable:

$$f_v \uparrow (OMMultipleInheritance, auxIMMultipleInheritance) = \{(joinClass, joinObject)\}$$

Predicate:

It is obvious that

$$OM.p_1 \simeq_{sem} IM.p_1$$
$$OM.p_2 \simeq_{sem} IM.p_2$$

Therefore, the predicates belong to the core.

Disjoint and Overlapping Generalizations

OM allows overlapping generalizations, as defined in *OMGeneralizationType* (p.86); however, *IM* does not. The following predicate in *IMInheritanceBasic* (p.137)

$$\forall gen : generalization;\ sub1, sub2 : object \mid \{sub1, sub2\} \subseteq subtype(\!|\ \{gen\}\ |\!) \bullet \mathrm{dom}(inherit \rhd \{sub1\}) \cap \mathrm{dom}(inherit \rhd \{sub2\}) = \emptyset$$

specifies this property of *IM*.

We give the following auxiliary schema, in which the auxiliary variable *IM.disjointGen* is defined.

auxIMGeneralizationType

$IMInheritance$

$auxIMMultipleInheritance$

$disjointGen : \mathbb{P}\ GENERALIZATION$

$disjointGen = generalization$

Variable:

$$f_v \uparrow (OMGeneralizationType, auxIMGeneralizationType) = \{(disjointGen, disjointGen)\}$$
$$OMGeneralizationType.Sig.Var \setminus \mathrm{dom}\, f_v = \{overlappingGen\}$$

Predicate:

$\underline{OM.p_1}$ vs. $\underline{auxIMGeneralizationType.p_1}$:

When $OM.overlappingGen = \emptyset$,

$$OM.p_1 \simeq_{sem} auxIMGeneralizationType.p_1$$

According to rule 10,

$$OM.p_1 \in Ext_{OM}.ccPred_{IM} \quad \text{and} \quad auxIMGeneralizationType.p_1 \in Ext_{IM}.ccPred_{IM}$$

$\underline{OM.p_2}$ vs. $\underline{IMInheritanceBasic.p_4}$:

Since $IM.disjointGen = IM.generalization$, we have

$$IMInheritanceBasic.p_4 \Leftrightarrow$$
$$\forall\, gen : disjointGen;\ sub1, sub2 : object \mid$$
$$\{sub1, sub2\} \subseteq subtype(\!|\ \{gen\}\ |\!) \bullet$$
$$\mathrm{dom}(inherit \rhd \{sub1\}) \cap \mathrm{dom}(inherit \rhd \{sub2\}) = \emptyset$$

It can be proven that

$$OMGeneralizationType.p_2 \simeq_{sem} IMInheritanceBasic.p_4$$

This also shows that a generalization in IM is comparable to a disjoint generalization in OM. The two predicates belong to the core.

$\underline{OM.p_3}$:

Predicate $OM.p_3$ is about overlapping generalization. It is specific to OM, so it belongs to $Ext_{OM}.ecPred$.

OMGenExtension (p.87) :

The variable $OM.extension$ defined in this schema is an "underlying" non-comparable variable, because new features can also be added to a subtype object in IM. Therefore, the variable and the predicates defined in the schema belong to $Core_{OM}$.

Abstract Classes and Abstract Operations

*OMClassType*1 (p.93) :

Classes in a generalization in OM can be abstract; however, objects in a generalization in IM cannot. We define two auxiliary variables in IM:

auxIMObjectType

$$concreteObject : \mathbb{P}\, OBJECT$$
$$leafObject : \mathbb{P}\, OBJECT$$

$$concreteObject \subseteq object$$
$$leafObject = (\operatorname{dom} inherit \setminus \operatorname{ran} inherit) \subseteq concreteObject$$

These two variables have similar meaning to $OM.concreteClass$ and $OM.leafClass$ in $OMClassType1$.

Variable:

$$f_v \uparrow (OMClassType1, auxIMClassType) = \{(concreteClass, concreteObject), (leafClass, leafObject)\}$$
$$OMClassType1.Sig.Var \setminus \operatorname{dom} f_v = \{abstractClass\}$$

Predicate:

We can prove that

$$OM.p_1 \simeq_{sem} IM.p_1$$
$$OM.p_3 \simeq_{sem} IM.p_2$$

Therefore, these two predicates belong to the core.

$OM.abstractClass$ is specific to OM, so is $OM.p_2$ which defines the semantics of $OM.abstractClass$. Therefore, $OM.p_2$ belongs to $Ext_{OM}.ecPred$.

OMOperationType2 (p.93) **:**

Operations are specific to OM, so are abstract and concrete operations. The variables defined in *OMOperationType2* belong to Ext_{OM}.

$$\{OM.abstractOp, OM.concreteOp\} \subset Ext_{OM}$$

The predicates defined in the schema belong to $Ext_{OM}.ecPred$.

7.2.11 Aggregations

Although *IM* does not support the concept "aggregation", it still can model the "aggregation" construct in the real-world in the form of binary relationships.

We can create a set of concepts and predicates in *IM* to correspond the concepts and predicates related to aggregation in OM, and then do the similar comparison as in Section 7.2.10. The result is

$$\{OM.aggregation, OM.grouping, OM.single,$$
$$OM.group, OM.groupedAssociation, OM.aggAssociation,$$
$$OM.parent, OM.child, OM.collectedIn\} \subset Core_{OM}$$

The predicates defined in schemas *OMGroupingBasic*, *OMGroupedClass*, *OMGroupedAssociation*, and *OMGrouping* belong to $Core_{OM}$.

We can assume that the objects involved in an aggregation have "is-part-of" relationships. Therefore, the variable $OM.isPartOf$ and the predicates defined in schema *OMIsPartOf* (p.75) belong to $Core_{OM}$.

Aggregation relationships in IM can also be fixed, variable, and recursive. Therefore, the variables and predicates defined in schema *OMAggregationType* (p.84) belong to $Core_{OM}$.

The above variables related to aggregation are "explicitly defined" non-comparable variables. However, two schemas about aggregation in OM, namely *OMDependence* (p.76) and *OMPropagation* (p.77), are specific to OM. The variables defined in these schemas belong to Ext_{OM}:

$$\{OM.existenceDependOn, OM.featurePropagate\} \subset Ext_{OM}$$

The predicates defined in these schemas belong to $Ext_{OM}.ecPred$.

7.2.12 Metadata

Schemas *OMPattern* (p.94) and *OMDescriptor* (p.95) are specific to OM. The variables defined in the schemas belong to Ext_{OM}.

$$\{OM.pattern, OM.instantiation, classDescriptor, classFeature\} \subset Ext_{OM}$$

The predicates defined in the schemas belong to $Ext_{OM}.ecPred$.

7.2.13 Modules

The variables $OM.module$ and $OM.containedIn$ defined in schema *OMModule* (p.83) are "explicitly defined" non-comparable variables, because modules in OMT are simply a mechanism to group a collection of classes. We can add auxiliary variables to IM to stand for the similar concepts. Therefore, the two variables and the predicate defined in the schema belong to $Core_{OM}$.

7.2.14 Relationship Formalization

In the information model, the relationships between objects are "formalized" by means of referential attributes. This mechanism adds details that strongly suggest the implementation of relationships as relational tables.

The object model in OM, on the other hand, does not have referential attributes, but still can model the relationships between classes.

If taking out the referential attributes from the formalizers (except associative and subtype objects, because the referential attributes in these formalizers also act as identifiers and should not be taken out), we still can map an object in IM to a class in OM if they have the same set of naming and descriptive attributes, and map a relationship to an association if they involve the same set of objects or classes.

Let fm be the set of formalizers whose referential attributes can be taken out:

$$fm = formalizer \setminus (associativeObject \cup \mathrm{dom}\ isKindOf)$$

After the referential attributes being taken out from every element in fm, variable $IM.attrDefinedIn$ becomes $IM.auxattrDefinedIn$, which is defined as follows.

$$IM.auxattrDefinedIn : ATTRIBUTE \leftrightarrow OBJECT$$

It satisfies

$$\forall\, o : fm \bullet IM.auxattrDefinedIn^{\sim}(\!|\, \{o\} \,|\!) = IM.attrDefinedIn^{\sim}(\!|\, \{o\} \,|\!) \setminus referentialAttr$$
$$\forall\, o : object \setminus fm \bullet IM.auxattrDefinedIn^{\sim}(\!|\, \{o\} \,|\!) = IM.attrDefinedIn^{\sim}(\!|\, \{o\} \,|\!)$$

We then redefine $f_v(OM.attrDefinedIn) = IM.attrDefinedIn$ to $f_v(OM.attrDefinedIn) = IM.auxattrDefinedIn$.

Since a formalizer in IM may still have a corresponding class in OM, the variable $IM.formalizer$ should be in $Core_{IM}$. The first predicate in *IMRelationshipFormalizingBasic* (p.140), which is about $IM.formalizer$, also belongs to $Core_{IM}$.

With the above modification, variable $IM.referencing$ is no longer needed for fm, neither are the predicates in schemas *IMReferencing* (p.141), and *IMPlayerFormalizing* (p.142). We put them into Ext_{IM}.

Association classes vs. Associative Objects:

The association classes are formalized in schema *OMAssociationClass* (p.65); the associative objects are formalized in schemas *IMAssociativeObjectFormalizing* (p.143) and *IMAssociativeObjectFormalizingType* (p.144). They have similar meaning and therefore are defined to be comparable. That is,

$$f_v(OM.associationClass) = IM.associativeObject$$

Relation variable $OM.modeledAs$ specifies the relationship between association classes and associations in OM; $IM.formalizedBy$ specifies the relationship between associative objects and relationships in IM. They are comparable. That is,

$$f_v(OM.modeledAs) = IM.formalizedBy$$

We also obtain that,

$$OMAssociationClass.p_1 \in Core_{OM}$$
$$IMAssociativeObjectFormalizing.p_1 \in Core_{IM}$$
$$\{OMAssociationClass.p_2, OMAssociationClass.p_3\} \subset Ext_{OM}.ccPred_{OM}$$
$$IMRelationshipFormalizingBasic.p_2 \in Ext_{IM}.ccPred_{OM}$$

The fourth predicate defined in *OMAssociationClass* imposes constraints on
$OM.attrDefinedIn$, so it belongs to $Ext_{OM}.ccPred_{OM}$. The second and third predicates defined in *IMAssociativeObjectFormalizing* are specific to IM, so they belong to $Ext_{IM}.ccPred_{IM}$.

$IM.singleOccurrence$ is an "explicitly defined" non-comparable variable, so it belongs to $Core_{IM}$; $IM.multipleOccurrence$ is specific to IM, so it belongs to

Ext_{IM}. The predicate defined in *IMAssociativeObjectFormalizingType* belongs to $Ext_{IM}.ccPred_{OM}$.

IMGenRelationshipFormalizing (p.145):

The predicates defined in this schema are about generalization relationships. They belong to $Ext_{IM}.ccPred_{IM}$.

IMReferentialAttributeDomain (p.145):

The predicates defined in this schema are about domain restriction on referential attributes of the formalizers other than the subtype objects. They belong to $Ext_{IM}.ecPred_{IM}$.

7.3 Summary

This section summarizes the result of the comparison presented in Section 7.2. Table 7.2 shows the variables representing concepts from the core. The first and the third column present the comparable variables, where auxiliary variables are labeled "(aux)". The second (or the fourth) column presents whether or not the predicates in Ext_{OM} (or Ext_{IM}) directly impose additional constraints on the variables in OM (or IM). Table 7.3 and Table 7.4 show the variables representing the concepts from Ext_{OM} and Ext_{IM} respectively.

The tables illustrate that the two theories have a big core and two small extensions. It is unsurprising because the modeling techniques of the two methods are very similar. The extension of OMT to the core arises mainly because of operations and methods defined in classes, and because of a variety of classes, such as derived classes, abstract classes, patterns, and class descriptors. As to attributes, OMT has definitions for default attribute value and derived attribute, which SMOOA does not have. The generalization construct of OMT differs from that of SMOOA mainly in that the former allows overlapping generalizations, abstract classes, and feature overriding. The aggregation construct of OMT has richer semantics in that it defines semantics of feature propagation along an aggregation network, and defines semantics of

existence dependency among the classes within an aggregation. The extension of SMOOA to the core arises mainly because it uses referential attributes to formalize relationships.

In general, SMOOA imposes more restrictions on the variables in the core than OMT does. The roles, the associations and the relationships in the two theories have the same restrictions. However, SMOOA is more strict with the category and relation variables related to attributes, features, identifiers, and generalizations. We can obtain more detailed differences between the comparable variables by classifying the core into four characterizing sets, which are demonstrated in Appendix B.

Since we formalize the similar design components in a similar way, the comparison of these components is fairly straightforward. A non-comparable variable of *OM* in the core means that OMT has an extra notation for certain design category; however, without such a notation, SMOOA still can model the same elements of that design category. It is the same with a non-comparable variable of *IM*. *OM* has more non-comparable variables listed in the core than *IM* has. This means that SMOOA uses a smaller set of concepts and notations than OMT, but still has the same expressive power in certain aspects. Most of the non-comparable variables in the core are "explicitly defined" non-comparable variables. There is only one pair of "mistyped" non-comparable variables, and one "technically different" non-comparable variable.

The comparison reveals some ambiguities residing in the methods. One example is the association classes and the associative objects. An early version of our comparison results showed that the two concepts were distinct. OMT states that "the attributes of an association are properties of the association, and cannot be attached to any class that plays a role in the association" (see [RBP+91]: p.32). We formalized this as a design rule:

OMAssociationClassAttribute

OMAssociationClass

$$\forall\, a : \mathrm{dom}\, modeledAs \bullet$$
$$hasFeature(\!| playerOfRole(\!| hasRole(\!| \{a\} |\!) |\!) |\!) \cap$$
$$hasFeature(\!| \{modeledAs(a)\} |\!) \cap attribute = \emptyset$$

From $IMAssociativeObjectFormalizing.p_3$, we know that the identifier of an associative object contains the combination of the identifiers of all the objects involved in the relationship. These two predicates contradict each other. Therefore, no instance of association classes can be mapped to an instance of associative objects, and vice versa. We concluded that the two variables were not comparable. This result came from our interpretation of the above ambiguous statement in OMT as a design rule. As this unexpected result is different from the intuitive assumption that the two concepts are comparable, we went back to the original material for a careful study, and found that the statement should be a design guideline, not a design rule.

Table 7.1: Mapping between the basic types and subsets of the basic types of *OM* and *IM*

type in OMT	type in SMOOA	type in OMT	type in SMOOA
ENTITY	*ENTITY*	*GROUPING*	n/a
CLASS	*OBJECT*	*GENERALIZATION*	*GENERALIZATION*
ASSOCIATION	*RELATIONSHIP*	*AGGREGATION*	n/a
FEATURE	*FEATURE*	*MODULE*	n/a
ATTRIBUTE	*ATTRIBUTE*	*RULE*	n/a
OPERATION	n/a	*KEY*	*IDENTIFIER*
ROLE	*ROLE*	*FUNCTION*	n/a
METHOD	n/a	n/a	*TEXT*
ATT_TYPE	n/a	*INSTANCE*	*INSTANCE*, *RW_INSTANCE*
ARG_TYPE	n/a	*OBJECT*	*RW_INSTANCE*
NAME	*NAME*	n/a	*OBJ_INSTANCE*
n/a	*KEY_LETTER*	*LINK*	*REL_INSTANCE*
n/a	*LABEL*	*ATOMIC_VALUE*	*ATOMIC_VALUE*

Note: The order in which the comparable types appear in the table is in accordance with the order in which they are introduced in the two theories.

Table 7.2: The variables in $Core_{OM}$ and $Core_{IM}$

variable in OM	constraint	variable in IM	constraint
entity		*entity*	
feature	$ccPred_{IM}$	*feature*	$ccPred_{IM}$
$definedIn^{(2)}$	$ccPred_{OM}$ $ccPred_{IM}$	*definedIn* (aux)	$ccPred_{IM}$
hasFeature	$ccPred_{OM}$	*hasFeature*	$ccPred_{IM}$
class	$ccPred_{IM}$	*object*	$ccPred_{IM}$
$baseClass^{(2)}$		*baseObject* (aux)	
$joinClass^{(2)}$		*joinObject* (aux)	
$concreteClass^{(2)}$		*concreteObject* (aux)	
$leafClass^{(2)}$		*leafObject* (aux)	
		$formalizer^{(6)}$	
associationClass		*associativeObject*	
attribute	$ccPred_{IM}$	*attribute*	$ccPred_{IM}$
attrDefinedIn	$ccPred_{OM}$	*attrDefinedIn*	$ccPred_{IM}$
		$hasAttribute^{(1)}$	
descriptiveAttr (aux)		$descriptiveAttr^{(2)}$	
namingAttr (aux)		$namingAttr^{(2)}$	
$baseAttr^{(2)}$		*baseAttr* (aux)	

(1): "redundant" non-comparable variable

(2): "explicitly defined" non-comparable variable

(3): "mistyped" non-comparable variable

(4): "underlying" non-comparable variable

(5): "subsumed" non-comparable variable

(6): "technically different" non-comparable variable

Table 7.2: The variables in $Core_{OM}$ and $Core_{IM}$ (continued)

variable in OM	constraint	variable in IM	constraint
valueRange	$ccPred_{OM}$	*domainSpec*	$ccPred_{IM}$
$valueOfType^{(4)}$			
$typeOfAttr^{(4)}$			
$restrict^{(4)}$			
$generalRestrict^{(4)}$			
$constraint^{(4)}$			
key		*identifier*	
hasKeyElement	$ccPred_{OM}$	*hasIdElement*	$ccPred_{IM}$
$keyedEntity^{(2)}$		*idedEntity* (aux)	$ccPred_{IM}$
isKeyOf	$ccPred_{OM}$	*isIdOf*	
$theKey^{(1)}$	$ccPred_{OM}$		
hasPreferredKey (aux)		$hasPreferredId^{(2)}$	
role		*role*	
playerOfRole		*playerOfRole*	
multiplicityOfRole		*multiplicityOfRole*	
roleDefinedIn		*roleDefinedIn*	
hasRole		*hasRole*	
singularRole		*single*	
optionalRole		*conditionSingle*	
onemulRole (aux)		$multiple^{(2)}$	
optmulRole (aux)		$conditionMultiple^{(2)}$	
$multipleRole^{(2)}$			
$orderedRole^{(2)}$			
association		*relationship*	
binaryAssociation		*binaryRelationship*	
one_oneAssoc		R_{11}	
one_onemulAssoc (aux)		$R_{1M}{}^{(2)}$	
one_optmulAssoc (aux)		$R_{1Mc}{}^{(2)}$	
$one\_mulAssoc^{(2)}$			
onemul_onemulAssoc (aux)		$R_{MM}{}^{(2)}$	
onemul_optmulAssoc (aux)		$R_{MMc}{}^{(2)}$	
optmul_optmulAssoc (aux)		$R_{McMc}{}^{(2)}$	
$mul\_mulAssoc^{(2)}$			

Table 7.2: The variables in $Core_{OM}$ and $Core_{IM}$ (continued)

variable in OM	constraint	variable in IM	constraint
$qualifiedAssociation^{(2)}$			
$qualify^{(2)}$			
$baseAssoc^{(2)}$		*baseRelationship* (aux)	
derivedAssoc		*comRelationship*	
$assocDeriving^{(3)}$		$composedBy^{(3)}$	
$groupedAssociation^{(2)}$			
genAssociation		*genRelationship*	
$aggAssociation^{(2)}$			
$grouping^{(2)}$			
generalization	$ccPred_{IM}$	*generalization*	$ccPred_{IM}$
$aggregation^{(2)}$			
$single^{(2)}$			
gen_single (aux)		*supertype*	
$group^{(2)}$			
gen_group (aux)		*subtype*	
$parent^{(2)}$			
gen_parent (aux)		*superRole*	
$child^{(2)}$			
gen_child (aux)		*subRole*	
$collectedIn^{(2)}$			
gen_collectedIn (aux)		*collectedIn*	
isKindOf		*isKindOf*	
isDescendentOf		*inherit*	
$isAncestorOf^{(1)}$			
$disjointGen^{(2)}$		*disjointGen* (aux)	
$extension^{(2)}$			
$isPartOf^{(2)}$			
$fixedAgg^{(2)}$			
$variableAgg^{(2)}$			
$recursiveAgg^{(2)}$			
$module^{(2)}$			
$containedIn^{(2)}$			
modeledAs	$ccPred_{OM}$	*formalizedBy*	$ccPred_{OM}$

Table 7.3: The variables in extension Ext_{OM}

operation	*implement*	*abstractOp*
argumentOfOp	*methodLookup*	*concreteOp*
resultOfOp	*derivedClass*	*existenceDependOn*
opDefinedIn	*clsDeriving*	*featurePropagate*
queryOp	*derivedAttr*	*pattern*
updateOp	*attrDeriving*	*instantiation*
defaultValOfAttr	*defaultValOverride*	*classDescriptor*
method	*methodOverride*	*classFeature*
argumentOfMethod	*overlappingGen*	
resultOfMethod	*abstractClass*	

Table 7.4: The variables in extension Ext_{IM}

referentialAttr	*referencing*	*multipleOccurrence*

Chapter 8

Conclusion

The statement of the thesis is "Following a rigorous approach, the comparison of the representational properties of the OOAD methods can be precise, detailed and objective." In this chapter, we summarize the work that supports the thesis claim and meets the desirable properties of a solution; we also suggest directions for future research and improvements.

8.1 Summary

This dissertation has two major contributions:

- *An approach to formalizing the representational properties of OOAD methods.*

  We have described the principles and guidelines for establishing a formal design theory for an OOAD method under the Theory-Model paradigm. Z variables are used to define the design-time and run-time concepts; and Z predicates are used to describe the rules these concepts must satisfy. The variables and predicates specify the formal semantics of the method. The formalization process is iterative. The verification of a design theory can be done via type-checking, inspection, reasoning, and execution.

We have applied the formalization approach to the object model of OMT and the information model of Shlaer-Mellor OOA. The result has shown that the formalization improves the accuracy of the methods and also improves one's understanding of the methods.

- *An approach to systematically comparing the OOAD methods.*

  We have given formal definitions, in terms of equivalent mapping between models, for the core, the extensions to the core, and the characterizing sets in the core. We have provided rules for classifying the portions of the design theories into the core, the extensions, and the characterizing sets, which reveal similarities and differences among the methods. The comparison activities follow an incremental process.

  We have compared the object model of OMT and the information model of Shlaer-Mellor OOA, and obtained a detailed comparison result. It has shown that given the formal description of the methods, one can precisely compare them based on the comparison mechanism. Formal comparison leads to a better understanding of the similarities and differences among the methods.

The formalization and comparison results also contribute to the research areas such as method evaluation, method integration, Meta-CASE tools design, and design reuse. These are briefly discussed in Section 8.2.

The comparison approach satisfies the desirable properties listed in Section 1.3:

- It is *general* because it accommodates the existing OOAD methods. In fact, the approach can be applied to any class of methods with similar modeling techniques.

- It is *precise* because it is based on the formal theories of the methods, and therefore formal reasoning about the equivalence of the predicates is possible.

- It is *objective* and *detailed* because it forces one to compare the entire theory of a method rather than choose specific topics.
- It provides a *systematic* comparison in that it provides guidelines and rules for one to formalize the methods and compare the corresponding theories step by step.

8.2 Future Work

Based on the results already achieved, the following work may be continued.

- Establish an ER model for each theory.

  As is discussed in Section 2.2, an ER model serves as a data model for managing design information. An ER diagram is also a means of graphical representation of the design theory.

  We use a variant of the entity-relationship approach [Che76] for this. Translation of our style of Z specification to an ER model is fairly straightforward. Category variables become entity sets, represented by boxes; predefined types and some given sets, such as $\mathbb{Z}$ and $ATOMIC\_VALUE$, become value sets, represented by ovals. Relation variables become relationship sets, usually represented by arrows, which point in the direction implied by the names of the Z functions or relations. Figure 3.2(a) illustrates an ER diagram for object model theory.

- Develop an executable representation of each theory to check its soundness.

  The translation method so far developed is discussed in Section 3.4. There is research on automatic translation from Z or other formal specifications to Prolog [Dic89, Hab91]. A topic for future research is to explore a suitable way of automatically translating the design theories and their models from Z specification to Prolog.

- Develop a detailed systematic mechanism for verifying the correctness, consistency and completeness of the theories, by means of formal techniques.

  Graham [Gra96] developed a method for formally testing the correctness of program visualization tools with respect to some theory describing the domain of the tool. The method is based on encoding the theory in logic, and using the PVS theorem prover to investigate the properties of the programs generated by the visualization tool. He discussed the possibility of combining our approach of verifying design with their approach. Future research along this direction can start from this interesting point.

- Formalize dynamic and functional view as well as other aspects relevant to the representational properties of OMT and Shlaer-Mellor OOA, and thoroughly compare the two methods.

  The OOAD methods are still being improved. Some components of a design method may not be well defined, theories for these components thereby may not be precise; some parts of a design theory may need change as the design method evolves, and so do the comparison results.

- Formalize other OOAD methods and compare them.

  We will consider Jacobson *et al.*'s Objectory [JCJO92] as the next method to be compared against OMT and SMOOA. This method is widely used, and its scenario-driven (use case) approach is different from OMT and SMOOA.

  The approach discussed in this dissertation has been applied to two methods. We anticipate that when more methods are taken into consideration, there will be cases which the existing decision rules cannot cover. Then new rules need to be added or some of the existing rules need to be improved to suit those circumstances.

The decision rules guide a fairly straightforward comparison when two comparable components are similar. As for two comparable components with disparity, however, the rules do not give an efficient solution: human creativity is needed to modify the design theories. This is where the approach needs to improve.

- Formalize other aspects of a design method, such as design process and metrics for better designs.

The following research can be built on the formalization and comparison results:

- *Method evaluation.* By examining the core and extensions of the design theories, one can evaluate the design methods under certain criteria, and therefore can obtain knowledge about the methods' comparative strengths and weaknesses. For example, one may want to examine the effect of absence of some notations in a method. SMOOA uses a smaller set of constructs than OMT. This sometimes means that a simple concept has to be represented in a complex way.

- *Method integration.* The purpose of the method integration is to enhance a method's functional capabilities and to improve its quality or usability [Son95, BW, Sho91]. The integration is characterized either as adding new components into the method or as replacing old components with new ones. Guided by formal semantics for the methods and detailed comparison results, one can not only determine what components of one method can be integrated into another, but also avoid conflicts or inconsistencies that might arise during integration. Because of this, our work can also make contribution to method standardization.

- *Meta-CASE tools design.* Meta-CASE tools allow definition and construction of CASE tools that support arbitrary methodologies [IL97,

HFMG94, Son95]. The knowledge of differences and similarities among the methodologies can be used to guide how various software tools should communicate and interoperate with each other. With the knowledge, the Meta-CASE tool users can customize a tool effectively. In addition, the knowledge can help to reduce the redundancy of data storage for method specifications.

- *Design reuse.* The work can benefit design reuse. A part of a design developed under one method may be reused as a part of another design developed under another method, if this partial design satisfies properties in the core.

Bibliography

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Survey*, pages 159–192, June 1982.

[AI91] Derek Andrews and Darell Ince. *Practical Formal Methods with VDM*. McGraw Hill, 1991.

[AK94] V.S. Alagar and D. Kourkopoulos. (In)completeness in specifications. *Information and Software Technology*, 36(6):331–342, 1994.

[BC95] Robert H. Bourdeau and Betty H.C. Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

[BMZ91] N. Boudriga, A. Mili, and R. Zalila. Didon: System for specification validation. *Information and Software Technology*, 33(7):189–498, 1991.

[Boe84] B.W. Boehm. Verifying and validating software requirement specifications and design specification. *IEEE Software*, (1):61–72, 1984.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Application.* The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA, 1994.

[BW] Susan Bodily and Scott N. Woodfield. Integrating object-oriented methods and models. Technical Report TMCB 3361, Computer Science Department, Brigham Young University, Provo, Utah 84602.

[CCW91] J. R. Cameron, A. Campbell, and P. T. Ward. Comparing software development methods: Example. *Information and Software Technology*, 33(6):386–402, 1991.

[Che76] Peter Pin-Shan Chen. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database System*, 1(1):9–36, 1976.

[CK73] C. C. Chang and H. Jerome Keisler. *Model Theory.* North-Holland Publishing Company, 1973.

[Cly93] Stephen W. Clyde. *An Initial Theoretical Foundation for Object-Oriented Systems Analysis and Design.* PhD thesis, Department of Computer Science, Brigham Young University, 1993.

[CMR92] John Cribbs, Suzanne Moon, and Colleen Roe. *An Evaluation of Object-Oriented Analysis and Design Methodologies.* SIGS Books, 1992.

[CY91a] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, Englewood Clifss, NJ, 1991.

[CY91b] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, Englewood Clifss, NJ, 1991.

[Dic89] A. J. J. Dick. Computer aided transformation of prolog specifications. Technical Report 10-1702-01, Racal Research Ltd., Reading, Berks RG2 0SB, May 1989.

[DL94] Thomas R. Dean and David A. Lamb. A theory model core for module interconnection languages. Technical Report ISSN-0836-0235-94-370, Department of Computing and Information Science, Queen's University, Canada, October 1994.

[EKW92] D. Embley, B. Kurtz, and S. Woodfield. *Object-Oriented System Analysis: A Model-Driven Approach.* Prentice Hall, Englewood Clifss, NJ, 1992.

[Fei93] Loe Feijs. *A Formalisation of Design Methods: A λ-calculus Approach to Systems Design with an Application to Text Editing.* Ellis Horwood, Chichester, West Sussex, England, 1993.

[FK92] Robert G. Fichman and Chris F. Kemerer. Object-oriented and conventional analysis and design methodologies: Comparison and critique. *Computer*, pages 22–39, October 1992.

[Fow93] Martin Fowler. OO methods: A comparative overview. *SIGS Publications*, pages 2–4, 1993.

[Fuc92] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.

[GE86] N. Gehani and A.D. Mcgettrick (Eds.). *Software Specification Techniques.* Addison-Wesley, 1986.

[Geh82] N. Gehani. Specifications: formal and informal – a case study. *Softw. – Pract. Exp.*, pages 433–444, December 1982.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[Gra96] T.C. Nicholas Graham. A method for the formal testing of program visualization tools. In *4th IEEE Workshop on Program Comprehension*, March 1996.

[Hab91] N. Habra. Computer-aided prototyping: Transformational approach. *Information and Software Technology*, 33(9):685–697, November 1991.

[HFMG94] Donovan Hsieh and Jr. Fred M. Gilham. A novel approach toward object-oriented knowledge-based software integration within a CASE framework. *Journal of Object-Oriented Programming*, pages 25–31, September 1994.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, Cambridge, Mass., 1994. ISBN 0-262-08229-2.

[Hun93] H. Hungar. Combining model checking and theorem proving to verify parallel processes. In *Proc. 5th International Conference on Computer Aided Verification*, volume 679 of Lecture Notes in Computer Science. Springer-Verlag, 1993.

[ICO95] ICON Computing, Inc. A comparison of OOA and OOD methods, 1995. From http://www.iconcomp.com/papers/index.html.

[IL97] Hosein Isazadeh and David Alex Lamb. CASE environments and MetaCASE tools. Technical Report ISSN-0836-0227-1997-403, Department of Computing and Information Science, Queen's University, Canada, February 1997.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, Wokingham, England, 1992.

[KC93] Gerald M. Karam and Ronald S. Casselman. A cataloging framework for software development methods. *IEEE Computer*, 26(2):34–47, 1993.

[LJR89] David Alex Lamb, Nitin Jain, and Arthur Ryman. A theory-model formalization of Jackson System Development. Technical Report TR 74.051, IBM, October 1989.

[LMZ96] David A. Lamb, Andrew Malton, and Xiaobing Zhang. Applying the theory-model paradigm. Technical Report ISSN-0836-0235-1996IR-01, Department of Computing and Information Science, Queen's University, Canada, February 1996.

[LPT93] Peter Gorm Larsen, Nico Plat, and Hans Toetenel. A formal semantics of data flow diagrams. &, November 1993.

[LS92] David Alex Lamb and Kevin A. Schneider. Formalization of information hiding design methods. In *Proceedings of the 1992 CAS Conference*, pages 201–214, Toronto, Canada, November 1992. IBM Canada Ltd. Laboratory.

[LvK94] Peter Lindsay and Erik van Keulen. Case studies in the verification of specification in VDM and Z. Technical Report No. 94-3, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Queensland 4072, Australia, March 1994.

[LX93] Karl Lieberherr and Cun Xiao. Formal foundations for object-oriented data modeling. *IEEE Transaction on Knowledge and Data Engineering*, June 1993.

[MO92] J. Martin and J. Odell. *Object-Oriented Analysis and Design*. Prentice Hall, Englewood Clifss, NJ, 1992.

[Mos95] Simon Moser. Metamodels for object-oriented systems: A proposition of metamodels describing object-oriented systems at consecutive levels of abstraction. *Software – Concepts and Tools*, 16(2):63–80, 1995.

[MP91] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. In R. Rashid, editor, *Carnegie Mellon Computer Science: A 25-year Commemorative*, pages 121–156. ACM Press and Addison-Wesley, 1991.

[MP92] David E. Monarchi and Gretchen I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, September 1992.

[MP93] Mike McMorran and Steve Powell. *Z Guide for Beginners*. Blackwell Scientific Publications, 1993.

[MS95] Irwin Meisels and Mark Saaltink. The Z/EVES reference manual (draft). Technical Report TR-96-5493-03a, ORA Canada, December 1995.

[MVL92] V. Misic, D. Velasevic, and B. Lazarevic. Formal specification of a data dictionary for an extended ER data model. *The Computer Journal*, 35(6):611–622, 1992.

[Nic91] J. E. Nicholls, editor. *Z User Workshop, York 1991: Proceedings of the Sixth Annual Z User Meeting*. Springer-verlag, December 1991.

[Obj95] Object Agency, Inc. A comparison of object-oriented development methodologies, 1995. From http://www.toa.com/pub/html/mcr.lhtml.

[Par93] Jeffrey Parsons. A cognitive foundation for comparing object-oriented analysis methods. In *IEEE 26th Hawaii International Conference on System Sciences*, volume 4, pages 699–708, 1993.

[Pay93] S.E. Paynter. *The formalization of software development using MASCOT*. PhD thesis, University of Southampton, UK, December 1993.

[PT91] S. Prehn and W. J. Toetenel, editors. *VDM'91: Formal Software Development Methods: 4th International Symposium of VDM Europe*. Springer-verlag, October 1991.

[Rat97] Rational Software Corporation. *Unified Modeling Language*, July 1997. From http://www.rational.com/uml/index.html.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[RLJ91] Arthur Ryman, David Alex Lamb, and Nitin Jain. Theories and models in software design. Technical Report ISSN-0836-0227-91-319, Department of Computing and Information Science, Queen's University, Canada, October 1991.

[RS94] M.D. Rice and S.B. Seidman. A formal model for module interconnection languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, January 1994.

[Rym89] Arthur Ryman. The theory-model paradigm in software design. Technical Report TR 74.048, IBM, October 1989.

[SC93] Robert C. Sharble and Samuel S. Cohen. The object-oriented brewery: A comparison of two object-oriented development

methods. *ACM SIGSOFT Software Engineering Notes*, 18(2):60–73, April 1993.

[SFD92] L. T. Semmens, R. B. France, and T. W. G. Docker. Integrated structured analysis and formal specification techniques. *The Computer Journal*, 35(6):600–610, 1992.

[Sho91] K W Short. Methodology integration: evolution of information engineering. *Information and Software Technology*, 33(9):720–731, November 1991.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object-Oriented System Analysis: Modeling the World in Data.* Prentice Hall, Englewood Clifss, NJ, 1988.

[SM92] Sally Shlaer and Stephen J. Mellor. *Object Lifecycle: Modeling the World in States.* Prentice Hall, Englewood Clifss, NJ, 1992.

[SO92] Xiping Song and Leon J. Osterweil. Toward objective, systematic design-method comparisons. *IEEE Software*, pages 43–53, May 1992.

[SO94a] Xiping Song and Leon J. Osterweil. Experience with an approach to comparing software design methodologies. *IEEE Transactions on Software Engineering*, 20(5):364–384, May 1994.

[SO94b] Xiping Song and Leon J. Osterweil. Using meta-modeling to systematically compare and integrate object-oriented modeling techniques. July 1994.

[Sol83] Henk G. Sol. A feature analysis of information systems design methodologies: Methodological considerations. In T. W. Olle,

H. G. Sol, and C. J. Tully, editors, *Information Systems Design Methodologies: A Feature Analysis*, pages 1–8, Amsterdam, North-Holland, 1983.

[Son95] Xiping Song. A framework for understanding the integration of design methodologies. *ACM SIGSOFT Software Engineering Notes*, 20(1):46–54, January 1995.

[Spi88] J. M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, 1988.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, Hemel Hempstead, 1989.

[Tse91] T. H. Tse. *A Unifying Framework for Structured Analysis and Design Models: An Approach using Initial Algebra Semantics and Category Theory*. Cambridge University Press, 1991.

[vdG92] G.P.M. van den Goor. Formalization and comparison of six object oriented analysis and design methods. Master's thesis, Department of Information Systems, University of Nijmegen, The Netherlands, March 1992.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331. Cambridge, June 1986.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Laura Wiener. *Designing Object-Oriented Software*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, 1990.

[YP93] George Yuan and Nixon Patel. An exploration of object-oriented methodologies for system analysis and design. In *Proceedings of the Workshop on Studies of Software Design*, pages 184–198, Baltimore, Maryland, May 1993.

[Zha94] Xiaobing Zhang. A theory-model formalization of shlaer-mellor object-oriented analysis. In *Proceedings of the 1994 CAS Conference*, pages 324–333, Toronto, Canada, November 1994. IBM Canada Ltd. Laboratory.

[Zha95] Xiaobing Zhang. A comparative study of object-oriented analysis and design methods. A paper submitted to the Department of Computing and Information Science, Queen's University, in conformity with the depth requirement for the PhD program, January 1995.

[Zhu94] Yong Zhuang. Object-oriented modeling in metaview. Master's thesis, University of Alberta, Canada, 1994.

Appendix A

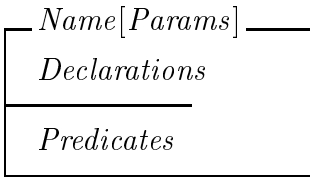
Summary of the Z Notation

This appendix provides a brief summary of Z notation (adapted from [MP93]). Since our formalization uses only a subset of the language constructs, some complex notations that are not used in the formalization are excluded from this summary.

Definitions and Declarations

Schema definition:

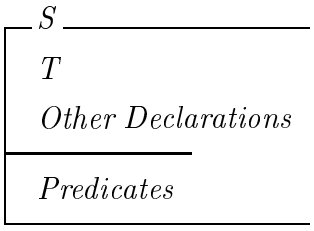
A *schema definition* assigns a name to a group of variable declarations and predicates that constrain their possible values. A schema can be used as a type. It has the following forms (with *generic parameters*):



or $Name[Params] \mathrel{\widehat=} [Declarations \mid Predicates]$

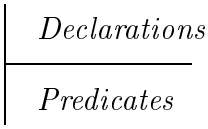
Schema inclusion:

A schema S can be included in the declarations of another schema T, in which case the declarations of S are merged with the declarations of T and the predicates of S and T are conjoined.



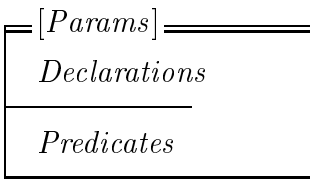
Axiomatic definition:

An *axiomatic definition* introduces one or more global variables in *Declarations*, and expresses constraints on these variables in *Predicates*.



or $\quad Declarations \mid Predicates$

Generic function definition:



given set	[ ]	$[S]$ Introduces a *given set*, S. A given set is a type.
abbreviation	==	$t == expression$ Introduces a global variable, t, whose value and type are given by *expression*.
declaration	:	$x : set\text{-}expression$ Introduces a variable, x, whose value is declared to be in the set given by *set-expression*.
datatype definition	::= |	$T ::= a \mid b \mid c$ It is equivalent to: $[T]$ $a, b, c : T \mid a \neq b \neq c \neq a$

constructor	$\langle\!\langle\ \rangle\!\rangle$	$T ::= f \langle\!\langle\ \mathit{set\text{-}expression}\ \rangle\!\rangle$ It is equivalent to: $[T]$ $f : \mathit{set\text{-}expression} \rightarrowtail T$

First-Order Logic

The following abbreviations are used in explaining logic and numbers:

S	–	set
P, Q	–	predicates

conjunction	$\wedge$	$P \wedge Q$
disjunction	$\vee$	$P \vee Q$
negation	$\neg$	$\neg P$
implication	$\Rightarrow$	$P \Rightarrow Q$
equivalence	$\Leftrightarrow$	$P \Leftrightarrow Q$
existential quantifier	$\exists$	$\exists x : S \mid P \bullet Q$ *True* if at least one element of the set S subject to P satisfies Q.
unique quantifier	$\exists_1$	$\exists_1 x : S \mid P \bullet Q$ *True* if exactly one element of the set S subject to P satisfies Q.
universal quantifier	$\forall$	$\forall x : S \mid P \bullet Q$ *True* if all elements of the set S subject to P satisfy Q.

Numbers

integers	$\mathbb{Z}$	
natural numbers	$\mathbb{N}$	$\mathbb{N} == \{x : \mathbb{Z} \mid x \geq 0\}$

positive numbers	$\mathbb{N}_1$	$\mathbb{N}_1 == \{x : \mathbb{Z} \mid x > 0\}$
integer range	$m \mathinner{..} n$	$m \mathinner{..} n == \{x : \mathbb{Z} \mid m \leq x \leq n\}$ (If $m > n$, then the set is empty.)
size	$\#$	$\#S$ The number of elements in the set S. $\#s$ The number of elements in the sequence s.
minimum of a set	min	$min\, S$ $(min\, S \in S) \land (\forall\, x : S \bullet minS \leq x)$
maximum of a set	max	$max\, S$ $(max\, S \in S) \land (\forall\, x : S \bullet minS \geq x)$
arithmetic operators	$+ \Leftrightarrow * \mathsf{div}\ \mathsf{mod}$	
arithmetic comparison	$\leq < = \neq > \geq$	

Sets

The following abbreviations are used in explaining sets:

S, T	–	sets
$x_1, x_2, ..., x_n$	–	elements of sets
P	–	predicate
E	–	expression

enumeration	$\{\ \}$	$\{x_1, x_2, ..., x_n\}$ (x_1, x_2, ..., x_n must all be of the same type.)
comprehension	$\mid \bullet$	$\{x : S \mid P\}$ The set containing those elements of S that satisfy P. $\{x : S \mid P \bullet E\}$ The set of values of E for all values of x in S which satisfy P.
membership	$\in$	$x \in S$

non-membership	$\notin$	$x \notin S$
empty set	$\emptyset$	
set equality	$=$	$S = T == (\forall s : S \bullet s \in T) \land (\forall t : T \bullet t \in S)$
set inequality	$\neq$	$S \neq T == (\exists s : S \bullet s \notin T) \lor (\exists t : T \bullet t \notin S)$
subset	$\subseteq$	$S \subseteq T == \forall x : S \bullet x \in T$
proper subset	$\subset$	$S \subset T == (S \subseteq T) \land (S \neq T)$
union	$\cup$	$S \cup T$
		$x \in S \cup T \Leftrightarrow x \in S \lor x \in T$
intersection	$\cap$	$S \cap T$
		$x \in S \cap T \Leftrightarrow x \in S \land x \in T$
difference	$\setminus$	$S \setminus T$
		$x \in S \setminus T \Leftrightarrow s \in S \land x \notin T$
power set	$\mathbb{P}$	$\mathbb{P} S$
		The set of all subsets of S.
non-empty power set	$\mathbb{P}_1$	$\mathbb{P}_1 S == \mathbb{P} S \setminus \{\emptyset\}$
finite subset	$\mathbb{F}$	$\mathbb{F} S$
		The set of all finite subsets of S.
non-empty finite subset	$\mathbb{F}_1$	$\mathbb{F}_1 S == \mathbb{F} S \setminus \{\emptyset\}$
distributed union	$\bigcup$	$\bigcup S == \{x : T \mid (\exists s : S \bullet x \in s)\}$
		where S is of type $\mathbb{P}(\mathbb{P} T)$.
distributed intersection	$\bigcap$	$\bigcap S == \{x : T \mid (\forall s : S \bullet x \in s)\}$
		where S is of type $\mathbb{P}(\mathbb{P} T)$.

Relations

The following abbreviations are used in explaining relations:

A, B, S	–	sets
a, b	–	elements of sets
R, F, G	–	relations

Cartesian product	$\times$	$A \times B == \{a : A;\ b : B \bullet (a, b)\}$
binary relation	$\leftrightarrow$	$A \leftrightarrow B == \mathbb{P}(A \times B)$
ordered pair (maplet)	$\mapsto$	$a \mapsto b == (a, b)$
first of pair	*first*	$\mathit{first}\,(a, b) = a$
second of pair	*second*	$\mathit{second}\,(a, b) = b$
domain	dom	$\mathrm{dom}\,R == \{a : A;\ b : B \mid (a, b) \in R \bullet a\}$ where $R : A \leftrightarrow B$.
range	ran	$\mathrm{ran}\,R == \{a : A;\ b : B \mid (a, b) \in R \bullet b\}$ where $R : A \leftrightarrow B$.
relational inverse	$^\sim$	$R^\sim == \{a : A;\ b : B \mid (a, b) \in R \bullet (b, a)\}$ where $R : A \leftrightarrow B$.
composition	$\fatsemi$	$R \fatsemi F == \{a : A;\ b : B;\ c : C \mid (a, b) \in R \land (b, c) \in F \bullet (a, c)\}$ where $R : A \leftrightarrow B$ and $F : B \leftrightarrow C$.
background composition	$\circ$	$R \circ F == F \fatsemi R$
domain restriction	$\lhd$	$S \lhd R == \{a : A;\ b : B \mid (a, b) \in R \land a \in S\}$ where $R : A \leftrightarrow B$ and $S \subseteq A$.
domain subtraction	$\ndres$	$S \ndres R == \{a : A;\ b : B \mid (a, b) \in R \land a \notin S\}$ where $R : A \leftrightarrow B$ and $S \subseteq A$.
range restriction	$\rhd$	$R \rhd S == \{a : A;\ b : B \mid (a, b) \in R \land b \in S\}$ where $R : A \leftrightarrow B$ and $S \subseteq B$.
range subtraction	$\nrres$	$R \nrres S == \{a : A;\ b : B \mid (a, b) \in R \land b \notin S\}$ where $R : A \leftrightarrow B$ and $S \subseteq B$.
relational image	$(\!\mid\ \mid\!)$	$R(\!\mid S \mid\!) == \{a : A;\ b : B \mid (a, b) \in R \land a \in S \bullet b\}$ where $R : A \leftrightarrow B$ and $S \subseteq A$.
overriding	$\oplus$	$F \oplus G == ((\mathrm{dom}\,G \ndres F) \cup G)$ The relation F overridden by the relation G.

identity relation	id	$\mathrm{id}\, S == \{s : S \bullet s \mapsto s\}$
iteration	^{k}	$R^0 == \mathrm{id}\, A;\ R^k == R^{k-1} \fatsemi R$
transitive closure	$^+$	$R^+ == R^1 \cup ... \cup R^n \cup ...$ where the type of R is $\mathbb{P}(A \times A)$.
reflexive transitive closure	*	$R^* == R^0 \cup R^+$ where the type of R is $\mathbb{P}(A \times A)$.

Functions

The following abbreviations are used in explaining functions:

A, B	–	sets
D	–	declaration
P	–	predicate
E	–	expression

partial function	$\pfun$	$A \pfun B == \{r : A \rel B \mid (\forall a : A;\ b_1, b_2 : B \mid (a, b_1) \in r \land (a, b_2) \in r \bullet b_1 = b_2)\}$
total function	$\fun$	$A \fun B == \{f : A \pfun B \mid \mathrm{dom}\, f = A\}$
finite partial function	$\ffun$	$A \ffun B == \{f : A \pfun B \mid \mathrm{dom}\, f \in \mathbb{F}\, A\}$
partial injection	$\pinj$	$A \pinj B == \{f : A \pfun B \mid f^\sim = B \pfun A\}$
total injection	$\inj$	$A \inj B == A \pinj B \cap A \fun B$
finite partial injection	$\finj$	$A \finj B == A \pinj B \cap A \ffun B$
partial surjection	$\psurj$	$A \psurj B == \{f : A \pfun B \mid \mathrm{ran}\, f = B\}$
total surjection	$\surj$	$A \surj B == \{f : A \psurj B \mid f \in A \fun B\}$
bijection	$\bij$	$A \bij B == \{f : A \inj B \mid f \in A \surj B\}$
lambda expression	λ	$\lambda D \mid P \bullet E == \{D \mid P \bullet t \mapsto E\}$ where t is the characteristic tuple of the declaration D.

choice (mu)	μ	$(\mu\, D \mid P \bullet E)$ The value of the expression E where the variables of the declaration D have values determined by predicate P.

Sequences

The following abbreviations are used in explaining sequences:

$S, S_1, S_2, ..., S_n, X, U, V$	–	sets
$x_1, x_2, ..., x_n$	–	elements of sets
s, t	–	sequences

enumeration	$\langle\ \rangle$	$\langle x_1, x_2, ..., x_n\rangle == \{1 \mapsto x_1, 2 \mapsto x_2, ..., n \mapsto x_n\}$
finite sequence	seq	$\operatorname{seq} X == \{f : \mathbb{N} \pfun X \mid \operatorname{dom} f = 1\,..\,\#f\}$
non-empty sequence	seq_1	$\operatorname{seq}_1 X == \operatorname{seq} X \setminus \{\emptyset\}$
injective sequence	iseq	$\operatorname{iseq} X$ $s = \operatorname{iseq} X \Rightarrow (\#s = \#\operatorname{ran} s)$
concatenation	$\frown$	$s \frown t == s \cup (\lambda\, i : \#s + 1\,..\,\#s + \#t \bullet$ $i \mapsto t(i \Leftrightarrow \#s))$
index restriction	$\upharpoonleft$	$U \upharpoonleft s$ The sequence s restricted to just those indexes that appear in the set U.
sequence restriction (filtering)	$\upharpoonright$	$s \upharpoonright V$ The sequence s restricted to just those elements that appear in the set V.
disjoint	disjoint	$\mathsf{disjoint}\ \langle S_1, S_2, ..., S_n\rangle == S_1 \cap S_2 \cap ... \cap S_n = \emptyset$
partition	partition	$\langle S_1, S_2, ..., S_n\rangle\ \mathsf{partition}\ S ==$ $\mathsf{disjoint}\ \langle S_1, S_2, ..., S_n\rangle \land S = \bigcup\{S_1, S_2, ..., S_n\}$

Schema operators

The following abbreviations are used in explaining schema operators:

S, T	–	schemas
$sigS, sigT$	–	signatures of schemas S and T
$predS, predT$	–	predicates of schemas S and T

conjunction	$\wedge$	$S \wedge T == [sigS;\ sigT \mid predS \wedge predT]$
disjunction	$\vee$	$S \vee T == [sigS;\ sigT \mid predS \vee predT]$
negation	$\neg$	$\neg\ S == [sigS \mid \neg\ predS]$
implication	$\Rightarrow$	$S \Rightarrow T == [sigS;\ sigT \mid predS \Rightarrow predT]$
equivalence	$\Leftrightarrow$	$S \Leftrightarrow T == [sigS;\ sigT \mid predS \Leftrightarrow predT]$
binding	θ	θS A binding between the components of S and their current values.
Delta	Δ	ΔS Values of the variables in S may change after an operation.
Xi	Ξ	ΞS Values of the variables in S do not change after an operation.
renaming	$[\ /\ ]$	$S[new/old]$ A schema consisting of schema S with component *old* renamed to *new*.
schema hiding	$\setminus\ (\ )$	$S \setminus (v_1, v_2, ...)$ A schema consisting of schema S with variables $v_1,\ v_2,\ ...$ removed from the signature and existentially quantified in the property.

selection	.	$w.x$ It denotes the x component of w, where $w : S$, S is a schema, and x is a variable declared in S.

Appendix B

Rules for Subdividing the Core

This Appendix presents the rules for subdividing the signatures in a core theory into characterizing sets. It also presents the characterizing sets in $Core_{OM}$ and $Core_{IM}$.

We introduce several notations to be used in the discussion:

Definition B.1 Given two comparable variables $T_1.v$ and $T_2.v$.

1. If $T_1.v$ and $T_2.v$ are in Sig_{Core}, then

   $$T_1.v =_{sem} T_2.v$$

2. If $T_1.v$ and $T_2.v$ are in Sig_{T_1} (or Sig_{T_2}), then

   $$T_1.v \subset_{sem} T_2.v \quad (\text{or } T_2.v \subset_{sem} T_1.v)$$

3. If $T_1.v$ and $T_2.v$ are in $Sig_{T_1 \bullet T_2}$, then

   $$T_1.v \neq_{sem} T_2.v$$

B.1 Basic Rules

The comparison for classifying the core into the characterizing sets proceeds incrementally. At each step, we take a pair of comparable variables from the

core, and a set of predicates that impose constraints on the variables, and then do the comparison guided by the following rules.

1. Given two comparable variables $T_1.v$ and $T_2.v$. If

   - there is no constraint on the variables, or
   - the two sets of comparable predicates $T_1.P$ and $T_2.Q$ that impose constraints on the variables are equivalent, and the free variables in the predicates are in Sig_{Core},

   then the two variables belong to Sig_{Core}. That is,

   $$T_1.v =_{sem} T_2.v$$

2. Given two comparable variables $T_1.v$ and $T_2.v$, and two sets of comparable predicates $T_1.P$ and $T_2.Q$ that impose constraints on the variables. If

   - $T_1.P \simeq_{sem} T_2.Q$ when some free variables in the predicates are already in Sig_{T_1} and the rest of the free variables are in Sig_{Core},

   or if

   - $T_1.P \sim_{sem} T_2.Q$ when
     - some free variables in $T_2.Q$ take certain restricted values, and/or
     - some free variables in the predicates are already in Sig_{T_1}, and
     - the rest of the free variables are in Sig_{Core},

   or if

   - $T_1.Pred \models T_2.Q[T_1/T_2]$ but there is at least one $T_1.p \in T_1.P$ so that $T_2.Pred \not\models T_1.p[T_2/T_1]$, and free variables in the predicates are in Sig_{Core} or Sig_{T_1}, or some free variables in $T_2.Q$ take restricted values.

then the two variables belong to Sig_{T_1}. That is,

$$T_1.v \subset_{sem} T_2.v$$

3. Given two comparable variables $T_1.v$ and $T_2.v$, and two sets of comparable predicates $T_1.P$ and $T_2.Q$ that impose constraints on the variables. If

   - $T_1.P \simeq_{sem} T_2.Q$ when some free variables in the predicates are already in Sig_{T_2} and the rest of the free variables are in Sig_{Core},

   or if

   - $T_1.P \sim_{sem} T_2.Q$ when
     - some free variables in $T_1.P$ take certain restricted values, and/or
     - some free variables in the predicates are already in Sig_{T_2}, and
     - the rest of the free variables are in Sig_{Core},

   or if

   - $T_2.Pred \models T_1.P[T_2/T_1]$ but there is at least one $T_2.q \in T_2.Q$ so that $T_1.Pred \not\models T_2.q[T_1/T_2]$, and free variables in the predicates are in Sig_{Core} or Sig_{T_2}, or some free variables in $T_1.P$ take restricted values.

   then the two variables belong to Sig_{T_2}. That is,

   $$T_2.v \subset_{sem} T_1.v$$

4. Given two comparable variables $T_1.v$ and $T_2.v$, and two sets of comparable predicates $T_1.P$ and $T_2.Q$ that impose constraints on the variables. If none of the conditions in rule 1 to rule 3 is satisfied, then the two variables belong to $Sig_{T_1 \bullet T_2}$. That is,

   $$T_1.v \neq_{sem} T_2.v$$

5. Given two comparable variables $T_1.v$ and $T_2.v$. Sometimes, we can directly prove $T_1.v =_{sem} T_2.v$, or $T_1.v \subset_{sem} T_2.v$, or $T_1.v \neq_{sem} T_2.v$, from the predicates that impose constraints on them and from the knowledge about the free variables in the predicates, without proving the equivalence of the predicates. We can treat "$=_{sem}$", "$\subset_{sem}$", and "$\neq_{sem}$" as normal set operators.

B.2 Adjustment Rules

The comparison process is iterative. During the course of comparison, for each comparable variable, more and more relationships between the variable and other variables are considered, and more and more constraints on the variable are discovered. All these constraints together finally determine which characterizing set the variable should belong to. Thus, it may be necessary to move the variable from one characterizing set to another when more constraints are imposed on it. We cannot completely avoid such adjustments, not because it is inevitable in theory, but is due to the nature of our stepwise comparison mechanism. The adjustment is guided by the following rules.

6. A comparable variable can only be moved from Sig_{Core} to Sig_{T_1} (or Sig_{T_2}) to $Sig_{T_1 \bullet T_2}$, but cannot be moved in the reverse direction.

7. Given two variables $T_1.v$ and $T_1.w$, where $T_1.w \subseteq T_1.v \times T_1.v' \times T_1.v'' \times \ldots$. $T_1.v$ is previously in certain characterizing set, and at the current stage we need to determine whether $T_1.v$ should still remain in the same set or be moved to another one, based on the current result of $T_1.w$. If $T_1.w$ further restricts the values that $T_1.v$ can take on, then the status of $T_1.v$ will have to change accordingly. Otherwise, there is no change to $T_1.v$.

   Example:

(a) $OM.attribute$ and $IM.attribute$ are in Sig_{Core} at the beginning of the comparison. After we compare $OM.attrDefinedIn$ ($OM.attrDefinedIn \subseteq OM.attribute \times OM.entiey$) and $IM.attrDefinedIn$ ($IM.attrDefinedIn \subseteq IM.attribute \times IM.object$), it is obtained that $IM.attribute$ cannot be the empty set but $OM.attribute = \emptyset$ is possible. Therefore, comparable variables $OM.attribute$ and $IM.attribute$ are moved to Sig_{IM}, that is,

$$IM.attribute \subset_{sem} OM.attribute$$

(b) $OM.opDefinedIn$ satisfies the predicate $OM.opDefinedIn \in OM.operation \leftrightarrow OM.class$. This predicate gives no further constraint on $OM.class$, thus, $OM.class$ remains unchanged.

8. Given that variable $T_1.v'$ is a subset of variable $T_1.v$, i.e., $T_1.v' \subseteq T_1.v$. $T_1.v$ may have to be moved to a different characterizing set based on the previous result about $T_1.v'$ and $T_1.v$. Table B.1 illustrates the rule. The first column shows the characterizing set which $T_1.v$ is in, and the first row shows the characterizing set which $T_1.v'$ is in. The rest shows which characterizing set $T_1.v$ as well as its counterpart $T_2.v$ should be moved to. The multiple characterizing sets listed in a table item mean that different constraints imposed on $T_1.v'$ and $T_1.v$ may yield different results.

9. Given that variable $T_1.v'$ is a subset of variable $T_1.v$, i.e., $T_1.v' \subseteq T_1.v$. $T_1.v'$ may have to be moved to a different characterizing set based on the previous result about $T_1.v'$ and $T_1.v$. Table B.2 illustrates the rule. The first column shows the characterizing set which $T_1.v'$ is in, and the first row shows the characterizing set which $T_1.v$ is in. The rest shows which characterizing set $T_1.v'$ as well as its counterpart $T_2.v'$ should be moved to.

Table B.1: Adjustment rule for $T_1.v$ based on the previous result of $T_1.v$ and $T_1.v'$.

$T_1.v$	$T_1.v'\,(T_1.v' \subseteq T_1.v)$				
	Sig_{Core}	Sig_{T_1}	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$	Ext_{T_1}
Sig_{Core}	Sig_{Core}	Sig_{T_1}	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$	Sig_{Core}, Sig_{T_2}
Sig_{T_1}	Sig_{T_1}	Sig_{T_1}	$Sig_{T_1}, Sig_{T_1 \bullet T_2}$	$Sig_{T_1}, Sig_{T_1 \bullet T_2}$	$Sig_{T_1}, Sig_{T_1 \bullet T_2}$
Sig_{T_2}	Sig_{T_2}	$Sig_{T_2}, Sig_{T_1 \bullet T_2}$	Sig_{T_2}	$Sig_{T_2}, Sig_{T_1 \bullet T_2}$	Sig_{T_2}
$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$

Table B.2: Adjustment rule for $T_1.v'$ based on the previous result of $T_1.v'$ and $T_1.v$.

$T_1.v'$	$T_1.v\,(T_1.v' \subseteq T_1.v)$			
	Sig_{Core}	Sig_{T_1}	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$
Sig_{Core}	Sig_{Core}	Sig_{Core}, Sig_{T_1}	Sig_{Core}, Sig_{T_2}	any
Sig_{T_1}	Sig_{T_1}	Sig_{T_1}	$Sig_{T_1}, Sig_{T_1 \bullet T_2}$	$Sig_{T_1}, Sig_{T_1 \bullet T_2}$
Sig_{T_2}	Sig_{T_2}	$Sig_{T_2}, Sig_{T_1 \bullet T_2}$	Sig_{T_2}	$Sig_{T_2}, Sig_{T_1 \bullet T_2}$
$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$

Note: "any" means any of the characterizing sets.

10. A variable $T_1.v$ sometimes has constraints from different point of view in several schemas. We may compare these schemas at several steps. In this case, we get several separate results about $T_1.v$. To put these results together, we use the rule illustrated in Table B.3. The first column shows the characterizing set $T_1.v$ is previously in, and the first row shows the current result about $T_1.v$. The rest shows which characterizing set $T_1.v$ as well as its counterpart $T_2.v$ should be moved to.

11. If a non-comparable predicate $T_1.p$ is about properties of a comparable variable $T_1.v$, then we may need to adjust $T_1.v$'s status, because the predicate may restrict the relationships between the variable and other variables and hence restrict the values the variable can take on. On the

Table B.3: Adjustment rule for $T_1.v$ based on its previous and current results

$T_1.v$ (previous)	$T_1.v$ (current)			
	Sig_{Core}	Sig_{T_1}	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$
Sig_{Core}	Sig_{Core}	Sig_{T_1}	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$
Sig_{T_1}	Sig_{T_1}	Sig_{T_1}	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$
Sig_{T_2}	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$
$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$

other hand, theory T_2 does not have a corresponding predicate, so it does not have similar restrictions on any variable in T_2. If this is the case, Table B.4 illustrates the rule for adjustment. Again, $T_1.v$'s counterpart $T_2.v$ should move accordingly.

Table B.4: Adjustment rule for $T_1.v$ based on a non-comparable predicate imposing constraints on $T_1.v$.

$T_1.v$ (before)	Sig_{Core}	Sig_{T_1}	Sig_{T_2}	$Sig_{T_1 \bullet T_2}$
$T_1.v$ (after)	Sig_{T_1}	Sig_{T_1}	$Sig_{T_1 \bullet T_2}$	$Sig_{T_1 \bullet T_2}$

However, when $T_1.v \in Core_{T_1}.Sig_{T_2}$, there is one exception: if the constraint is on those models of T_1 that do not have corresponding models in T_2, then the status of $T_1.v$ does not change.

12. Once a variable v is moved from one characterizing set to another, we have to reconsider other related variables in the subtheory that have already been compared, and make the corresponding changes to these variables by repeatedly applying rule 1 to rule 11. This process of updating the characterizing sets terminates when no variable needs a further change.

B.3 Characterizing Sets of OM and IM

The tables in this section illustrate the comparison result. Each table is for one characterizing set. Sig_{OM} is empty.

Table B.5: The variables in characterizing set Sig_{Core}

variable in OM	constraint	variable in IM	constraint
role		*role*	
multiplicityOfRole		*multiplicityOfRole*	
roleDefinedIn		*roleDefinedIn*	
hasRole		*hasRole*	
singularRole		*single*	
optionalRole		*conditionSingle*	
onemulRole (aux)		$multiple^{(2)}$	
optmulRole (aux)		$conditionMultiple^{(2)}$	
$multipleRole^{(2)}$			
$orderedRole^{(2)}$			

Table B.5: The variables in characterizing set Sig_{Core} (continued)

variable in *OM*	constraint	variable in *IM*	constraint
association		*relationship*	
binaryAssociation		*binaryRelationship*	
one_oneAssoc		R_{11}	
one_onemulAssoc (aux)		${R_{1M}}^{(2)}$	
one_optmulAssoc (aux)		${R_{1Mc}}^{(2)}$	
one_mulAssoc$^{(2)}$			
onemul_onemulAssoc (aux)		${R_{MM}}^{(2)}$	
onemul_optmulAssoc (aux)		${R_{MMc}}^{(2)}$	
optmul_optmulAssoc (aux)		${R_{McMc}}^{(2)}$	
mul_mulAssoc$^{(2)}$			
one_optAssoc		R_{11c}	
opt_onemulAssoc (aux)		${R_{1cM}}^{(2)}$	
opt_optmulAssoc (aux)		${R_{1cMc}}^{(2)}$	
opt_mulAssoc$^{(2)}$			
opt_optAssoc		R_{1c1c}	
qualifiedAssociation$^{(2)}$			
qualify$^{(2)}$			
baseAssoc$^{(2)}$		*baseRelationship* (aux)	
derivedAssoc		*comRelationship*	
assocDeriving$^{(3)}$		*composedBy*$^{(3)}$	
groupedAssociation$^{(2)}$			
genAssociation		*genRelationship*	
aggAssociation$^{(2)}$			
disjointGen$^{(2)}$		*disjointGen* (aux)	
aggregation$^{(2)}$			
parent$^{(2)}$			
gen_parent (aux)		*superRole*	
child$^{(2)}$			
gen_child (aux)		*subRole*	
fixedAgg$^{(2)}$			
variableAgg$^{(2)}$			
recursiveAgg$^{(2)}$			
module$^{(2)}$			

Table B.6: The variables in characterizing set Sig_{IM}

variable in OM	constraint	variable in IM	constraint
$entity$		$entity$	
$feature$	$ccPred_{IM}$	$feature$	$ccPred_{IM}$
$definedIn^{(2)}$	$ccPred_{OM}$ $ccPred_{IM}$	$definedIn$ (aux)	$ccPred_{IM}$
$hasFeature$	$ccPred_{OM}$	$hasFeature$	$ccPred_{IM}$
$class$	$ccPred_{IM}$	$object$	$ccPred_{IM}$
$baseClass^{(2)}$		$baseObject$ (aux)	
$joinClass^{(2)}$		$joinObject$ (aux)	
$concreteClass^{(2)}$		$concreteObject$ (aux)	
$leafClass^{(2)}$		$leafObject$ (aux)	
		$formalizer^{(6)}$	
$associationClass$		$associativeObject$	
$attribute$	$ccPred_{IM}$	$attribute$	$ccPred_{IM}$
$attrDefinedIn$	$ccPred_{OM}$	$attrDefinedIn$	$ccPred_{IM}$
		$hasAttribute^{(1)}$	
$descriptiveAttr$ (aux)		$descriptiveAttr^{(2)}$	
$namingAttr$ (aux)		$namingAttr^{(2)}$	
$baseAttr^{(2)}$		$baseAttr$ (aux)	
$valueRange$	$ccPred_{OM}$	$domainSpec$	$ccPred_{IM}$
$valueOfType^{(4)}$			
$typeOfAttr^{(4)}$			
$restrict^{(4)}$			
$generalRestrict^{(4)}$			
$constraint^{(4)}$			
key		$identifier$	
$hasKeyElement$	$ccPred_{OM}$	$hasIdElement$	$ccPred_{IM}$
$keyedEntity^{(2)}$		$idedEntity$ (aux)	$ccPred_{IM}$
$isKeyOf$		$isIdOf$	
$theKey^{(1)}$			
$hasPreferredKey$ (aux)		$hasPreferredId^{(2)}$	
$playerOfRole$		$playerOfRole$	
$containedIn^{(2)}$			

Table B.6: The variables in characterizing set Sig_{IM} (continued)

variable in OM	constraint	variable in IM	constraint
$grouping^{(2)}$			
$generalization$	$ccPred_{IM}$	$generalization$	$ccPred_{IM}$
$single^{(2)}$			
gen_single (aux)		$supertype$	
$group^{(2)}$			
gen_group (aux)		$subtype$	
$collectedIn^{(2)}$			
$gen\_collectedIn$ (aux)		$collectedIn$	
$isKindOf$		$isKindOf$	
$isDescendentOf$		$inherit$	
$isAncestorOf^{(1)}$			
$extension^{(2)}$			
$isPartOf^{(2)}$			

Table B.7: The variables in characterizing set $Sig_{OM \bullet IM}$

variable in OM	constraint	variable in IM	constraint
$modeledAs$	$ccPred_{OM}$	$formalizedBy$	$ccPred_{OM}$
		$singleOccurrence^{(2)}$	