An Algebraic Approach to Object-Oriented Software Engineering

Shin NAKAJIMA

November 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
at
The University of Tokyo

Abstract

Formal Description Techniques (FDTs) and *Object Technology* (OT) form two main streams as basis for improving productivity and quality of software systems. FDTs and OT, however, have a long history starting around early 1970's. FDT is a system development method that uses mathematically-based formal specification languages. OT is another system development method. Its basis is the concept of *object*, which constitutes the world to be modeled and at the same time is a primitive computational entity in the computer system. Although the origin of FDT and OT is quite divergent, the two are now converging.

Object, the basis of the object-oriented paradigm, is a data abstraction, and is a functional closure that encapsulates the internal states and the accompanying procedures. OT now sees a widespread use in industry because the technology has evolved in a variety of ways to meet demands necessary for a practical use. The variation includes (1) object-oriented paradigm and programming, (2) high-level reuse architecture such as object-oriented frameworks, and (3) object-oriented modeling methods.

The algebraic specification technique, a branch of FDTs, has its basis on data abstraction and provides a concept of *signature* for a basis for precise descriptions of object interface. This seems to imply that the algebraic technique is expected to be an adequate FDT for object-oriented software system. The algebraic technology, however, sees an extensive work on theoretical aspects and language design only. Algebraic languages and tools, matured enough for practical use, are not many. Applying the technology to software development process is hardly known.

The dissertation discusses a new approach to object-oriented software development method with CafeOBJ, which is one of the most advanced language/tools that have potential for use in industry. With concrete case studies, the dissertation addresses the issues of the algebraic specification technology: (a) expressiveness, (b) module decomposition, and (c) use in development process.

Chapter 2 presents a survey on the subject matter, and summarizes the background of this work. Chapter 3 is primarily concerned with the first issue, "expressiveness". It discusses how one represents basic object-oriented algebraic specifications in CafeOBJ. A novel idea is to integrate the Maude model with the Ambient calculus. The purpose is to provide a machinery to manipulate a set of closely related objects and messages as one unit. The technique allows to write specifications representing (1) semantics of distributed software architecture, and (2) a high level concurrency control for shared resources. Thus it extends the expressibility of the

basic Maude model.

Chapter 4 proposes an object-oriented modeling method for obtaining CafeOBJ descriptions, and covers all the three issues above. It first identifies what design models are needed to represent the concepts in scenario-based object-oriented modeling methods, and then presents how one encodes the design models in CafeOBJ. It contributes to the second issue on "module decomposition" because the modeling method provides a guideline for identifying CafeOBJ modules that are clearly traceable from entities in the problem domain. It also mentions about the third issue in that the proposed method is an integration with the conventional informal modeling method at an early stage of software development.

Chapter 5 reports a practice of applying CafeOBJ to describing the ODP trader specifications in the ITU-T recommendation document. And it is concerned with the first two issues, "expressiveness" and "module decomposition." The ODP trader has been recognized as an important application of FDTs, and this work is a first attempt of applying algebraic specification language to the ODP trader description. The case study demonstrates that CafeOBJ is expressive enough to represent both the information and computational viewpoints. And the resultant CafeOBJ modules are clearly traceable from the recommendation specifications because they reflect the organization of the original ITU-T recommendation document.

Chapter 6 also covers all the issues by using CafeOBJ in development of object-oriented frameworks for a trading server. The discussion covers its whole development process including both the problem analysis and the Java program construction phases. The case study includes a proposal of a problem-oriented design method, which identifies essential design models that are necessary to represent the design artifact of the trading server. Then identified design models are represented in CafeOBJ, which shows expressiveness of the language. CafeOBJ is used as a design validation checker. The design method can also be considered as a problem-oriented guideline for identifying the CafeOBJ modules, which contributes as a module decomposition strategy.

Chapter 7 summarizes the contribution of this work, and presents a list of future works on applying CafeOBJ to object-oriented software development process.

Acknowledgments

I would like to express my sincere gratitude to Professor Tetsuo Tamai for the guidance and encouragement he has given me in the preparation of this dissertation. His insights on software design and on the inter-relationship between formal description techniques and conventional design methods drove me to clarify my vague ideas. He has also read this dissertation several times, and his comments have improved it immensely on each iteration.

I am most certainly grateful to the other members of the dissertation committee, Professor Satoru Kawai, Professor Kazunori Yamaguchi, Professor Masami Hagiya, and Professor Kokichi Futatsugi of Japan Advanced Institute of Science and Technology. Their comments were especially helpful in improving this dissertation.

In addition to act as a dissertation committee member, Professor Kokichi Futatsugi had introduced me to the algebraic specification technique. Through several research meetings that many well-known international researchers participated in, he taught me how one pursued research problems in academia. He encouraged me to use CafeOBJ as a research vehicle and gave me valuable suggestions on this work. I also thank for all members of the CafeOBJ development team, especially Dr. Ataru T. Nakagawa and Toshimi Sawada of SRA. Without their continuous efforts, I can not use CAFE, the CafeOBJ specification construction environment.

Professor Kenji Ohmori of Hosei University had opened the door for me to the object-oriented software research area. When he was with the NEC Corporation, he encouraged me to study the object technology that was emerging in the year of 1983. The object technology became an essential ingredient of this work.

I am also deeply indebted to those who have supervised me at the NEC Corporation. Mr. Takashi Torii, the President of the NEC Field Engineering Service, gave me a chance of a two-years visit to the Computer System Research Laboratory of NEC. As a senior manager of the laboratory, Dr. Katsuya Hakozaki, now a Professor at the University of Electro-Communications, put me in the research group headed

by Dr. Kenji Ohmori, which was really an event to start my career as a research engineer at NEC. Dr. Masahiro Yamamoto, now a Professor at Hosei University, Dr. Satoshi Goto, a Vice President of the NEC Laboratories, Dr. Nobuhiko Koike, now a Professor at Hosei University, and Dr. Takeshi Yoshimura, all taught me how to carry out research work in industry. Their supervision and strong guidance helped me much continue this research work at NEC.

I was lucky to have many colleagues in NEC who had interests in the object technology that this dissertation was concerned with. Many thanks go to Youichi Miyashita, Akihiko Konagaya, Shingo Fukui, Kazuo Otake, Toshio Tonouchi, Masahiro Tomono, Atsuhiko Yamanaka, and Tomoji Kishi. Discussions with them helped much to deepen my understanding of the object technology.

Last but not least, special thanks go to my family. I thank my wife Masako and our daughter Hanako who gave me numerous moments of joy and the strength required to finish this research and to write the dissertation. Finally, I thank my parents, Dr. Junsuke and Junko Higashino, and Dr. Hiromichi and Noriko Nakajima for their love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Algebraic Approach to Object Technology

Formal Description Techniques (FDTs) and *Object Technology* (OT) form two main streams as basis for improving productivity and quality of software systems. It was only 1990's when the technology gained a wide acceptance as potential sources for the improvement. FDTs and OT, however, have a long history starting around early 1970's. FDT is a system development method that uses mathematically-based formal specification languages. OT is another system development method. Its basis is the concept of *object*, which constitutes the world to be modeled and at the same time is a primitive computational entity in the computer system. Although the origin of FDT and OT is quite divergent, the two are now converging.

Object, the central idea of OT, is a procedure-embedded datatype, which knows how to implement any given operation on itself. Object is accompanied with the concept of class and instance, which is attributed to Simula. OT now sees a widespread use in industry because the technology has evolved in a variety of ways to meet demands necessary for a practical use. The variation includes (1) object-oriented paradigm and programming, (2) high-level reuse concepts such as object-oriented frameworks, and (3) object-oriented modeling methods. In order to realize the convergence of OT with FDT, one should fully understand the nature and the above mentioned variations of object-oriented software technology. And the presumed technology should be evaluated in view of applying itself to the development of real-world software system in the industrial settings.

The algebraic specification technique, a branch of FDTs, has its basis on data

abstraction which was first recognized concretely as the *class* concept of Simula language. And the algebraic specification method provides techniques for the structured specification, validation and analysis of abstract datatypes.

From a historical perspective, Simula is a common ancestor of both the algebraic specification techniques and the object-oriented programming languages. More importantly, the algebraic specification techniques provide a concept of *signature* for a basis for precise descriptions of object interface. However, the most work on the algebraic technology came from interests in the academic research community and were mainly concerned with theoretical frameworks. Less attention has been paid to practical aspects. Among the algebraic languages and tools, CafeOBJ/CAFE is one of the most advanced language/tools matured enough for practical use. As compared with other FDTs such as the Z notation, VDM or the B method, algebraic specification techniques are hardly applied to software development process. The issues that hinder one from practical application would be the following three points: (1) expressiveness, (2) module decomposition, and (3) use in development process.

1. Expressiveness

   In applying CafeOBJ at an early stage of object-oriented software development, one describes design artifacts from various viewpoints. Each viewpoint may need a different design model or computational model. It must be confirmed that CafeOBJ can represent such important design models and that the encoding, if necessary, is desirably straight-forward.

2. Module Decomposition

   The issue is important in writing a medium to large scale specification, which is composed of a lot of CafeOBJ modules. A guideline is necessary for decomposing a given problem into a set of well-defined CafeOBJ modules. The guideline may be problem-specific in order to provide a sharply focused help.

3. Use in Development Process.

   The issue is related to the management of a software development process with CafeOBJ. It is clear that one cannot develop software systems with CafeOBJ alone, and that one may use conventional modeling methods and programming languages as well. It needs a guideline for divisions of labour between the conventional technologies and CafeOBJ.

It needs an extensive work on the way of applying CafeOBJ to development of practical software in order to show that CafeOBJ, or the algebraic specification technique generally, is an industrial-strength formal description technique.

1.2 Outline of the Dissertation

This dissertation adopts an algebraic logic language CafeOBJ that already has some of object-oriented concepts, and discusses how one integrates CafeOBJ in a software development process of object-oriented software. In order to concretely tackle three issues (expressivenss, module decomposition, and use in development process), this dissertation takes an approach of case studies.

First, Chapter 2 presents a survey of the existing research work from two perspectives, the formal description techniques including CafeOBJ and the object technology. As a start of the research, a careful analysis of the object-oriented technology is conducted to identify concrete problems in regard to object-oriented software development. The problems to be focused are what one is faced with in regard to developing object-oriented frameworks (OO-FWKs).

This research includes four approaches: (1) Maude/Ambient, (2) GILO, (3) ODP Trader Recommendation, and (4) Trading Server Development. These cover the most important four different aspects in relation to applying CafeOBJ to development of object-oriented frameworks.

Chapter 3 (Maude/Ambient) is primarily concerned with the first issue, "expressiveness." It discusses how one represents basic object-oriented algebraic specifications in CafeOBJ. Later chapters employ the encoding techniques presented in Chapter 3 in the respective case studies.

Chapter 4 (GILO) covers all the three issues above. It first identifies what design models are needed to represent the concepts in scenario-based object-oriented modeling methods, and then presents how one encodes the models in CafeOBJ. It contributes to the second issue on "module decomposition" because the modeling method provides a guideline for identifying CafeOBJ modules that are clearly traceable from entities in the problem domain. It also deals with the third issue in that the proposed method is an integration with the conventional informal modeling method at an early stage of development.

Chapter 5 (ODP Trader Recommendation) is concerned with the first two issues, "expressiveness" and "module decomposition." It applies CafeOBJ to describe the information and computational viewpoint specifications of the ODP trader. The

ODP trader has been recognized as an important application of FDTs. The case study demonstrates that CafeOBJ is expressive enough to represent both viewpoints. And the resultant CafeOBJ modules are well-structured because they reflect the organization of the recommendation document. The outcome of Chapter 5 contributes to the problem analysis activity conducted in Chapter 6.

Chapter 6 (Trading Server Development) also covers all the issues by using CafeOBJ in development of object-oriented frameworks for a trading server. The discussion covers its whole development process including both the problem analysis and Java program construction phases. The case study includes a proposal of a problem-oriented design method, which identifies essential design models that are necessary to represent the design artifact of the trading server. Then identified design models are represented in CafeOBJ, which shows expressiveness of the language. CafeOBJ is used as a design validation checker. The design method can also be considered as a problem-oriented guideline for identifying the CafeOBJ modules, which contributes as a module decomposition strategy.

Last, Chapter 7 concludes the dissertation with a summary of the contributions and a list of future works.

1.3 Related Work

This section presents a summary of comparison with related works. A survey on both in FDT and OT is found in Chapter 2, which explains the background of the subject matter in depth.

In the algebraic logic research community, introducing object-oriented concepts to the specification language has been one of the main themes. FOOPS [44] and OOZE [3] are two early efforts to incorporate object-oriented programming concepts in OBJ. They focus on theoretical interest in the integration. Maude [78], based on concurrent rewriting logic, can represent state changes in the algebraic tradition. The Maude style concurrent object model is now a standard for object-oriented algebraic specification. CafeOBJ [27][28][29] has clear semantics based on hidden-order sorted rewriting logic, and provides a specification construction environment CAFE [38]. The work in Chapter 3 adopts the Maude concurrent object model and presents an encoding method in CafeOBJ. A novel contribution is an integration of the Maude object with the idea of the Ambient Calculus [18], which provides a new primitive machinery to manipulate a set of strongly related objects and messages as one unit. And it opens a way to write semantics of distributed system architecture.

Integrating FDT with conventional object-oriented modeling methods is another active area in the decades. Two early works [43][54] propose the modeling guidelines to obtain specification written in the Z notation [110] from the analysis model based on data-driven object-oriented modeling method. Unfortunately, the derived description written in the Z notation is difficult to reason about because behavior analysis tool is not available. ROOA [84] uses LOTOS instead of the Z notation, and the resultant specifications can be validated by execution if one properly uses an executable subset of LOTOS.

In the object-oriented modeling community, however, a scenario-based methodology [20] now sees a wide acceptance. After several proposals such as CRC [9], OOSE [65], RDD [121], and Fusion [23], UML (Unified Modeling Language) [135] pushes a step further to fully use the scenario concept. Scenario-based modeling method is more important than data-driven one in view of a practical application in industry. Since the scenario-based modeling method is a recent proposal, very few work tries to integrate the method with FDT. One important research is Wirsing's work [124] on a formal object-oriented design based on Jacobson's OOSE. He puts emphasis on the role of proof checking in the refinement process, and does not aim to have executable descriptions. On the other hand, a novel contribution of the work in Chapter 4 is a modeling method to obtain executable CafeOBJ specification that is in accordance with scenario-based modeling methods.

Using FDTs in ITU-T standard recommendation documents is an important application area of FDTs. And the ODP trader has been a widely used concrete example because of its importance in distributed computing environments. Several FDTs are used to describe the ODP trader specification [33][34][75]. Fischer et al. [34] and Bowman et al. [16] analyze the characteristics of the ODP standard. One of the important result is summarized as a folklore in [34], which says that no single FDT can specify the richness of the ODP trader. The work in Chapter 5 is a first attempt of the application of the algebraic specification technique to the ODP trader, and it shows that CafeOBJ is expressive enough to describe both the information and computational viewpoints. The work opens a way to using CafeOBJ as a FDT language for the ITU-T standard recommendation document.

Last, the work in Chapter 6 is unique in that none has been published to discuss a whole development life-cycle of object-oriented frameworks with FDTs. The key element of the work, the design aspect approach, is strongly influenced by Jackson [64]. He suggests the importance of the problem-oriented method, but does not propose a specific modeling method in the usual sense. The aspect-centered design is a practice following his philosophy. Another work having much impact on this

work is the parallel iterative development model [120]. The model greatly relaxes constraints posed by the conventional water-fall style development. That various design activities can be done in any meaningful order makes a smooth integration of CafeOBJ with conventional development techniques. In regard to the use of FDT tools, NASA [32] uses PVS [103] as a tool for validating properties of requirement models described in OMT. The present work uses CafeOBJ/CAFE as a checker for design descriptions of the aspect solutions. The design is multiparadigm, not restricted to the object-orientation as in [32].

Chapter 2

Backgrounds

2.1 Introduction

Formal Description Techniques (FDTs) and Object Technology (OT) form two main streams as basis for improving productivity and quality of software systems. It is only recently when the technology gains a wide acceptance as a potential source. FDTs and OT, however, have a long history starting around early 1970's.

The basis of FDT is mathematically-based languages for describing and verifying software systems. Its main concern is to increase quality by following rigorous ways of development; producing descriptions at an early stage of the development by using mathematically rigorous languages, or verifying properties of the descriptions with a proof-oriented method through the development process.

OT started with object-oriented programming (OOP), whose aim is to increase productivity by providing a paradigm that reduces semantic gaps between the world to be modeled and the program text. Later, the paradigm is adapted at early stages of software development, and evolves to object-oriented modeling methods for improving both the productivity and quality of design.

Although the origin of FDT and OT is quite divergent, the two are now converging. For example, one can use the modeling method based on OT to analyze a given problem, and then use FDT for following rigorous development of software system. One can design a new specification language that has OT concepts. Alternatively, one can use FDT to have rigorous semantics for notation and language used in object-oriented modeling methods. One may think of other ways of combination. In any case, FDT and OT are not conflicting, but are complement with each other.

Thus, studying how one uses both technologies in software development process is an important research issue.

This chapter presents a survey of FDT and OT. The emphasis is on the algebraic specification technique and the object-oriented framework.

2.2 FDTs in Software Development

2.2.1 Lightweight use of FDTs

Formal Description Technique (FDT) is a system development method that uses mathematically-based formal specification languages [21]. The specification language has precise syntax and semantics, whose basis is mathematical entity such as set, function, or algebra. Because the language provides notation primitives more abstract than those of programming languages, one can express the target system in an abstract manner. FDT aims at increasing quality of system at early stages of the development.

As for a role of FDT in software system development process, two approaches have been experimented: (1) FDT as a basis for proof-oriented method, and (2) FDT as a language for abstract artifact. The former concerns more about verification, and the latter is satisfied with having precise descriptions.

The proof-oriented method is originated from early days of FDT, and has its basis on a stepwise refinement style of software development. Starting from abstract descriptions of the system, one refines them in a systematic manner to obtain descriptions that can be represented in programming languages. Each refinement step involves transformation of an abstract description into a concrete one. The step may generate proof obligations stating some properties that the transformed description should satisfy. The obligations are discharged during the development process by human designers with possible tool supports (theorem provers). The proof-oriented method was first introduced in 1970's to develop a high quality PL/I compiler at IBM Vienna Laboratory [68]. Later, the technique has been applied to developing safety critical systems. A notable successful project is the Météor automatic train operation system for Paris metro [10].

The proof-oriented method is adequate for those software systems that care less about the development cost. It needs to overcome issues relating to re-education of software engineers to have enough knowledge for performing logical proofs, and re-establishment of software development process to be centered around the proof-

oriented method. Further, because the stepwise approach has its basis on the waterfall style model of development, it is not suitable for software that needs a trial and error style formalization of the system requirement.

The second way is using FDT as a language for abstract artifact. The purpose is to prepare precise *design* documents by using rigorous mathematically-based specification languages. In preparing such documents, one can find many ambiguities or inconsistencies in input information (usually informal documents). Further, the precise document itself is of a great value because it is a basis for future maintenance and extending functionalities. A notable project is using the Z notation for documenting API of CICS online transaction system [55][56]. The idea can be extended to use FDT as a language for standardization recommendation documents such as those developed by ITU-T [34]. The recommendation document should be unambiguous and consistent because many engineers consult the document as a *bible*.

The abstract language approach is more *lightweight* than the proof-oriented use of FDT in that one is satisfied with having abstract descriptions and cares less about verification. However, determining a right abstraction level is pretty hard, which depends on both the characteristics of the application and the objective of constructing the abstract descriptions. A smooth integration of FDT with conventional software development process would be a key issue.

Another observation relates to the fact that most software system suffers from not having a well-defined requirement descriptions. It is because requirement capture is hard and needs a trial and error to reach a final description. As the activity is at an early stage of development, the descriptions are more or less abstract. Further, the descriptions are to be compact for economy of exchanging information and are preferable for mechanical validation. This leads to an idea of using FDT as a ultra high-level language that have application-specific vocabulary to express the artifact in a compact manner.

The development process with FDT should take into account an iterative style of development. One may follow a development model more flexible than the one adopted in the stepwise refinement approach. Parallel-iterative style of development [120] is one such proposal that lowers a barrier of the integration.

In summary, the proof-oriented use of FDT is *heavyweight* and can find a place in the development of safety-critical systems. Use of FDT as a language for abstract artifact is accessible in industry as a new technology to support early stages of software development. The technology is accompanied with some form of an iterative development style.

2.2.2 Algebraic Description Techniques and OBJ

Because the basis of FDTs is mathematically-based language, the characteristics of the language have great impact on the use in the development process. The languages are classified into two broad classes [119]: (1) model-oriented, and (2) property-oriented.

In a model-oriented language, one defines a system functionality *directly* by constructing a *model* of the system in terms of mathematical entities such as sets, relations, tuples, functions, or sequences. Model-oriented languages include VDM, Z, and B. These languages are used to describe functional behavior of operations that constitute software system. One assumes an abstract state of the system, and defines each operation individually by a pair of the pre- and post-conditions. The conditions specify state transformations abstractly. This style of specification is sometimes called state-oriented.

In a property-oriented language, one defines the system functionality *indirectly* by stating a set of properties that the system must satisfy. Property-oriented languages are further broken into two categories, (1) axiomatic and (2) algebraic. In axiomatic languages, the properties are expressed by a set of axioms that have basis, for example, on first-order predicate logic. Iota, Larch and Anna are examples in this category. In algebraic languages, software system is defined to be heterogeneous algebra, and its properties are specified by a set of equational axioms. OBJ, ASL, Act One, and CASL are algebraic languages.

The algebraic specification language has its basis on data abstraction which has first been recognized concretely as the *class* concept of Simula language. And the algebraic specification method provides techniques for the structured specification, validation and analysis of abstract datatypes [39][47][52][123].

The basic idea of the algebraic approach includes description of data structures by giving the names of carrier sets, the names of functions on the sets, and equations constraining the properties of the functions. The first rigorous formalization of abstract datatypes was done within standard many-sorted algebra using initiality in early 1970's. Clear has many of the important ideas including parameterized modules, and may be regarded as a prototypical specification language. The semantics of Clear is given in terms of the categorical notion of institutions. Owing to its modern style of defining semantics, Clear has much influence on later algebraic specification languages.

Among others, the OBJ family can be seen as implementations of Clear for the case of order-sorted equational logic [35][36][37][45][48]. Through several revisions

since late 1970's, OBJ3 [46] developed in early 1990's at SRI is used most widely in the family. OBJ3 is based on the use of initial algebra in conditional equational logic, and thus the descriptions in OBJ3 are executable. It has an efficient order-sorted term rewriting engine, and also includes parameterized modules a la Clear.

The algebraic specification language is suitable for use as a ultra high-level language for some particular application domain. It is because the property-oriented style leads to define application-specific vocabulary as a specification building block (module). And with a carefully prepared library modules, one can express the artifact in a compact manner.

It is interesting to remark two things relating to the algebraic specification techniques and the object technology. First is that Simula [25] is a common root of both algebraic specification languages and object-oriented programming languages in that the *class* concept of Simula is a modern way of describing data abstraction.

Second is a fact that a concept of *signature* is an important contribution of the algebraic specification technology. In the algebraic specification technology, a signature is a collection of both sort and operation declarations, and provides a description of syntactical information that the defined algebra has. The operation declaration consists of an operation symbol (function symbol), an arity (a list of sorts for the arguments) and a value sort (a sort for the return value).

Since a signature plays a significant role to provide syntactical interface information, the idea has also been adopted in the object technology. In CORBA [133], an IDL interface, defining a description of functional objects, is basically the same as a signature of the algebraic technology. Since an IDL interface definition provides all the interface information, one can construct and implement both the client and server programs independently as long as they conform to the IDL interfaces. One can find a similar idea in typed object-oriented programming languages. Java [5] has a concept or a language construct of *interface* to define a type information, which is visible to the outside of objects that conform to the interface. And the terminology *signature* is used in a somewhat different way in relation to the concept of method. A signature of a method consists of the name of the method and the number and type of formal parameters to the method, which is roughly the same as the operation declaration of the algebraic technology.

Both in IDL and Java, the concept or the language construct of *interface* is the most important feature in describing well-defined external interfaces of objects. And the concept is strongly related to a *signature* in algebraic specification technology. With these historical observations, the algebraic technique is expected to be an adequate FDT for object-oriented software systems.

2.3 Object-Oriented Development

2.3.1 Object-Oriented Software Technology

Object software technology can be traced back to late 1960's, and the technology now sees a widespread use in industry with evolution in its various aspects. The aspects can be divided into three: (1) object-oriented paradigm and programming, (2) high-level reuse architecture such as object-oriented frameworks, and (2) object-oriented modeling methods.

Object-oriented paradigm and programming The early days of the object technology owes much to Alan Kay, who proposed a new personal media, the *Dynabook*, that is today seen as a laptop computer [69]. With the *Dynabook*, one could have the power to handle virtually all of its owner's information-related needs. To fulfill the requirement, the software system of the *Dynabook* should implement graphic-based interactive human interface and manipulate various media of data such as animation or audio. With other members of XeroxPARC, Alan Kay then conceptualized a new computing environment for the *Dynabook* in which an object-oriented programming language Smalltalk played an central role.

Although the idea for Smalltalk is influenced by several ancestors, Simula (the concept of class and instance) and Planner (actor as a universal control model) are the most important from a viewpoint of object-oriented programming [118][129]. A version of Smalltalk in 1972 already had the most of the concepts found in today's object-oriented programming [108]. After several revisions [60], the language has matured and been publicly announced as Smalltalk-80 [51] with several articles in BYTE magazine [132] and later the three-volume textbooks. Smalltalk-80, however, suffered execution efficiency problems. Thanks to evolution of both hardware and software technology, Squeak [61], a latest implementation of Smalltalk language, achieves extremely high performance so that Alan Kay's dream in the late 1960's comes true in the late 1990's.

Object-oriented paradigm promotes a way of viewing the *world* as a composite of *objects* [51];

> Everything is object.

Object constitutes the world to be modeled and at the same time is a primitive computational entity in the computer system. Because both are traceable, semantic gap usually found in the design and the program text can be greatly reduced.

Encapsulation, polymorphism, and inheritance are usually considered as three characteristics of object-oriented programming. As early as 1972, Alan Kay has recognized the importance of using intentional definitions of data structures and of passing messages to them [58]. At the center of the idea is *object* as a procedure-embedded datatype. The *object* knows how to implement any given operation on itself. This view of *object* plays a central role in object-oriented programming. And compared with abstract datatype, object can be said to be a functional closure that encapsulates its state and accompanying procedures [24].

Object-oriented frameworks Another interesting observation is that objects are not isolated, but have interaction with each other to show a coherent functionality. This can be best summarized as the following statement [57];

> No object is an island.

Based on this observation, a particular style of system architecture, object-oriented framework, is introduced [26][122].

Object-oriented framework is a promising solution technology for both introducing well-structured architecture to software system and improving reusability of software components. A framework is a reusable design of a program or a part of a program expressed as a set of classes [26][67]. The unit of reuse is not a single class definition, but a set of strongly related classes that constitute the main body of the framework.

IBM adopted the idea of object-oriented framework to call it the *application framework* [79]. It tries to emphasize that any significant object-oriented application takes the form of the framework. In order to lay emphasis on the essence of the framework programming style, IBM uses the slogan *Hollywood Principle* or "Don't call us, we call you," summarizing that the main control flow in the framework *calls* the user-supplied method filling in the presumed calling site (a hot spot). Conventionally, on the other hand, object-oriented programs enjoy library reuse where a user-supplied main module *calls* the reusable library classes.

Apart from the IBM campaign, the object-oriented framework technology is used widely in industry. Today, no significant software system can be constructed in object-oriented programming languages without the framework style of development.[1] Developing well-defined object-oriented frameworks, however, requires design skill

[1]Some experience of the author is found in [80][91][113][114][115][126].

higher than that of the most software engineers. The difficulty comes from two reasons; framework design involves (1) to identify adequate participant objects in the interactions that a framework implement, and (2) to identify *hot spots* [98] that is a portion of program points to be customized. Both requires insightful analysis of the participant collaborations (global flows of control) and of forecasts on possible future customizations. It needs a proper modeling method to help develop object-oriented frameworks.

Object-oriented modeling methods In early 1980's, Booch [14] has invented a terminology, "Object-Oriented Development," as a new software development method for systems written in Ada. The method includes a way of decomposing a system based on the concept of object. The motivation is that traditional functional approaches are not adequate for building object-based programs in Ada. As discussed above, object is helpful in reducing the semantic gap and making the design and the program text traceable. Thus, the concept of object is also expected to play an important role in early stages of software development process. This observation leads to elaborating nearly thirty object-oriented analysis and design methods in the next several years [59].

Object-oriented method generally has two roles; (1) it provides basic notations to represent various aspects of *objects*, and (2) it provides further notations and guidelines to help derive object definitions. Different methods assume different guidelines but fall into two broad categories [105]: data-driven methodology and responsibility-driven methodology. The latter is sometimes called scenario-based methodology [20].

In data-driven methodology such as OMT [102], structural relationships between objects are the main concerns at an early stage of development. Most functional aspects of objects are left until later. The data-driven methods focus on structural aspects of a system of objects and emphasize the object relationships such as class inheritance, aggregation, or association links.

In scenario-based methodology, such as that of CRC [9], OOSE [65], and RDD [121], an object is an entity that is responsible for a particular functionality that the system provides. Identifying objects requires an analysis of functional behavior of the system as a whole, and involves decomposition of functional coupling between objects.

Some design methods include modeling concepts and notations for development of object-oriented frameworks. Because collective behavior of objects is important,

scenario-driven methodology is more adequate than data-driven one. The methods include collaboration-based design [9][20][92], role-based design [101], and design pattern[41][42][98].

OOSE [65] and RDD [121] are two pioneers to propose scenario-based object-oriented methodology. In OOSE, a scenario is called *use case* to put emphasis on the use in analysis or business process modeling. RDD is more on the design stage, and concerns with responsibility of objects participating in the scenarios. Responsibility can be seen as an abstract external interface specification.

Fusion [23] is one of the second-generation object-oriented methodologies that is a result of critical analysis of OMT, the Booch Method, RDD, and others. The analysis stage is based on OMT and the use case concept of OOSE with the addition of the pre- and post-condition style of specifications for capturing the desired behavior of the system. The design stage adopts CRC and RDD. Fusion, throughout both the analysis and design stages, emphasizes the scenario concept by borrowing ideas from OOSE or RDD/CRC.

Booch and Rumbaugh's unified method supports the scenario concept [15]. UML (Unified Modeling Language) [135], which started from the unified method, pushes a step further to fully use the scenario concept with interaction diagrams (sequence diagram and collaboration diagrams).

Design pattern [41][42] is a catalog of useful design *idioms* mined mostly from object-oriented graphic user-interface programs written in C++. One of the original application of the design pattern is to document object-oriented frameworks [67]. However, since the design pattern is found to be of a great significance on its own right, the idea of the pattern is now widely used in industry. The idea of accumulating useful patterns can be traced back to C. Alexander on his pattern language in urban and building designs [4] and the Programmer's Apprentice Project at MIT [100]. The MIT project focused on collecting program-level idioms (idioms for writing Lisp expressions or Ada statements) as programming knowledge. Their interest was more on representing such knowledge and reasoning mechanism on it. The work did not see a wide acceptance in the software engineering community.

Catalysis [30] can be considered as the two and half or the third generation because it outgrows Fusion in that Catalysis is more on the object interactions and the formal description techniques. Catalysis uses the UML design notations, and makes full use of OCL (Object Constraint Language) to express the pre- and post-conditions or other variations of behavior description in a textual form. Catalysis also provides process patterns for object-oriented software development. A core concept found in the process is closely related to the idea of *component* that is

a reusable unit and includes the concept of object-oriented framework or design pattern.

In summary, object-oriented framework is a promising solution technology. It elucidates that any significant application system consists of more than one objects and that thus their collective behavior needs to be analyzed. The technology is already in use for developing non-trivial software systems in industry, which motivates many methodologists to devise modeling methods focusing on the nature of framework. Unfortunately, designing well-organized object-oriented framework is still an art, and is short of formality to become a stable technology.

2.3.2 Convergence of FDT and OT

According to the three aspects of current practice in object-oriented software technology, the convergence of FDT and OT also takes three forms. All the three share a common goal to establish a new development method for object-oriented software with formal description techniques.

The first category of research work is centered around to define object-oriented extensions of existing formal specification languages [19][72][73][88][111]. The purpose is to provide rigorous design notations for object-oriented design descriptions. Most of the works, however, has concentrated on incorporating basic object-oriented concepts such as state encapsulation, property inheritance, and polymorphism into the respective host specification language. These investigations do not extensively address issues relating to either object-oriented frameworks or object-oriented modeling method. The issues on modeling collective behavior is out of scope. Since the algebraic specification technique is, in a naive view, the most suitable potential candidate for incorporating the object-oriented concept, the following is mostly concentrated on the algebraic logic languages.

Larch [53] is a two-tiered specification language, in which the algebraic specification provides common vocabulary. An interface language component uses the vocabulary to describe behavioral aspects of functions or procedures. Larch/C++ [74] is one of the Larch family languages. It is primarily intended to be used for writing the interface specifications of C++ member functions in the state-oriented style.

FOOPS [44] extends OBJ to incorporate object-oriented programming concepts. It also makes explicit the distinction between a class and a module. The former serves as a template for object definition, while the latter is used for organizing specification in a modular manner. OOZE [3] is a specification construction and

analysis environment for FOOPS, and it provides a specification animator based on OBJ3 [46].

Maude [78], started as an extension to OBJ3, provides an elegant construct for object-orientation based on concurrent rewriting logic. Since concurrent rewriting logic is suitable for modeling *changes*, either stateful objects or reactiveness can be modeled precisely in the algebraic methods. Maude now extends the concurrent rewriting logic to be *reflective* and aims to be a host for various symbolic processing systems that have clear logical semantics [22].

CafeOBJ [27][28][29][38] is also a descendant from OBJ3, and has clear semantics based on hidden-order sorted rewriting logic. The logic subsumes order-sorted equational logic, a subset of concurrent rewriting logic, and hidden algebra [49][50]. Since CafeOBJ has clear operational semantics, specifications written in it can be executable.

The second category of research focuses on methodology issues. [43][54] propose an integration of the Z notation [110] with object-oriented design methodology. Unfortunately, the derived description written in the Z notation is difficult to be reasoned about because behavior analysis tool is not available.

A recent activity of the precise UML [136] also uses the Z notation and similar mathematical tools for formalization of UML notations [71][135]. Since UML is a family of notations, the precise UML activity have various areas of research including (1) precise semantics of diagram notations such as Class Diagram, (2) stepwise refinement rules on the diagrams, (3) precise semantics of OCL (Object Constraint Language).

NASA has conducted several case studies on the lightweight use of formal methods in the requirement modeling [32], which includes an integration of OMT [102] and PVS [103]. The OMT diagram descriptions are manually translated into specification fragments that can be fed into PVS. Then, properties that the OMT description must holds are reasoned about by using PVS. In a sense, PVS is used as a design validation checker.

ROOA [84] proposes a methodology for object-oriented analysis and uses LOTOS for its underlying rigorous notation. Object is modeled as a LOTOS process. Since LOTOS has executable semantics, ROOA specification can be validated by execution.

Wirsing [124] proposes a formal object-oriented design based on Jacobson's OOSE [65] and in which a Maude-based formal object model is encoded in an algebraic specification language Spectrum. He also discusses the role of proof checking in the refinement process.

The third category concerns about high-level object-oriented architecture such as object-oriented framework or design pattern that aims to increase reusability. The work is one such that chooses a particular style of reuse approach and formalizes its rigorous representation to reason about some of important properties of the specificands. A key idea in this category is to elucidate the importance of global structural aspects of the system and of abstract views of control flow between the structural components. Thus, the work is somewhat based on the notion of Software Architecture and ADL (Architecture Description Language) [7][77][106].

Inverardi and Wolf [62][63] propose an ADL that is based on CHAM (Chemical Abstract Machine). The reactive features relating to the architecture artifact can be reasoned about by using the reduction rules of CHAM [13]. Mikkonen [81] uses DisCo, a variant of UNITY, for formal descriptions of some of design patterns in [42], and devises a method of composing the patterns. Sousa [109] uses Wright specification language [106] to describe the component and container event protocols. Wright language in turn has its basis on CSP and thus employs a model-checking algorithm to verify properties.

As discussed in Section 2.3.1, *object* is not just a data abstraction, but is a functional closure that encapsulates the internal states and accompanying procedures. At early stages of software development, *object* is best understood as an entity with a variety of computational facets. Further, object-oriented frameworks and scenario-based methods promote a view of modeling collective behavior of objects. Unfortunately, none of the work is reported to cover all the aspects of object just mentioned.

This dissertation uses an algebraic logic language CafeOBJ that already has some of object-oriented concepts, and discusses how one integrates CafeOBJ in a process using conventional scenario-based object-oriented modeling method to develop object-oriented frameworks. Thus, it covers all the aspects of the object-oriented software development technology augmented with the algebraic specification technique.

Chapter 3

Object-Oriented Algebraic Specification in CafeOBJ

3.1 Introduction

This chapter introduces what object-oriented algebraic specification is and how the description is written in an algebraic logic language CafeOBJ.

Object in object-oriented programming is a procedural abstraction of a data type, and it can be interpreted as a function closure [24]. A classic view of object as abstract datatype is no longer adequate. The key idea is to model the functional behavior of a data type in terms of state changes, but not by using a set of static relationships. Thanks to the underlying concurrent rewriting machinery, CafeOBJ can represent state changes in the algebraic logic framework. Finding a right computational entities in the object-oriented algebraic specification and its encoding method is the primary concern.

Section 3.2 gives a brief introduction of CafeOBJ. The explanation is mainly for specifiers, who are *users* of CafeOBJ.[1] And it emphasizes syntactic and pragmatic issues with illustrative examples. Section 3.3 presents an encoding method of Maude concurrent object model in CafeOBJ. Section 3.4 proposes an idea of integrating the Maude model and the Ambient calculus, in which an ambient is an entity to enclose more than one closely related concurrent objects. These form a basis of the object-oriented algebraic specification written in CafeOBJ.

[1]Those, who are concerned with the theoretical aspects of CafeOBJ, may find the book [104] a useful starting point.

3.2 Informal Introduction for Specifiers

CafeOBJ has two kinds of axioms[2] to describe functional behavior [40][104]. An equational axiom (*eq*) is based on equational logic and thus is suitable for representing static relationships. A rewriting axiom (*trans*) is based on a subset of concurrent rewriting logic and is suitable for modeling state changes.

3.2.1 Abstract Datatype

Here is a simple example, a CafeOBJ specification of `LIST`. The module `LIST` defines a generic abstract datatype `List`. `_ _` (juxtaposing two data of the specified sorts) is a `List` constructor. Two accessor or observer functions `hd` and `tl` are the standard ones. `|_|` returns the length of the operand list data and is a recursive function over the structure of the list.

```
mod! LIST[X :: TRIV] {
  [ NeList, List ]   [ Elt < NeList < List ]
  protecting (NAT)
  signature {
    op nil : -> List
    op __ : List List -> List {assoc id: nil}
    op __ : NeList List -> NeList
    op __ : NeList NeList -> NeList
    op hd : NeList -> Elt
    op tl : NeList -> List
    op |_| : List -> Nat
  }
  axioms {
    var X : Elt     var L : List

    eq hd (X L) = X .
    eq tl (X L) = L .

    eq | nil | = 0 .
    eq | X   | = 1 .
    eq | X L | = 1 + | L | .
  }
```

[2]Hidden algebra is not considered.

```
}
```

The module N-LIST imports the module LIST and adds definitions of some utility functions such as n-hd and n-tl. The function n-hd returns the specified number (N) of elements from the head of the list, and n-tl discards N elements.

```
mod! N-LIST[X :: TRIV] {
  protecting (LIST[X])
  signature {
    op n-hd : Nat NeList -> List
    op n-tl : Nat NeList -> List
    op rev  : List -> List
    op nhd-aux : Nat NeList NeList -> NeList
    op rev-aux  : List List -> List
  }
  axioms {
    var N : Nat     vars L L'  : List   var X : Elt

    eq n-hd (N, L) = nhd-aux (N, L, nil) .
    ceq nhd-aux (N, L, L') = rev(L')  if N == 0 .
    ceq nhd-aux (N, (X L), L') = nhd-aux ((N - 1), L, (X L')) if N > 0 .

    ceq n-tl (N, L) = L  if N == 0 .
    ceq n-tl (N, (X L)) = n-tl ((N - 1), L) if N > 0 .

    eq  rev L = rev-aux(L,nil) .
    eq rev-aux(nil,L') = L' .
    eq rev-aux((X L), L') = rev-aux(L, (X L')) .
  }
}
```

The above examples also show a typical use of modules in a structured way. (1) The module LIST defines a basic data structure (List) by providing constructors and observers. (2) Another module N-LIST introduces further utility functions with importing the LIST module. Such utility modules are expected to constitute a reusable library.

3.2.2 Concurrent Object

The Maude concurrent object [78] can easily be encoded in CafeOBJ. The Maude model relies on `Configuration` data and rewriting rules based on concurrent rewriting logic. `Configuration` is a snapshot of global states consisting of objects and messages at some particular time. Object computation (sending messages to objects) proceeds as rewriting on `Configuration`. In addition, Maude has a concise syntax to represent the object term (`<(_:_)|_>`) and some encoding techniques to simulate *inheritance*. The Maude model can be considered as a standard encoding for concurrent objects in algebraic specification languages [92][125].

Below is an example of object definition. The module `ITERATOR` defines an Iterator object, which maintains a list of data and returns the specified number of data when requested by a `next-n` message.[3]

```
mod! ITERATOR[X :: TH-ITERATOR-AID, Y :: TH-ITERATOR-MSG] {
  extending (ROOT)
  protecting (ITERATOR-VALUE)
  [ IteratorTerm < ObjectTerm ]
  [ CIdIterator < CId ]
  signature {
    op <(_:_)|_> : OId CIdIterator Attributes -> IteratorTerm
    op Iterator : -> CIdIterator
  }
  axioms {
    vars O R : OId   var L : List   var N : NzNat
    var REST : Attributes

  ctrans next-n (O,N,R) <(O : Iterator)|(body = L), (REST)>
  =>  <(O : Iterator)|(body = n-tl(N,L)), (REST)>
      return(R,true) outArgs(R,n-hd(N,L))     if N <= |L| .

  ctrans next-n (O,N,R) <(O : Iterator)|(body = L), (REST)>
  => <(O : Iterator)|(body = L), (REST)> return(R,false)  if N > |L| .

  trans destroy(O,R)  <(O : Iterator)|(REST)> => void(R) .
  }
}
```

[3]It is a CafeOBJ encoding of the IDL iterator interface with functional behavior at an abstract level (Section 5.5).

The ITERATOR is a parameterized module. Both TH-ITERATOR-AID and TH-ITERATOR-MSG provide specification of the parameter module. The former introduces the attribute name that an Iterator object has, and the latter defines all the messages that the object can respond to.

```
mod* TH-ITERATOR-AID {
  extending (AID)
  signature {  op body : -> AId   }
}

mod* TH-ITERATOR-MSG {
  extending (MESSAGE)
  signature {
    op next-n : OId NzNat OId -> Message
    op destroy : OId OId -> Message
   }
}
```

The module ITERATOR imports two other modules ROOT and ITERATOR-VALUE. The module ROOT is a runtime module that provides the symbols necessary to represent Maude concurrent objects. That is, it provides the following sort symbols: Configuration to represent the snapshot, Message for messages, ObjectTerm for the body of objects which consists of Attributes (a collection of attribute name and value pairs), CId for class identifiers, and OId for identifiers of object instances.

As shown in the above example, a user-defined class should define a concrete representation of the object term (<(_:_)|_>) in a new sort (IteratorTerm) and a class identifier constant (Iterator) in another new sort (CIdIterator). The axioms part has a set of rewriting rules (either trans or ctrans), each of which defines a method body. In writing the method body, one often refers to symbols defined in other modules such as, for example, the sort List and the related utility functions. The module ITERATOR-VALUE is supposed to import all the modules such as N-LIST[NAT] necessary for the ITERATOR.

3.3 Encoding Concurrent Object in CafeOBJ

This section explains an encoding method of the Maude concurrent object in CafeOBJ. It first presents the original Maude concurrent object model [78] and then shows the CafeOBJ modules to encode the model.

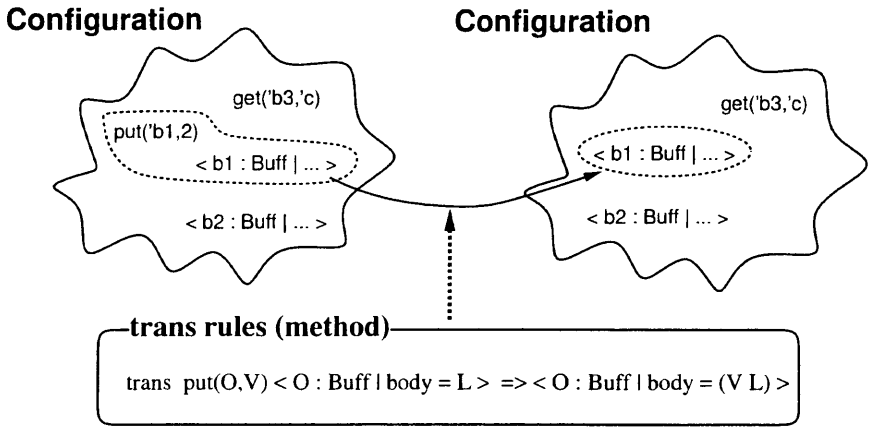


Figure 3.1: Snapshot of Concurrent Objects

3.3.1 Maude Concurrent Object Model

Maude concurrent object model is a milestone to provide logical framework for representing state changes and concurrency in the algebraic technology. The core idea is a new logic called *concurrent rewriting logic.*

Figure 3.1 illustrates the basic idea of Maude concurrent object system that elucidates three important things: (1) Configuration, (2) `trans` rule (transition rule), and (3) object term. The model is similar to the transition semantics of the actor model [1], which describes the semantics in terms of how a system of actors change in the course of message events occurred in the system.

The configuration is a multiset of objects and messages, and maintains execution snapshots of the concurrent object system. The `trans` rule rewrites a portion of the configuration at the left to become the one at the right. Since the `trans` rule in Figure 3.1 acts on a pair of specified message and object, the rule can be regarded as a method of the object. An object in the configuration has its own concrete representation as

$$< O : C|attr >$$

where O refers to object identifier, C is an identifier of class that the object belongs to, and attr is a set of attributes comprising the internal state of the object.

The example shows that a message (`put('b1,2)`) and an object (`< b1 : Buff | ... >`) are matched with the left-hand side of the `trans` rule, and

$$
\begin{aligned}
& M_1 \ldots M_n < O_1 : C_1 | \text{attr}_1 > \ldots < O_m : C_m | \text{attr}_m > \\
\longrightarrow \quad & < O_{i1} : C'_{i1} | \text{attr}'_{i1} > \ldots < O_{ik} : C'_{ik} | \text{attr}'_{ik} > \\
& < Q_1 : D_1 | \text{attr}''_1 > \ldots < Q_p : D_p | \text{attr}''_p > \\
& M'_1 \ldots M'_q \qquad\qquad \text{if } C
\end{aligned}
$$

Figure 3.2: General Form of Transition Rule

that the rule rewrites the two terms into a new object term (`< b1 : Buff | ... >`) with the updated attribute values as specified in the right-hand side of the rule. The rewriting process continues until no more message and object pair is found in the configuration.

While the example in Figure 3.1 is simple and easy to understand as an execution mechanism of object, the Maude concurrent object model is more general and expressive. Figure 3.2 shows a general form of the transition rule in [78]. A few remarks are in order.

- The messages $M_1 \ldots M_n$ disappear.
- The state and possibly even the class of the objects $O_{i1} \ldots O_{ik}$ may change.
- All the other objects O_j vanish.
- New objects $Q_1 \ldots Q_p$ are created.
- New messages $M'_1 \ldots M'_q$ are sent.

The rewriting is fired only when the condition C is satisfied. Further, the rewriting process involves an exhaustive search for finding terms (objects and messages) in the configuration that match any of the rules. Matching modulo ACI (Associative Commutative Idempotence) on the configuration accomplishes the exhaustiveness in the search process.

The Maude model is quite general and expressive, and thus is not easy for specifiers to write object-oriented algebraic specifications. Figure 3.3 illustrates four types of restricted transition rules.

(a) Classical Object

$$M < O : C_0|attr > \quad \longrightarrow \quad < O : C_0|attr' > \quad M'_1 \ldots M'_q \quad \text{if} \quad C$$

(a) Classical Object.

$$M_1 < O : C_1|attr_1 > \quad \longrightarrow \quad < O : C_1|attr'_1 > \quad M'_1 \ldots M'_q \quad \text{if} \quad C$$
$$\ldots$$
$$M_m < O : C_m|attr_m > \quad \longrightarrow \quad < O : C_m|attr'_m > \quad M'_1 \ldots M'_q \quad \text{if} \quad C$$

(b) An Object with Multiple Bodies.

$$M < O_1 : C_1|attr_1 > \ldots < O_m : C_m|attr_m >$$
$$\longrightarrow \quad < O_1 : C_1|attr'_1 > \ldots < O_m : C_m|attr'_m > M'_1 \ldots M'_q \qquad \text{if} \quad C$$

(c) A Multiparty Interaction.

$$M_1 \ldots M_n < O : C_0|attr > \quad \longrightarrow \quad < O : C_0|attr' > \quad M'_1 \ldots M'_q \quad \text{if} \quad C$$

(d) Merging Multiple Messages.

Figure 3.3: Restricted Transition Rules

An object O is a receiver of the message M, and changes its own states to be attr' while sending the messages $M'_1 \ldots M'_q$ to objects. They may include the original receiver itself.

(b) An Object with Multiple Bodies

An object O consists of multiple object terms, each of which is described by a different class C_i. The object O is supposed to be instantiated from multiple classes C_i.

(c) A Multiparty Interaction

More than one objects $O_1 \ldots O_m$ are involved in a message event M. It implies that all the objects synchronously participate in the interaction.

(d) Merging Multiple Messages

More than one messages $M_1 \ldots M_n$ are involved in one interaction to achieve synchronization of messages.

Although other rules are possible in principle, the four rules are typical examples to express a set of interacting concurrent objects. The restricted rules act as a guideline or a set of templates for use by specifiers.

3.3.2 CafeOBJ Descriptions

The Maude concurrent object model provides a basis for encoding *objects* in the algebraic specification technique. Meseguer discusses an encoding method in terms of Maude syntax [78]. The basic idea is to use rewriting logic deduction modulo ACI, and to use other advanced language features of Maude such as sort constraints. Further, Maude supports special syntax for defining object modules (`omod` $\cdots$ `endom`). The object module is just a syntax sugar and is translated into a set of Maude basic modules (`mod` $\cdots$ `endm`), whose semantics are given by concurrent rewriting logic. Because the Maude basic module and CafeOBJ differ slightly, encoding the Maude concurrent object model in CafeOBJ requires further considerations. The CafeOBJ encoding below is almost a reformulation of the presentation in [78].

The CafeOBJ encoding method consists of two components: (1) a guideline of writing user-defined class in CafeOBJ, and (2) a set of *runtime* modules for the user-defined class modules. The runtime modules are referred to as the Object System Kernel. More concretely, a CafeOBJ module defining a new user class must import

the Object System Kernel. The module ITERATOR in Section 3.2.2 is an example to define Class Iterator. The discussion below concentrates on the Object System Kernel.

The module ROOT is a main module that every user-defined class module must import. ROOT provides all the *symbols* necessary for defining class. First, as in Maude, term representation of object instance takes a form of <(_:_)|_> that is a value of sort ObjectTerm. The term <(_:_)|_> has three arguments, (1) object identifier (OId), (2) class identifier (CId), and (3) a set of attributes (Attributes) that define internal status of the object instance.

```
mod! ROOT {
  extending  (CONFIGURATION)
  extending  (MESSAGE)
  protecting (CID)
  protecting (OID)
  protecting (ATTRIBUTES)

  [ ObjectTerm ]
  [ ObjectTerm < Configuration,  Message < Configuration ]

  signature {
    op <(_:_)|_> : OId CId Attributes -> ObjectTerm
    op return : OId AttrValue -> Message
    op void : OId  -> Message
  }
}
```

The object attributes (Attributes) are a set of a pair of attribute name (AId) and value (AttrValue). The module ATTRIBUTES provides the basic definitions.

```
mod! ATTRIBUTES {
  [ Attributes, Attribute ]
  protecting (AID)
  protecting (ATTR-VALUE)
  [ Attribute < Attributes ]

  signature {
    op null  : -> Attributes
    op (_=_) : AId AttrValue -> Attribute
```

```
    op (_,_) : Attributes Attributes -> Attributes {assoc comm id: null}
  }
}
```

The sort `AttrValue` corresponds to a set including all the values that can be attribute values of the object instances. This means that any sort introducing attribute values should be a subsort of `AttrValue`. The module `BASIC-VALUE` illustrates that object identifies (`OId`), natural numbers (`Nat`), and boolean values (`Bool`) can be attribute values.

```
mod! ATTR-VALUE {
  [ AttrValue ]
}

mod! BASIC-VALUE {
  extending (ATTR-VALUE)
  protecting (OID)
  protecting (NAT)
  [ OId < AttrValue,  Nat < AttrValue,  Bool < AttrValue ]
}
```

The module `CONFIGURATION` is another main module that introduces concurrency. An execution snapshot of concurrent objects is maintained by a configuration (`Configuration`). A constructor (`__`) is defined to be `{assoc comm id: null}` indicating that rewriting on a `Configuration` data is ACI. Further, as shown in the module `ROOT`, both `ObjectTerm` and `Message` are subsorts of `Configuration` in order that messages and objects can be members of the snapshot.

```
mod! CONFIGURATION {
  [  Configuration ]

  signature {
    op null : -> Configuration
    op (__) : Configuration Configuration -> Configuration
                                          {assoc comm id: null}
  }
}

mod! MESSAGE {
```

```
  [ Message ]
  [ DebugMessage Exception ]
  [ DebugMessage < Message ]
  [ Exception < Message ]
}
```

The module `MESSAGE` introduces two further subsorts (`DebugMessage` and `Exception`). Roles of both are obvious.

3.4 Encoding Ambient in CafeOBJ

The Maude concurrent object model allows us to describe system specifications in terms of a set of interacting concurrent objects. In complex software systems, however, some objects are closely related and others are not. Thus, enclosing more than one closely related concurrent objects would be a valuable machinery to describe complex systems. Such machinery may be used to express architectural organization of the system. It is because the machinery can be used to decompose the whole system into several distinguishable components.

This section proposes to integrate the calculus of mobile ambients [18] with the Maude concurrent object [78]. The presentation will not rigorously follow the calculus nor reformulate it, rather will borrow from the calculus the concept of ambient to encapsulate a set of concurrent objects and messages and the three mobile capabilities as the primitives. The encoding shows that making `Configuration` first-class introduces flexibility to the basic Maude concurrent object model in an interesting way [93].

3.4.1 The Mobile Ambient

The calculus of Mobile Ambients [18] follows an idea of asynchronous π-calculus [83] and provides *ambients* to encapsulate a *process* in order to express that a certain process exists in a certain ambient. The ambient is the unit of mobility; that is, an ambient together with the enclosed process does move. An ambient can enter other ambients, go out of other ambients, and open others to do interaction at the enclosing process level. Figure 3.4 shows the syntax of the base calculus and Figure 3.5 depicts some of the reduction rules related to the three mobile capabilities,

n		names
P, Q	::=	processes
	(ν n)P	restriction
	0	inactivity
	P | Q	composition
	!P	replication
	n[P]	ambient
	M.P	action
M	::=	capabilities
	in n	can enter n
	out n	can exit n
	open n	can open n

Figure 3.4: Syntax of Ambient Calculus

n[in m.P | Q] | m[R]	$\rightarrow$	m[n[P | Q] | R]
m[n[out m.P | Q] | R]	$\rightarrow$	n[P | Q] | m[R]
open n.P | n[Q]	$\rightarrow$	P | Q

Figure 3.5: Reduction Rules (a part)

in, *out*, and *open*. The calculus also specifies further reduction rules and a set of structural congruence relationships.[4]

The first rule in Figure 3.5 gives operational semantics for the `in` primitive. A process `P` preceded by a capability `(in m)` in an ambient `n[...]` can interact with an adjacent ambient `m[...]`. The result is that the ambient `n[...]` enters `m[...]`. In a similar manner, `(out m).P` can go out of another ambient `m[...]`. Last, `open n.P` with `n[...]` proceeds to interact at the enclosing process level while the `n[...]` ambient disappears.

3.4.2 CafeOBJ Descriptions

[4] [18] also shows that the base calculus is Turing Machine equivalent.

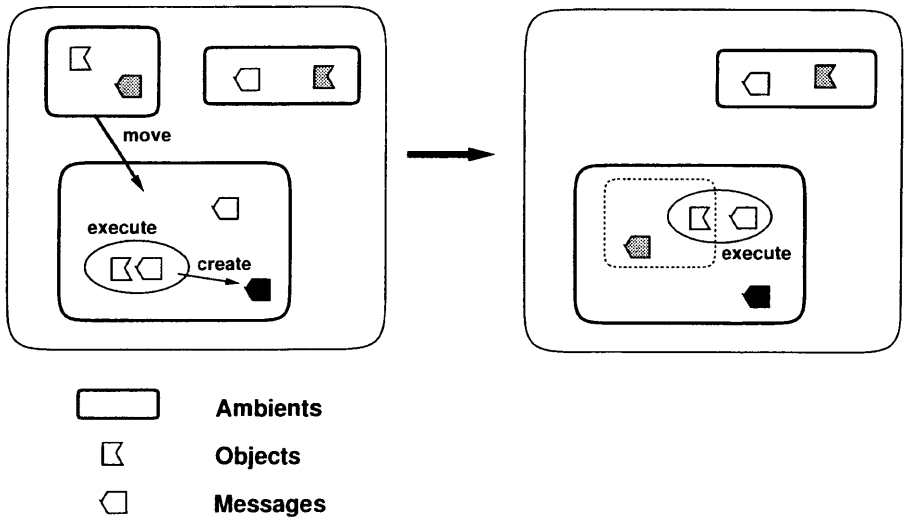


Figure 3.6: Ambients and Objects

Figure 3.6 illustrates a basic idea of the specification technique for enclosing closely related objects. The key point is the introduction of *ambients* that encapsulate *objects* and *messages*. An ambient that packs objects and messages is the entity to move around. The Figure 3.6 shows a situation where three ambients are in the system.[5] Here, an object executes a message to create a new message in the bottom ambient, while the upper left ambient tries to enter the bottom one. These two scenarios are independent, which implies both can happen concurrently. The upper left ambient moves its location while enclosing an object and a message in it. After the upper left ambient successfully goes into the bottom one, it disappears in order to cause object and message interactions. Thus, the coming object and the resident message are matched in execution as shown in the right figure.

Figure 3.7 shows the *kernel* of the mobile objects. The module `ROOT` provides functionalities for simulating the Maude concurrent object (Section 3.3). The cor-

[5]An object and a message with the same color are supposed to be matched in execution.

```
mod! MOBILE-OBJECTS {
  extending (ROOT)
  [ Capability Handle ]
  signature {
    op _@_  : Capability Configuration -> Configuration
    op _[_] : Handle Configuration -> Configuration
    op open : Handle -> Capability
    op in   : Handle -> Capability
    op out  : Handle -> Capability
  }
  axioms {
    vars P Q R : Configuration
    vars N M : Handle

    trans ((N)[ (in(M)@(P)) Q ]) ((M)[ R ]) => (M)[ ((N)[ P Q ]) R ] .
    trans (M)[ ((N)[ (out(M)@(P)) Q ]) R ]  => ((N)[ P Q ]) ((M)[ R ]) .
    trans (open(N)@(P)) (N)[ Q ]            => P Q .
  }
}
```

Figure 3.7: Mobile Objects

respondence with the Mobile Ambient is as follows;

```
Process  →  Configuration
Name     →  Handle
```

In other words, an ambient encapsulates a `Configuration` data consisting of objects and messages and introduces a boundary to the top level `Configuration` that defines the whole system. The ambient is a device to make `Configuration` a first-class data that can be manipulated from objects. With the ambient, one can write specifications that have a hierarchy of `Configurations`. It provides flexibility of handling `Configuration` that is necessary to represent *mobility* of objects and/or messages. Last, the three rules in the module `MOBILE-OBJECTS` are essentially the same as those in Figure 3.5. This shows that the encoding of ambient in CafeOBJ is truly straightforward.

3.4.3 An Example : Shared Resources

When more than one concurrent objects constitute a system, some control structure should be introduced to manipulate shared resources. Some specification fragment should be responsible for *locking*. Two of the restricted transition rules in Figure 3.3, (c) and (d) can express synchronization between objects or messages. Synchronization at the level of objects or messages alone, however, is not easy to describe necessary mechanism in a compelx system. The following discusses how to express synchronization on shared resources by using the idea of ambient.

The encoding of concurrent objects in CafeOBJ relies on the concurrent rewriting of `Configuration`. Since atomicity is ensured solely for binding an object and a message, one has to introduce some form of *transaction* consisting of a series of message executions. The situation is similar to the case of the actor model [1]. Since the actor model is intrinsically concurrent, a set of high level actors are needed to provide control abstractions such as locking or synchronous invocation [2].

From the specifier's viewpoint, the separation of concerns is a key issue. That is, a specification fragment for locking is explicitly separated from the fragments for the application logic part (concurrent objects). The idea of ambients provides a concise way to implement high level control abstractions.

The following fragment provides a template (or a pattern) to encode a locking abstraction. The identifiers `request`, `return`, `repeat`, `do`, and `complete` are instances of sort `Message` and `server` refers to an `ObjectTerm` data. Several auxiliary

functions, whose values are ambients, are also introduced.

```
trans request server => repeat do server .
trans do server      => complete server .
trans repeat complete server
=>    return-message(M') dup-secretary(M,W) open(M)@( server ) .

eq message(M)          = (M)[ open(M)@( request ) ] .
eq secretary(M,W)      = (M)[ in(M)@(open(W)@(null)) ] .
eq shared-object(W,M)  = (W)[ open(M)@( server ) ] .
eq return-message(M)   = (M)[ out(W)@(open(M)@( return )) ] .
eq dup-secretary(M,W)  = (M)[ out(W)@(in(M)@(open(W)@(null))) ] .
```

The idea is that the ambient `secretary` accepts an in-coming message `message` and hands it to `shared-object` while `secretary` itself disappears until `server` in the ambient `shared-object` processes the contents of the in-coming message. After the completion, `secretary` reappears to accept a possible further `message`. While `server` is doing some computations, there is no `secretary` available, which means that any request for `server` is blocked. The trace of reduction steps just mentioned is outlined below.

```
message(M) secretary(M,W) shared-object(W,M)
=> (M)[ open(M)@( request ) ] (M)[ in(M)@(open(W)@(null)) ]
   (W)[ open(M)@( server ) ]
=> (M)[ open(M)@( request ) (M)[ open(W)@(null) ] ]
   (W)[ open(M)@( server ) ]
=> (M)[ ( request ) open(W)@(null) ] (W)[ open(M)@( server ) ]
=> (W)[ (M)[ ( request ) ] open(M)@( server ) ]
=> (W)[ request server ]
=> (W)[ return-message(N) dup-secretary(M,W) open(M)@( server ) ] .
=> (W)[ (N)[ out(W)@(open(N)@( return )) ]
        (M)[ out(W)@(in(M)@(open(W)@(null))) ]
        open(M)@( server ) ]
=> (N)[ open(N)@( return ) ] (W)[ open(M)@( server ) ]
   (M)[ in(M)@(open(W)@(null)) ]
```

Note that the trace also indicates how an object (`server`) in an ambient (`W`) sends a message (`return`) to other objects in a neighbour ambient (`N`) by using the auxiliary function `return-message`.

Last, the term in the last line will be readable if we rewrite back the term to the symbols of the auxiliary functions above. However, it needs a manual control over the rewriting process in the CAFE environment since the the above term is a normal form in the current setting. The result of manual rewriting is shown below.

```
= message(N) shared-object(W,M) secretary(M,W)
```

3.5 Discussions

This chapter presented how one encoded the Maude concurrent object model in CafeOBJ and introduced a new machinery (the ambient) in order to manage more than one closely related objects as a basic unit. The encoding method of the Maude concurrent object model essentially follows the presentation in [78]. It, however, includes some consideration from specifier's viewpoints. Although the idea of the ambient is borrowed from [18], integration with the Maude concurrent object model is new. The contribution of this chapter is to show that CafeOBJ can encode the important primitive machineries for describing object-oriented specifications.

A summary on the method to encode Maude in CafeOBJ follows. First, the functionality of the Object System Kernel is divided into eight modules. The Kernel provides abstract runtime interpreter for the Maude concurrent object system. And a proper division of labour beween the eight modules can contribute much to customize the interpreter behavior easily. Actually, introducing `Exception` is necessary when one needs to write objects implementing IDL interfaces. This was done by adding a new subsort `Exception` in the module `MESSAGE` only.

Second, the encoding method makes use of the parameterized module of CafeOBJ. Specifically, a module that defines the body of the class has two parameters. One for module providing all the attributes names, and the other having all the message terms that the method of the class concerns with.

Third, because CafeOBJ does not provide syntactic macro, the CafeOBJ class definition can offer no special syntax for defining class modules. One has to write the *translated* bare module directly. Thus, the encoding method needs a clear guideline of writing user-defined classes in CafeOBJ, which, however, is somewhat a drawback of this work.

Integrating the ambient with the Maude concurrent object model opens a way to write semantics of the distributed system architecture. The ambient encapsulates a `Configuration` data consisting of objects and messages, and introduces a

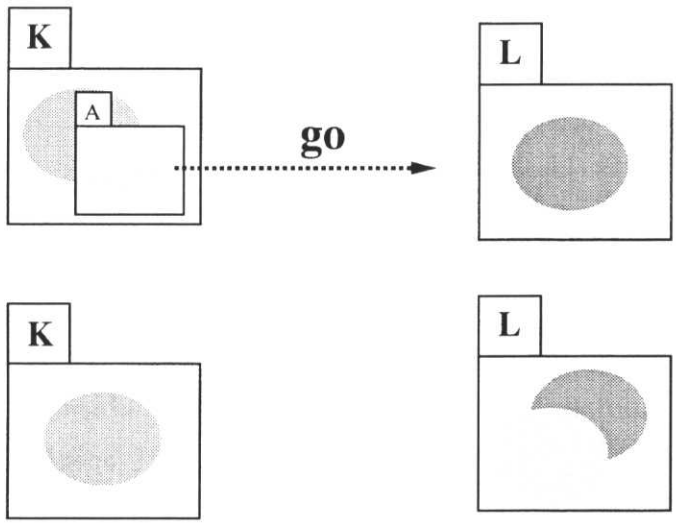


Figure 3.8: Three Party Interaction

boundary for the enclosing entities. The flexibility arises from the result of making `Configuration` as a first-class data. The machinery provides a way to write system specifications that manipulate a set of objects and messages as one unit. Further, specifications to manage a hierarchy of `Configurations` are also possible. As discussed in Section 3.4, one can represent an access control mechanism for shared resources.

As for the *mobility* of objects and/or messages in archtectural semantics, the three primitives of the ambient are too low or too general. One may see two reasons. (1) The rules for the ambient primitives are all two party interactions while *mobility* may be modeled in a compact manner using a *three* party interaction. (2) Every ambient can in principle move around while not all the components are *mobile* in real software systems.

Figure 3.8 illustrates a situation where an ambient `A` moves from the `K` ambient to the `L` ambient. One may also assume that the `K` and the `L` are stationary. The following fragment describes the rule in CafeOBJ.

```
op go : Handle Location Location -> Event
op (_)[_] : Handle Configuration -> Ambient
op (_)[|_|] : Location Configuration -> Node
```

```
var A : Handle  vars K L : Location
vars C1 C2 C3 : Configuration

trans (K)[| (A)[C1] go(A,K,L) C2 |] (L)[| C3 |]
=>    (K)[| C2 |] (L)[| C1 C3 |] .
```

The function symbol `go(A,K,L)` refers to an event that an ambient with `A` moves from a stationary node `K` to another node `L`.

For a comparison, one may emulate the situation (Figure 3.8) by using the ambient rules as below. In the reduction process, a symbol with * (such as `K*`) denotes a constant.

```
(K*)[ (A*)[ out(K*)@(in(L*))@(C1*) ] (C2*) ] (L*)[ open(A*)@(C3*) ]
-> (K)[ (C2*) ] (A)[ in(L*)@(C1*) ] (L*)[ open(A*)@(C3*) ]
-> (K)[ (C2*) ] (L*)[ (A)@[ (C1*) ] open(A*)@(C3*) ]
-> (K)[ (C2*) ] (L*)[ (C1*) (C3*) ]
```

The `L*` ambient should have `open(A*)` primitive in order to *open* the incoming ambient (`A*`) and enable interactions at the enclosing `Condiguration` (objects and messages) level. It shows that the destination ambient with `L*` has to know the name or handle of the incoming ambient. The reduction process is slightly different from what is illustrated in Figure 3.8 with the above three party rule. The latter shows that the destination node does not presume the name or handle of the incoming ambient.

As discussed here, one can test and check various situations with appropriate reduction rules. And importantly every situation can be validated mechanically if one encodes the reduction rules in CafeOBJ. The discussions relating to the Maude with the ambient presents such an example.

This chapter has presented a way to encode in CafeOBJ two of important primitive machineries for describing object-oriented specifications. The machineries, the Maude concurrent object and the ambient, are useful, but are not sufficient for applying CafeOBJ to modeling software systems. Modeling method for CafeOBJ descriptions is the theme of the next chapter.

Chapter 4

Modeling Method for CafeOBJ Specifications

4.1 Introduction

As discussed in Chapter 3, CafeOBJ provides language constructs suitable for the object-oriented algebraic specification. In order to apply the technology to software development, however, a modeling method to help obtain stable specification is desirable. The following two aspects of the modeling method is important: (1) formalization of a given problem description and (2) decomposition of the problem into a set of CafeOBJ specification modules of manageable size. Each module in the resultant CafeOBJ specification is expected to be traced from an entity in the problem domain space.

Object-oriented methodology generally has two roles; (1) it provides basic notations to represent various aspects of *objects*, and (2) it provides further notations and guidelines to help derive object definitions. Different methodologies assume different guidelines but fall into two broad categories [105]: data-driven methodology and responsibility-driven methodology. The latter is sometimes called scenario-based methodology [20]. In data-driven methodology such as OMT [102], structural aspects of objects and relationship between objects are the main concern at an early stage of development; functional aspects of objects are left until later.

In scenario-based methodology, such as that of OOSE [65] or RDD/CRC [121], an object is an entity that is responsible for a particular functionality that the whole system provides. Identifying objects requires analysis of functional behavior

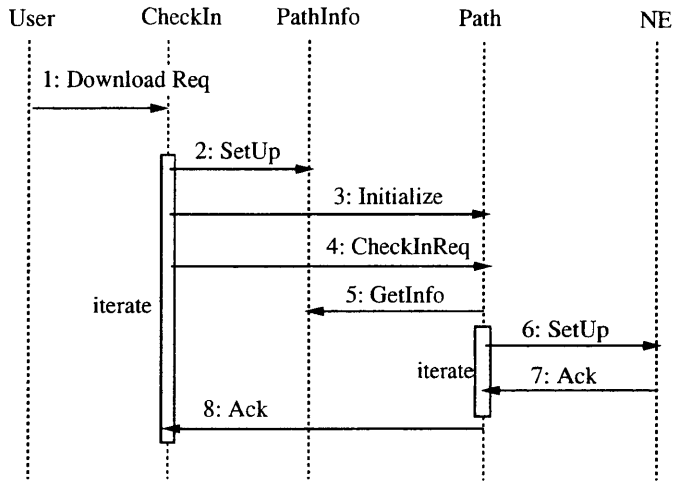


Figure 4.1: Event Trace Diagram

of the system and involves separating out the functional coupling between objects. Thus, the method puts emphasis on validating the functional behavior of more than one object in early stages of the development. It implies that object-oriented descriptions in CafeOBJ may find comfortable places in the scenario-based modeling process because of the executability or a basis for rapid prototyping. This chapter adopts a scenario-based object-oriented method to derive CafeOBJ specifications.

4.2 Scenario-based Modeling Method

A significant object software system usually consists of a lot of constituent objects. The objects work together to implement a coherent functionality that users of the system enjoy. Thus, collective behavior of objects is important [87], whose situation is best summarized as a famous statement [57];

> No object is an island.

Further, since information hiding or data encapsulation is an intrinsic nature of object, each object constituting the system should have a well-defined interface. The above observation has led to scenario-based object modeling method. It focuses on

scenario that represents interactions between constituent objects so as to identify well-defined set of objects. The approach has been successful in the user interface design and now becomes popular in the world of object-oriented modeling [20].

A scenario is actually a prototypical history of an execution and consists of both a sequence of messages exchanged among participant objects. Figure 4.1 is an example representation of scenario used in object-oriented modeling. The horizontal arrow refers to a message sending event, and the numbers in the diagram specify the order of events where an event is a message to some object. The diagram is called a message sequence chart (MSC) or an event trace diagram.

While the usage and abstraction level of the notations differ in each methodology, the essence of a scenario is to have two notions: (1) the information about participant objects, and (2) the information about the sequence of their interactions. Current methodologies employ diagrams as shown in Figure 4.1 to represent scenarios. Since diagram is not rigorous enough to be a basis for mechanical analysis tool, design validation support is difficult so that the quality of the design is solely dependent on design reviews by experienced engineers.

4.3 GILO : Intermediate Design Notation

4.3.1 Overview

In order to integrate CafeOBJ with a scenario-based object-oriented modeling methodology [15][20][121], this section proposes a modeling method that uses an intermediate design notation GILO (Generic Interaction Language for Objects) [87]. And the proposed method provides a guideline to obtain executable CafeOBJ descriptions in accordance with the scenario-based methodology [89][90].

Since a scenario requires multiple features of software design to be represented and its encoding is therefore complicated, it is not easy to write specifications in CafeOBJ directly from scenario-based design descriptions. The proposed approach is to provide an intermediate design notation GILO, which has two abstraction viewpoints, collaboration and class, to capture functional aspects of a scenario [87]. This means that deriving a CafeOBJ description from its GILO counterparts is simply a matter of translating the operational semantics of GILO. The resultant CafeOBJ description consists of a set of modules which faithfully reflect the analyzed structure of the problem. Figure 4.2 illustrates an overview of the development process.

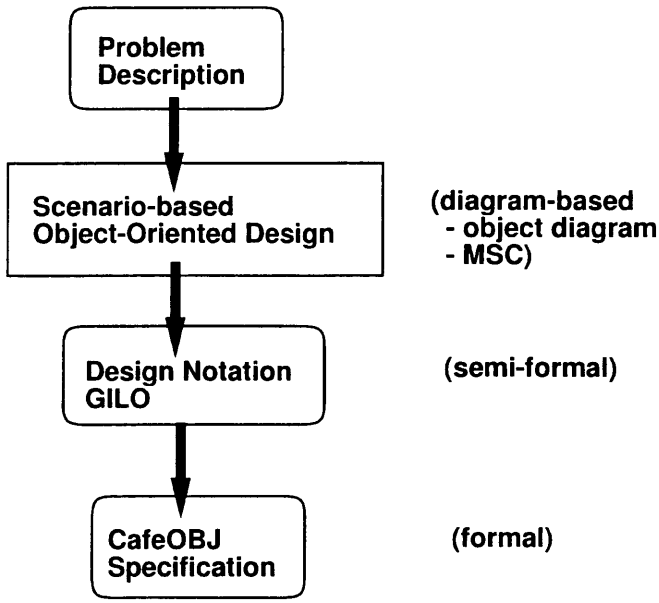


Figure 4.2: Overview of Development Steps

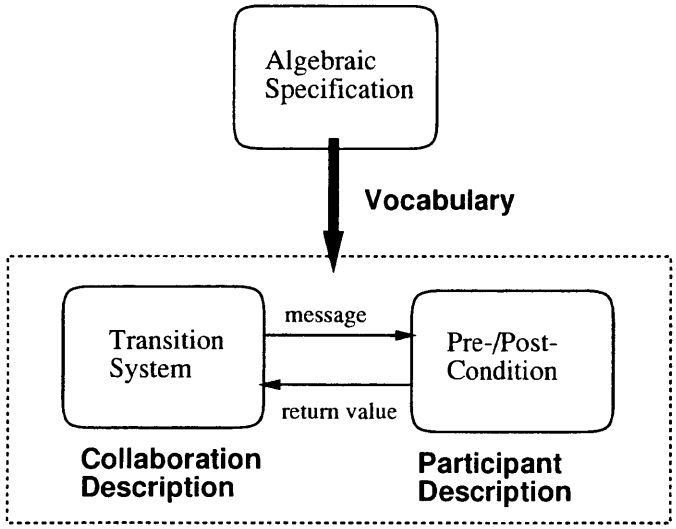


Figure 4.3: GILO Model

A scenario needs to make two aspects of the specification components explicit: the behavior of participant objects and the behavior of global interaction between them. GILO provides two abstraction viewpoints: class for the former component and collaboration for the latter [87]. Since it is difficult to encode both characteristics in one specification model, we use a multiparadigm specification approach [130] and define relationships between the components precisely. Figure 4.3 illustrates the basic idea of GILO, which provides three kinds of specification components: (1) Collaboration, (2) Class, and (3) Common Vocabulary.

Collaboration

Since collaboration is responsible for the way participant objects exchange messages, dynamic reactiveness is essential. For collaboration, we have adopted a transition system, which can be considered an abstract computational model of the Message Sequence Chart (MSC) shown, for example, in Figure 4.1. The model also allows conditional branching and iteration, both of which a MSC cannot express, and thus is more general. Conversely a MSC is a record of an execution history generated by the transition system. Since a message is sent to some of the participant objects in

the course of state transition, the transition system can be considered to explicitly describe how the global interaction of the objects proceeds. Collaboration also includes declarations of participant objects.

Class

In regard to the definition of participant objects, methods of the classes for the objects provide behavioral aspects. Since an object has internal states and method invocation often results in changes in those states, it is natural to model the method behavior in terms of state-oriented specifications or a combination of the pre- and post-conditions. In addition, a method may return some value as its result. We allow to use the introduction of return variables for this purpose. That is, the method can update a predefined variable *ret* to return some value to its caller.

Common Vocabulary

In describing the definition of collaboration or class, we often use auxiliary symbols to which we assign some particular meanings. The third component provides a set of common vocabularies that are employed to interpret each symbol in collaboration or class. We have adopted an abstract datatype technique for this component of GILO specification. This idea is inspired by the Larch family of specification languages [53], where the shared language component defines common vocabulary and the interface language uses the vocabulary to describe the behavioral specifications of procedures or functions. The common vocabulary of GILO corresponds to the shared language of Larch.

4.3.2 GILO by Examples

Next presents the three kinds of GILO components by using concrete examples.

Collaboration Description

Collaboration has two components: (1) declaration of participant objects, and (2) state-machine to represent dynamic behavior. Figure 4.4 shows an example, Collaboration SimpleExample. This has three participant objects; *user* is an object of Class User, *dir* is an object of Class Directory, and *nodes* is an object of Class NodeList that is a list of Node object. A *dir* object maintains the directory information that establishes a mapping between the access key and the nodes. The example

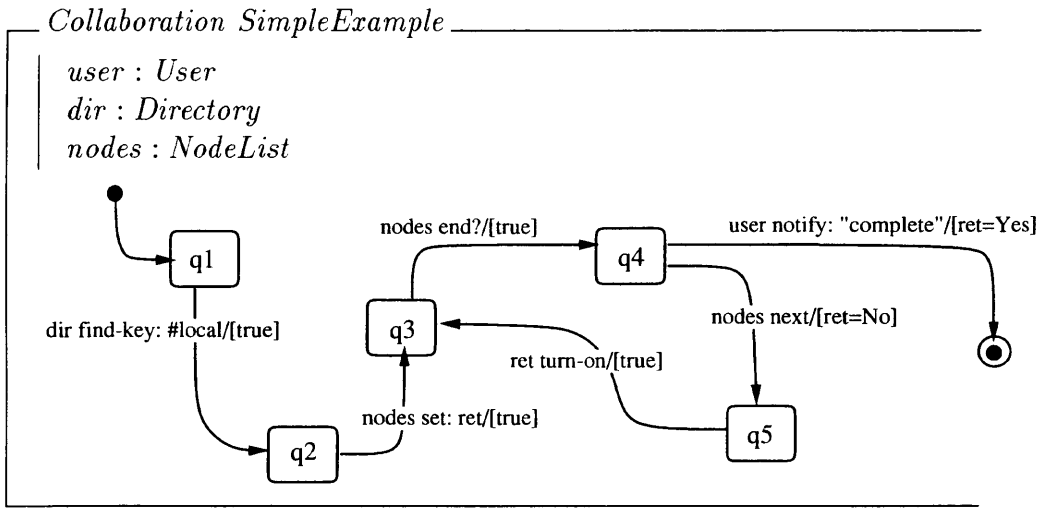


Figure 4.4: Collaboration Example

collaboration describes a flow that (1) obtaining *nodes* object by looking up the *dir*, (2) sending a turn-on message to each node object in the *nodes*, which forms a iterative loop, and (3) sending a notification to *user* when the process completes.

A diagram-based labeled transition system is used to represent dynamic behavior of collaboration. Figure 4.4 is an example, which has 7 states that correspond to a default initial state, five states from q1 to q5, and a final state denoted by $\odot$. And each transition arc has a label denoting a message to one of participant objects and a condition to fire the transition. For example, the expression attached to the arc from q4 to q5 specifies that a *next* message is sent to the object denoted by the variable *nodes* when a return value of the previous message (*ret*) equals to *No*:

nodes next/[ret=No].

In order to express an operational interpretation of the transition system, Coloured PetriNet (CPN) [66] is used. Figure 4.5 is a CPN representation equivalent to the transition system in Figure 4.4. *Colour* of a token v carries an environment to manage status of participant objects.

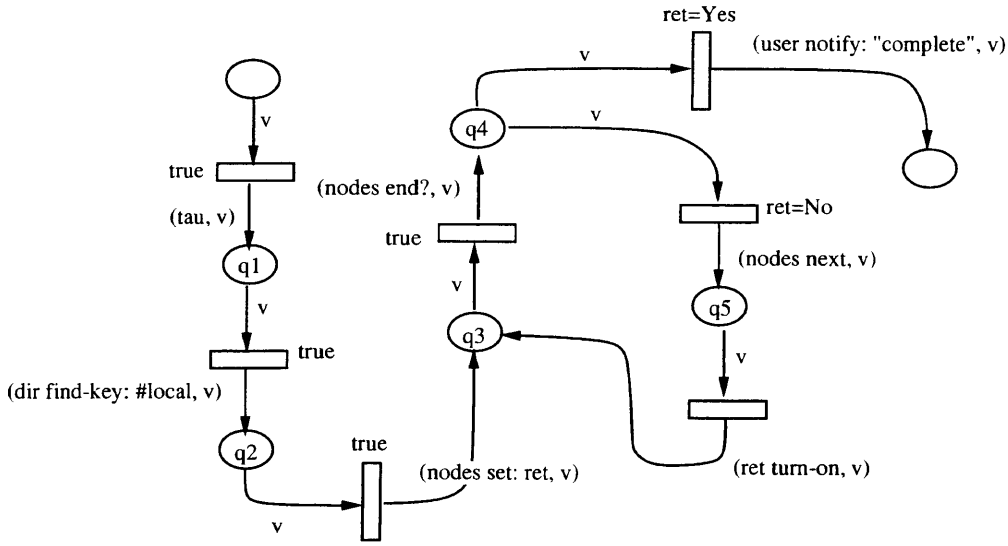


Figure 4.5: Equivalent CPN Description

Class Description

Class also has two components: (1) attributes constituting internal structure of objects, and (2) definitions of method body. Method is invoked when an object receives appropriate messages.

Figure 4.6 shows an example, Class NodeList, whose object is one of the participant of Collaboration SimpleExample. Class uses a box notation as a concrete syntax, which is similar to the Z notation [110]. A small box named State[1] has a set of attribute declarations. In the example, a NodeList object has an attribute *ptr* of sort *List of OId*. The other three are method definitions. For example, the method *next* has a precondition (**assumes:**) of $\neg$ *(null ptr)* and a postcondition (**ensures:**) specifying an update in the attribute *ptr* with a return value (*ret*). And $\Delta(ptr)$ shows that this method updates only the *ptr* attribute in the course of the method execution. Similarly, the *end?* method shows by using $\Xi(ptr)$ that it does not change value of *ptr*.

Common Vocabulary

Collaboration SimpleExample above uses two constants *yes* and *no*. Class NodeList assumes List-related functions such as *head* or *null*. These auxiliary symbols should be defined somewhere.

GILO provides the third component to define such common vocabulary by using order-sorted algebra. Figure 4.7 shows definitions for *YESNO* and *LIST*. As seen from the examples, common vocabulary is introduced as an abstract datatype definition.

4.3.3 Translation to CafeOBJ

The section discusses how to derive CafeOBJ descriptions from GILO representations [92]. Figure 4.8 illustrates several groups of CafeOBJ modules. They constitute *runtime* modules to represent the CafeOBJ modules derived from GILO.

Object System Kernel (Section 3.3.2) is a set of CafeOBJ modules to encode the object model. It is basically an encoding of the Maude concurrent object model, and includes the definitions of concurrent object, message, and configuration that manages the former two components. *Collaboration System Kernel* provides a machinery to encode collaboration. The idea is to define a special class of concurrent

[1] A reserved word.

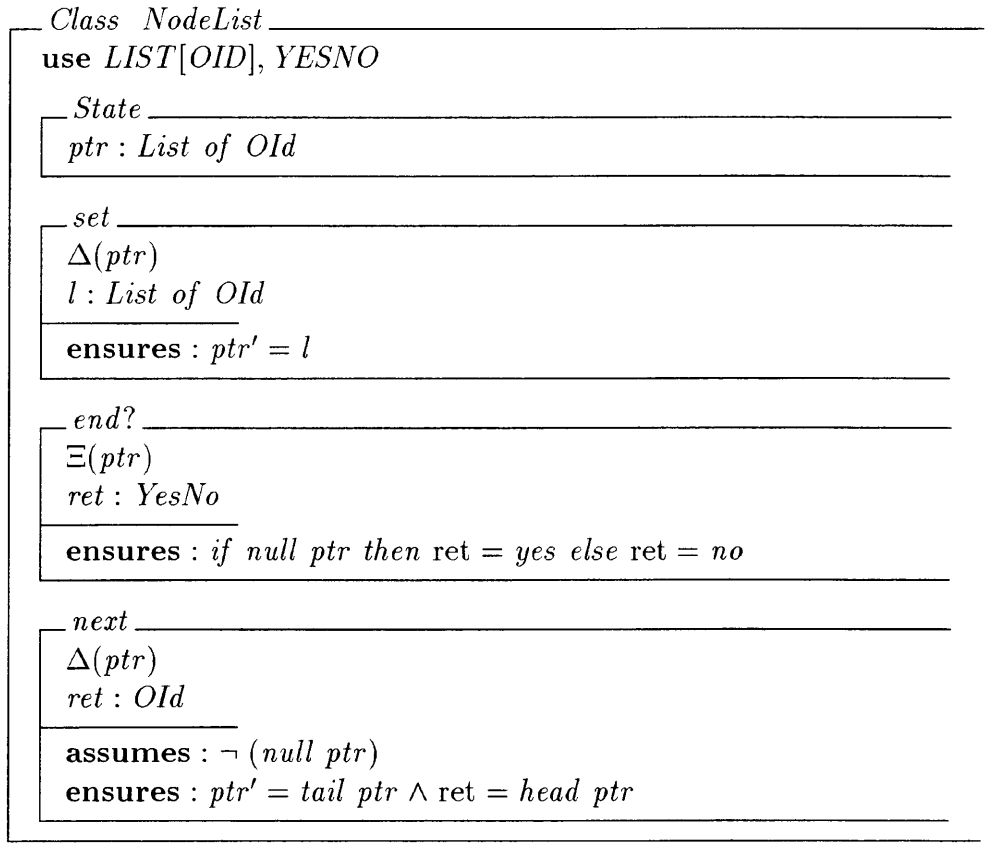


Figure 4.6: Class Example

Module YESNO

[*YesNo*]

| *YesNo* ::= *yes* | *no*

Module LIST[*X* :: *TRIV*]

protecting *NAT*

[*Elem* < *NeList* < *List*]

| *List* ::= *nil* | _ _ : *List* × *List*
| *NeList* ::= _ _ : *NeList* × *List*
| *head* _ : *NeList* → *Elem*
| *tail* _ : *NeList* → *List*
| *null* _ : *List* → *Bool*
| | _ | : *List* → *Nat*

| *E* : *Elem*
| *L* : *List*
| ---
| *head E L* = *E*
| *tail E L* = *L*
| *null nil* = *true*
| *null E L* = *false*
| | *nil* | = 0
| | *E L* | = 1 + | *L* |

Figure 4.7: Common Vocabulary Example

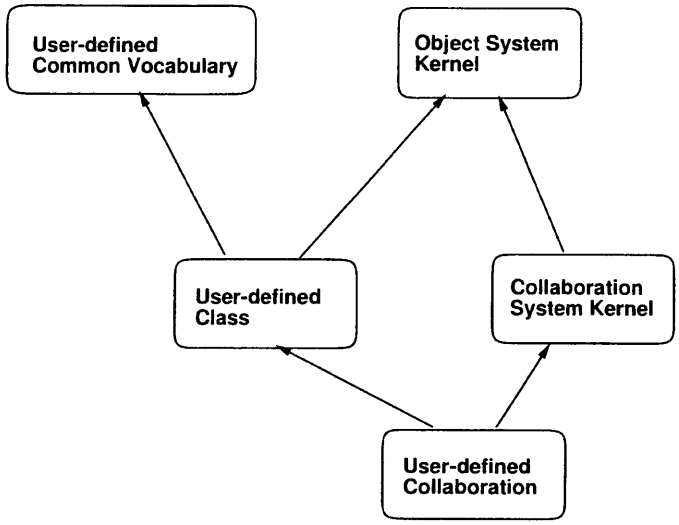


Figure 4.8: Module Relationship Overview

object for representing CPN. The implementation makes use of *Object System Kernel*. *User-defined* modules are categorized into *Common Vocabulary*, *Class*, and *Collaboration*. *User-defined Class* uses *Object System Kernel*, and *User-defined Collaboration* is based on *Collaboration System Kernel*. Each of the user-defined module has a direct counterpart in the GILO description.

Collaboration System

Collaboration System Kernel provides a basic machinery for executable user-defined collaboration. It makes use of the object system and defines general-purpose concurrent objects to represent a CPN interpretation of the labeled transition system used in the collaboration.

In order to simulate *markings* in CPN, module `MARKING` is introduced to represent execution snapshots of a state-machine.

```
mod! MARKING {
  extending (ATTR-VALUE)
  [ State,  Marking ]
  [ State < Marking < AttrValue ]
```

```
  signature {
    op empty : -> Marking
    op (_,_) : Marking Marking -> Marking {assoc comm id: empty}
  }
}
```

The CafeOBJ representation of collaboration is an object belonging to Class Machine, that is implemented by the module MACHINE. A Machine object has two attributes; marking refers to the snapshot mentioned above, and participants denote a list of object identifiers (OIds), each one being a participant of the collaboration. Further, a Machine object can respond to two messages, fire and on, to proceed state transitions. Every CafeOBJ module for user-defined collaboration imports the module MACHINE to become powered for execution.

```
mod! MACHINE {
  extending  (ROOT)
  protecting (MACHINE-MESSAGE)

  [ MachineTerm,  CIdMachine ]
  [ MachineTerm < ObjectTerm,  CIdMachine  < CId ]

  signature {
    op <(_:_)|_> : OId CIdMachine Attributes -> MachineTerm
    op Machine : -> CIdMachine
    op make-machine : OId Marking OIds -> MachineTerm
  }

  axioms {
    var O : OId
    var M : Marking
    var L : OIds

  eq make-machine(O,M,L)
  =  <(O : Machine)| (marking = M), (participants = L)> .
  }
}
```

Following two auxiliary modules are necessary to complete the definition.

```
mod! MACHINE-ATTR-VALUE {
  extending (AID)
  extending (BASIC-VALUE)
  using (COLLECTION[OID] * { sort Collection -> OIds })
  [ OIds < AttrValue ]
  signature {
    op marking : -> AId
    op participants : -> AId
  }
}

mod! MACHINE-MESSAGE {
  extending  (ROOT)
  protecting (MACHINE-ATTR-VALUE)
  protecting (MARKING)
  signature {
    op on : OId -> Message
    op fire : OId -> Message
  }
}
```

Translation Example

Below shows the CafeOBJ modules for the example GILO descriptions (Figures 4.4, 4.6 and 4.7). The modules are obtained by manual translation.

A CafeOBJ module `SIMPLE-EXAMPLE` is a translation of Collaboration Simple-Example. Apart from the `ROOT`, it imports two modules. `SIMPLE-EXAMPLE-STATES` provides constant definitions for the necessary states (`q0`-`q5`, and `qf`). `SIMPLE-EXAMPLE-PARTICIPANTS` contains all the definitions of objects that are participant of the collaboration. The rewriting rules together simulate the transitions specified in the collaboration. Note that the rule template of Merging Multiple Messages in Figure 3.3 (d) is extensively used to express sequential execution of control in the method descriptions. For example, the third rule in `SIMPLE-EXAMPLE` has two messages `on(O)` and `return(O,V)` in order to express that the rule is fired only after the message `return(O,V)` is generated. Here the message `return(O,V)` is a result of a previous invocation of a participant object method. In the case of the third rule, the message `return(O,V)` is generated as a result of `find-key(dir,'local,O)` appeared in the RHS of the second rule. It is a message sent to `dir` object requesting a `NodeList` object that is indexed with `'local`.

```
mod! SIMPLE-EXAMPLE {
  extending (MACHINE)
  protecting (SIMPLE-EXAMPLE-STATES)
  protecting (SIMPLE-EXAMPLE-PARTICIPANTS)
  axioms {
   var O : OId    var M : Marking    var P : OIds
   var REST : Attributes

   trans fire(O)
      <(O : Machine)|(marking = (q0, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q1, M)), (participants = P), (REST)>
      on(O)  .

   trans on(O)
      <(O : Machine)|(marking = (q1, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q2, M)), (participants = P), (REST)>
      find-key(dir, 'local,O) on(O) .

   trans on(O) return(O,V)
      <(O : Machine)|(marking = (q2, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q3, M)), (participants = P), (REST)>
      set(nodes, V,O) on(O) .

   trans on(O) void(O)
      <(O : Machine)|(marking = (q3, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q4, M)), (participants = P), (REST)>
      end?(nodes, O) on(O) .

   ctrans on(O) return(O,V)
      <(O : Machine)|(marking = (q4, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (qf, M)), (participants = P), (REST)>
      notify(user, 'complete, O) on(O) if V == Yes .

   ctrans on(O) return(O,V)
      <(O : Machine)|(marking = (q4, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q5, M)), (participants = P), (REST)>
      next(nodes, O) on(O) if V == No .

   trans on(O) return(O,V)
```

```
      <(O : Machine)|(marking = (q5, M)), (participants = P), (REST)>
   => <(O : Machine)|(marking = (q3, M)), (participants = P), (REST)>
      turn-on(V, O) on(O) .
  }
}
```

The next CafeOBJ module is a translation of Class NodeList (Figure 4.6). Rewriting rules (`trans`、`ctrans`) correnponds to the three methods.

```
mod! CLASS-NODE-LIST {
  extending (ROOT)
  protecting (NODE-LIST-MSG)
  [ NodeListTerm, CIdNodeList ]
  [ NodeListTerm < ObjectTerm, CIdNodeList < CId ]
  signature {
    op <(_:_)|_> : OId CIdNodeList Attributes -> NodeListTerm
    op NodeList  : -> CIdNodeList
  }
  axioms {
    vars O R  : OId
    vars L L' : List
    var REST  : Attributes

    trans set(O,L,R) <(O : NodeList)|(ptr = L'), (REST)>
    => <(O : NodeList)|(ptr = L), (REST)> void(R) .

    ctrans end?(O,R) <(O : NodeList)|(ptr = L), (REST)>
    => return(R, Yes) <(O : NodeList)|(ptr = L), (REST)> if null(L) .

    ctrans end?(O,R) <(O : NodeList)|(ptr = L), (REST)>
    => return(R, No) <(O : NodeList)|(ptr = L), (REST)> if not null(L) .

    ctrans next(O,R) <(O : NodeList)|(ptr = L), (REST)>
    => return(R, head(L)) <(O : NodeList)|(ptr = tail(L)), (REST)>
    if not null(L) .
  }

}
```

Common vocabulary of GILO is just a syntax-suger of CafeOBJ module. A translation to CafeOBJ is straightforward.

```
mod! YES-NO {
  [ YesNo ]
  signature {
    ops Yes No : -> YesNo
  }
}

mod! LIST [X :: TRIV] {
  protecting (NAT)
  [ NeList List, Elem < NeList < List ]
  signature {
    op nil : -> List
    op __  : Lilst List -> List
    op __  : NeList List -> List
    op head : NeList -> Elem
    op tail : NeList -> List
    op null : List -> Bool
    op |_|  : List -> Nat
  }
  axioms {
    var E : Elem
    var L : List

    eq head (E L) = E .
    eq tail (E L) = L .
    eq null (nil) = true .
    eq null (E L) = false .
    eq | nil | = 0 .
    eq |(E L)| = 1 + |(L)| .
  }
}
```

4.4 A Case Study : Object-Oriented Modeling

This section presents a case study in which CafeOBJ specifications are derived by following the proposed development steps (Figure 4.2) [90][92]. Key points of the steps are (1) to construct scenarios by identifying participant objects and their interactions, and (2) to construct GILO descriptions. The case study illustrates the

role of three GILO components in the overall specification.

4.4.1 *SAKE* Warehouse Problem

The *SAKE* Warehouse problem is a standard common problem [127]. Since its first appearance in the literature, it has been used as a standard benchmark problem for comparing various design methodologies in the software engineering community in Japan [112][127]. The problem is compact but has essential features commonly found in a lot of business application software.

The following is an English translation of the *SAKE* Warehouse Problem [112].

> A warehouse of X Sake Retailing Company accepts several containers everyday. Each container contains sake bottles, possibly of multiple brands. The number of brands that can be mixed in one container is up to ten. The total number of brands to be treated is about 200.
>
> A warehouse keeper stores each container carried into the warehouse without any rearrangement and sends a container contents notice to a clerk. He also ships out sake bottles by the shipment direction forwarded from the clerk. Stored bottles are never repacked into another container, nor kept in another place. An emptied container is immediately carried out of the warehouse.
>
> container contents notice:
>
> container number (5 digits)
>
> carried-in time (hour/day, month/year)
>
> brand, quantity (repeat)
>
> The clerk receives dozens of shipment orders per day and sends a shipment direction to the keeper for each order. An order comes by an order form or by telephone and each order must designate just one brand. If the brand is out of stock or in short for the ordered quantity, the clerk will tell it to the customer and adds the order to the waiting list. And when the designated brand is supplied to meet the order, the clerk will issue a shipment direction.
>
> In a shipment direction, containers that will become empty are notified.
>
> Develop a system that supports the work of the clerk (notifying out of stock status, issuing shipment direction forms and listing the outstanding orders).

shipment direction form:
    order number
    customer number
    container number
    brand, quantity
    empty mark
waiting list:
    customer name
    brand, quantity

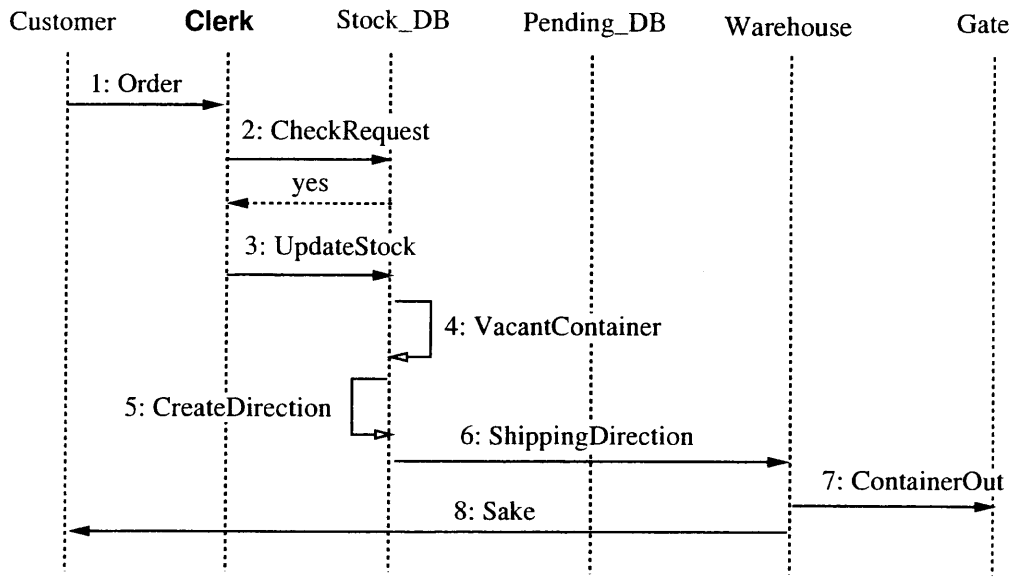
- No loss of sake will occur either during the transportation or during the storage.
- As some part of the problem description may not be realistic, sophisticated functions such as exception handling can be minimal.
- Ambiguities may be resolved by appropriate interpretation.

4.4.2 Problem Scenarios

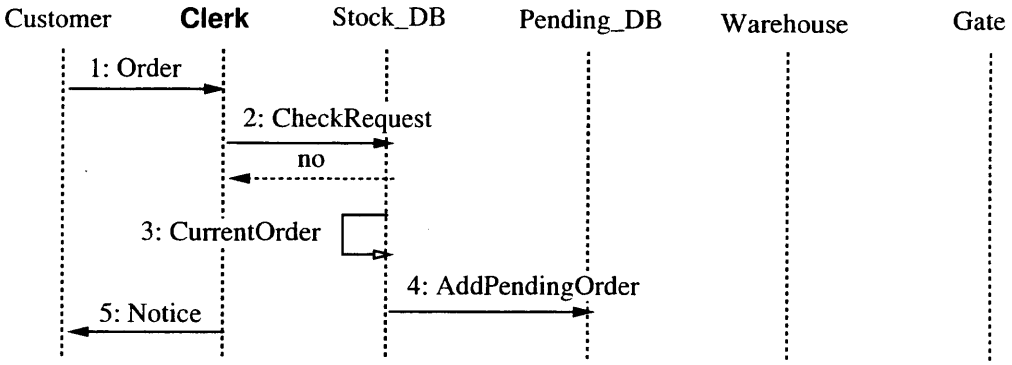
The modeling step starts with the scenario construction. Analysis of the problem description results in identification of two main scenarios, *Order Arrival from Customer* and *Container Arrival*. Scenario in general has more than one subscenarios: one for a main flow and others for handling exceptional cases. For example, the main flow of the *Order Arrival from Customer* scenario is to deliver requested bottles of sake, while the order is added in a waiting list when enough stock is not available.

Figure 4.9 shows two subscenarios for the *Order Arrival from Customer* by using Message Sequence Charts (MSC); (a) the retailing company has enough stock to fulfill the order, and (b) the order is added to the pending order database because the stock is insufficient. In the normal case (a), the arriving order initiates the subscenario (step a1). This step is followed by a check of whether there is enough stock to fulfill the order (step a2). If so, the stock database is updated (step a3). After empty containers are collected (step a4), a shipping direction to the warehouse keeper is created (step a5). In the case (b), on the other hand, the order is added to the pending order database (step b4) and a notification is issued that the order is in the waiting list (step b5).

In the process of constructing the above MSCs, the responsibility or abstract functionality of each participant object is identified. Of the six participants, the



(a) In Stock



(b) Out of Order

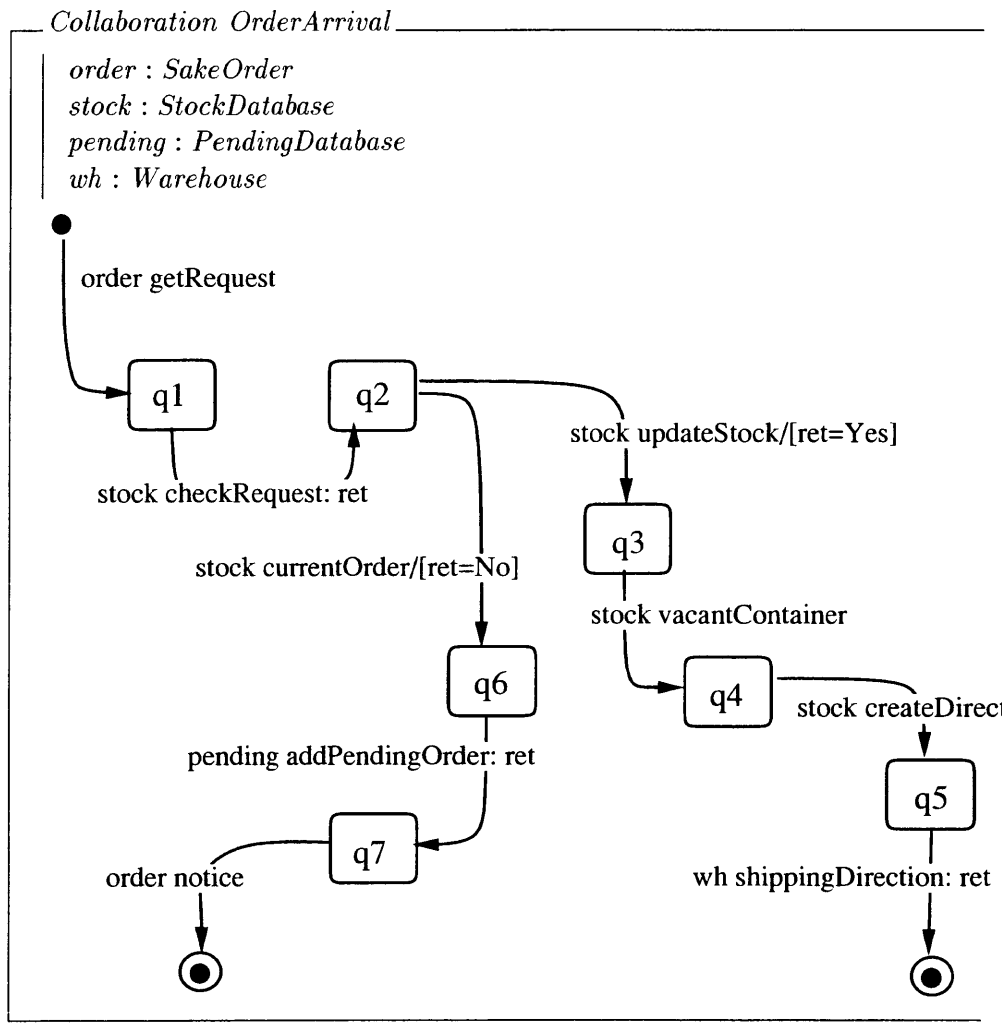
Figure 4.9: Order Arrival from Customer

two key objects in the subscenarios are the Stock Database and the Pending Order Database. Other objects may be considered auxiliary and constitute the environment in which the whole scenarios are described completely. Constructing another scenario, Container Arrival, also helps elaborate the definitions of two database objects.

4.4.3 GILO Descriptions

The second step involves construction of the GILO specification. Since GILO has three components, division of labor between them is a key aspect of the specification construction. First, since collaboration is responsible for the global flow of messages between participant objects, it is constructed by combining MSCs of subscenarios that together constitute one scenario. Second, since an object has states and is modeled in terms of state changes, writing GILO class involves to find attribute data that a participant object maintains internally and methods that operate on the data. Third, common vocabulary modules are introduced. The modules provide interpretation of symbols used in object methods and are purely functional (no side-effect).

First, below is a GILO description of Collaboration OrderArrival. It is constructed by combining the two MSCs in Figure 4.9 with an introduction of appropriate conditional branching at the state q2. The transition sequence from q2 to q5 corresponds to the normal case shown in Figure 4.9 (a) while the sequence from q2 to q7 corresponds to Figure 4.9 (b).



In formalizing GILO model of collaboration, arguments of message are also identified so that all the information necessary to define object interfaces is determined. Note that a collaboration does not explicitly specify the initiator object (an object that sends a message) but shows only the sequence of message events. Collaboration is, therefore, somewhat more abstract than a MSC that explicitly specifies the message sender as well as the receiver.

Below is a partial description of Class StockDatabase. It defines the interface specification of the class that is in accordance with the collaboration. In identify-

ing the interface specification of Class StockDatabase, we have taken into account Collaboration ContainerArrival as well as Collaboration OrderArrival, although the current discussion presents the latter one only. Of the seven method in the definition, *addNewStock* and *selectPendingOrders* come from Collaboration ContainerArrival.

Class StockDatabase

addNewStock : *List of Stock* → *List of Stock*
checkRequest : *Request* → *YesNo*
updateStock : *void* → *void*
vacantContainer : *void* → *void*
createDirection : *void* → *ShippingDirection*
selectPendingOrders : *List of Request* → *List of Request*
currentOrder : *void* → *Request*

The next step is to elaborate the internal structure of the class and the functional specification of each method.

Class StockDatabase

use *STOCK_DB*, *YESNO*, *NAT*

State

$stock\_number : NzNat$
$order\_number : NzNat$
$contents : StockDB$
$match : List\ of\ Stock$
$vacant : List\ of\ Container$
$current : Request$

Init

$State'$

---

ensures : $stock\_number' = 1$
$\wedge\ order\_number' = 1$
$\wedge\ contents' = nilDB$
$\wedge\ match' = nilLS$

addNewStock

$\Delta(contents, stock\_number)$
$ls, ret : List\ of\ Stock$

---

ensures : $contents' = add\_DB(contents, ls, stock\_number)$
$\wedge\ stock\_number' = stock\_number+ \mid ls \mid$
$\wedge\ ret = ls$

checkRequest

$\Delta(match, current)$
$\Xi(contents)$
$req : Request$
$ret : YesNo$

---

assumes : $\neg\ (contents == nilDB)$
ensures : $match' = check(contents, brand(req), quantity(req))$
$\wedge\ current' = req$
$\wedge\ (if\ match' == nil\ then\ ret = No\ else\ ret = Yes)$

---

assumes : $contents == nilDB$
ensures : $ret = No$

Class StockDatabase(cont.)

updateStock

$\Delta(contents, vacant)$
$\Xi(match)$

assumes : $\neg\ (match == nil)$
ensures : $contents' = update(contents, match)$
$\wedge\ vacant' = nil$

vacantContainer

$\Delta(vacant)$
$\Xi(contents)$

assumes : $vacant == nil \wedge \neg\ (contents == nil)$
ensures : $vacant' = collect\_vacant(contents)$

createDirection

$\Delta(match, vacant, order\_number)$
$\Xi(contents)$
$ret : ShippingDirection$

assumes : $\neg\ (current == nil)$
ensures :
$ret = create\_direction(order\_number, client(current), match, vacant)$
$\wedge\ match' = nil$
$\wedge\ vacant' = nil$
$\wedge\ order\_number' = order\_number + 1$

selectPendingOrders

$\Xi(contents)$
$ls : List\ of\ Request$
$ret : List\ of\ Request$

ensures : $ret = first\_come\_first\_serve(contents, ls)$

currentOrder

$\Xi(current)$
$ret : Request$

ensures : $ret = current$

The StockDatabase consists of several attributes that constitute the object states. The attribute *contents* maintains the content of the database, and *stock_number* keeps track of a value that gives a unique identification number to each stock that the database has. Three other attributes are used to store values that are processed in the course of executing the collaboration.

Class StockDatabase also declares that it uses the common vocabularies, STOCK_DB, YESNO, and NAT, to describe its own behavioral specification. The module STOCK defines an abstract datatype to represent stock, and provides a constructor *stock* and other functions such as *container*.

Module STOCK

protecting *SAKE_BASICS*, *NAT*

[*Stock*]

$Stock ::= stock : Container \times Brand \times Nat \times Nat$
$container\_ : Stock \rightarrow Container$
$brand\_ : Stock \rightarrow Brand$
$quantity\_ : Stock \rightarrow Nat$
$id\_ : Stock \rightarrow Nat$
$decr\_stock\_quantity\_ : Stock \times Nat \rightarrow Stock$

$C : Container$
$B : Brand$
$Q\ Q'\ I : Nat$

$container(stock(C, B, Q, I)) = C$
$brand(stock(C, B, Q, I)) = B$
$quantity(stock(C, B, Q, I)) = Q$
$id(stock(C, B, Q, I)) = I$
$decr\_stock\_quantity(stock(C, B, Q, I), Q') = stock(C, B, Q - Q', I)$

The module STOCK_DB defines functions to realize the main functionality of Class StockDatabase.

Module STOCK_DB

using $DB[STOCK]$
using $LIST[STOCK]*(sort\ List\ to\ ListStock, op\ nil\ to\ nilLS)$
using $LIST[REQUEST]*(sort\ List\ to\ ListRequest, op\ nil\ to\ nilRQ)$
using $LIST[CONTAINER]*(sort\ List\ to\ ListContainer, op\ nil\ to\ nilC)$

$add\_DB\ \_ : Database \times ListStock \times Nat \rightarrow Database$
$stock\_to\_DB\ \_ : ListStock \times Nat \rightarrow Database$
$check\ \_ : Database \times Brand \times Nat \rightarrow ListStock$
$collect\_brand\ \_ : Database \times Brand \times ListStock \rightarrow ListStock$
$check\_quantity\ \_ : Database \times Nat \times ListStock \rightarrow ListStock$
$update\ \_ : Database \times ListStock \rightarrow Database$
$update\_aux\ \_ : Database \times ListStock \times Database \rightarrow Database$
$collect\_vacant\ \_ : Database \rightarrow ListContainer$
$first\_come\_first\_serve\ \_ : Database \times ListRequest \rightarrow ListRequest$

$D : Database$
$LS : ListStock$
$I\ J\ Q : Nat$
$C : Container$
$B : Brand$

$add\_DB(D, LS, I) = (stock\_to\_DB(LS, I)\ D)$
$stock\_to\_DB(nilLS, I) = nilLS$
$stock\_to\_DB(stock(C, B, Q, J)LS, I) = (stock(C, B, Q, I)\ stock\_to\_DB(LS, I+1))$

$D : Database$
$B : Brand$
$X : Stock$
$LS\ LS' : ListStock$
$I\ Q : Nat$

$check(D, B, Q) = check\_quantity(collect\_brand(D, B, nilSL), Q, nilLS)$
$collect\_brand(nilDB, B, LS) = LS$
$collect\_brand((X\ D), B, LS)$
$= if\ (brand(X) == B)\ then\ collect\_brand(D, B, (X\ LS))$
$else\ collect\_brand(D, B, LS)$
$check\_qauntity(nilLS, Q, LS) = if\ Q < 0\ then\ nilLS\ else\ LS$
$check\_quantity((S\ LS'), I, LS) = check\_stock((S\ LS'), I - quantity(S), LS)$
$\ldots(omitted)\ldots$

$\ldots(omitted)\ldots$

The module STOCK_DB is basically a parameterized DB module that has STOCK as the actual parameter. The DB module is a basic one common to both STOCK_DB and PENDING_DB, the latter of which provides vocabulary for Class Pending-Database. The module STOCK_DB adds further auxiliary functions to the module DB so that Class StockDatabase is defined in a compact manner.

4.4.4 CafeOBJ Descriptions

The final step of the modeling process is simply to translate the GILO descriptions into the CafeOBJ modules. Below shows some of the examples.

The module ORDER-ARRIVAL is a CafeOBJ version of Collaboration ORDER-ARRIVAL.[2]

```
mod! ORDER-ARRIVAL {
  extending (MACHINE)
  protecting (ORDER-ARRIVAL-STATE)
  protecting (ORDER-ARRIVAL-PARTICIPANTS)

  signature { op history : -> AId   }

  axioms {
  var O : OId
  var C : CIdMachine
  vars M H : Marking
  var REST : Attributes
  var Q : Request
  var V : YesNo
  var D : Direction

  trans on(O) <(O : C)| (marking = (q0, (M))),
                        (participants = ('order 'stock 'pending 'wh)),
                        (history = (H)), (REST)>
  =>  get-request('order,O)
     <(O : C)| (marking = (q1, (M))),
               (participants = ('order 'stock 'pending 'wh)),
```

[2]We have added the `history` attribute for recording the execution history of the collaboration in order to *debug* the behavior.

```
            (history = (q1, H)), (REST)> on(O) .

trans on(O) <(O : C)| (marking = (q1, (M))),
                      (participants = ('order 'stock 'pending 'wh)),
                      (history = (H)), (REST)>
   return(O,Q)
=>  check-request('stock,Q,O)
   <(O : C)| (marking = (q2, (M))),
             (participants = ('order 'stock 'pending 'wh)),
             (history = (q2, H)), (REST)> on(O) .

ctrans on(O) <(O : C)| (marking = (q2, (M))),
                       (participants = ('order 'stock 'pending 'wh)),
                       (history = (H)), (REST)>
   return(O,V)
=>  current-order('stock,O)
   <(O : C)| (marking = (q6, (M))),
             (participants = ('order 'stock 'pending 'wh)),
             (history = (q6, H)), (REST)> on(O)
if V == No .

ctrans on(O) <(O : C)| (marking = (q2, (M))),
                       (participants = ('order 'stock 'pending 'wh)),
                       (history = (H)), (REST)>
   return(O,V)
=>  update-stock('stock,O)
   <(O : C)| (marking = (q3, (M))),
             (participants = ('order 'stock 'pending 'wh)),
             (history = (q3, H)), (REST)> on(O)
if V == Yes .

trans on(O) <(O : C)| (marking = (q6, (M))),
                      (participants = ('order 'stock 'pending 'wh)),
                      (history = H), (REST)>
   return(O,Q)
=>  add-pending-order('pending,Q,O)
   <(O : C)| (marking = (q7, (M))),
             (participants = ('order 'stock 'pending 'wh)),
             (history = (q7, H)), (REST)> on(O) .
```

```
  trans on(O) <(O : C)| (marking = (q7, (M))),
                        (participants = ('order 'stock 'pending 'wh)),
                        (history = (H)), (REST)>
     void(O)
  =>  notice('order,O)
     <(O : C)| (marking = (f1, (M))),
               (participants = ('order 'stock 'pending 'wh)),
               (history = (f1, H)), (REST)> .

  trans on(O) <(O : C)| (marking = (q3, (M))),
                        (participants = ('order 'stock 'pending 'wh)),
                        (history = (H)), (REST)>
     void(O)
  =>  vacant-container('stock,O)
     <(O : C)| (marking = (q4, (M))),
               (participants = ('order 'stock 'pending 'wh)),
               (history = (q4, H)), (REST)> on(O) .

  trans on(O) <(O : C)| (marking = (q4, (M))),
                        (participants = ('order 'stock 'pending 'wh)),
                        (history = H), (REST)>
     void(O)
  =>  create-direction('stock,O)
     <(O : C)| (marking = (q5, (M))),
               (participants = ('order 'stock 'pending 'wh)),
               (history = (q5, H)), (REST)> on(O) .

  trans on(O) <(O : C)| (marking = (q5, (M))),
                        (participants = ('order 'stock 'pending 'wh)),
                        (history = H), (REST)>
     return(O,D)
  =>  shipping-direction('wh,D,O)
     <(O : C)| (marking = (f2, (M))),
               (participants = ('order 'stock 'pending 'wh)),
               (history = (f2, H)), (REST)> .
  }
}
```

The module CLASS-STOCK-DB-BEHAVIOR corresponds to the body of Class StockDB.

```
mod! CLASS-STOCK-DB-BEHAVIOR [X :: CLASS-STOCK-DB-ATTR,
                              Y :: CLASS-STOCK-DB-MSG]  {
  extending (ROOT)

  [ StockDBTerm,  CIdStockDB ]
  [ StockDBTerm < ObjectTerm ]
  [ CIdStockDB  < CId ]

  signature {
    op <(_:_)|_> : OId CIdStockDB Attributes -> StockDBTerm
    op StockDatabase : -> CIdStockDB

    op make-stock-database : OId -> StockDBTerm
    op cr-body : StockDBTerm Request Stocks OId -> Configuration
  }

  axioms {
  vars O R : OId
  var REST : Attributes
  vars  L M M1 : Stocks
  var D : StockDB
  var RS : Requests
  var   V : Containers
  vars Q Q1  : Request
  var  N : Nat

  eq make-stock-database(O)
  = <(O : StockDatabase)| (stock-number = 1),
                          (order-number = 1),
                          (contents  = nilSDB),
                          (match  = nilS),
                          (vacant = nilC),
                          (current  = no-request) > .


  trans add-new-stock(O,L,R)
```

```
  <(O : StockDatabase)| (stock-number = N),
                        (contents  = D), REST >
=> <(O : StockDatabase)| (stock-number = N + length(L)),
                         (contents  = add-db(D,L,N)), REST >
  return(R,L) .


ctrans check-request(O,Q,R)
  <(O : StockDatabase)| (contents  = D), (current = Q1), REST >
=> return(R,No) <(O : StockDatabase)| (contents  = D), (current = Q), REST >
if (D == nilSDB) .

ctrans check-request(O,Q,R) <(O : StockDatabase)| (contents  = D), REST >
=> cr-body(<(O : StockDatabase)| (contents  = D), REST >,
           Q, check(D,brand(Q),quantity(Q)), R)
if not (D == nilSDB) .


ceq cr-body(<(O : StockDatabase)| (match  = M1), (current  = Q1), REST >,
           Q, M, R)
= return(R,No)
   <(O : StockDatabase)|(match  = M), (current  = Q), REST >
if M == nilS .

ceq cr-body(<(O : StockDatabase)| (match  = M1), (current  = Q1), REST >,
           Q, M, R)
= return(R,Yes)
   <(O : StockDatabase)|(match  = M), (current  = Q), REST >
if not(M == nilS) .

ctrans update-stock(O,R)
   <(O : StockDatabase)| (contents  = D),
                         (match  = M),
                         (vacant = V), REST >
=> void(R) <(O : StockDatabase)| (contents  = update-stock(D,M)),
                         (match  = M),
                         (vacant = nilC), REST >
if not(M == nilS) .
```

```
  ctrans vacant-container(O,R)
     <(O : StockDatabase)| (contents  = D),
                           (vacant = V), REST >
  => void(R) <(O : StockDatabase)| (contents  = D),
                                   (vacant = collect-vacant(D)), REST >
  if ((V == nilC) and (not(D == nilSDB))) .


  trans create-direction(O,R)
     <(O : StockDatabase)| (order-number = N),
                           (match  = M),
                           (vacant = V),
                           (current = Q), REST >
  =>  return(R, direction(N,client(Q),M,V))
     <(O : StockDatabase)| (order-number = (N + 1)),
                           (match  = nilS),
                           (vacant = nilC),
                           (current = no-request), REST > .


  trans select-pending-orders(O,RS,R)
     <(O : StockDatabase)| (contents  = D), REST >
  => return(R, first-come-first-serve(D,RS))
    <(O : StockDatabase)| (contents  = D), REST > .

  trans current-order(O,R) <(O : StockDatabase)| (current  = Q), REST >
  => return(R,Q) <(O : StockDatabase)| (current  = Q), REST > .
  }
}
```

The module STOCK is the common vocabulary STOCK.

```
mod! STOCK {
  [ Stock ]

  protecting (CONTAINER)
  protecting (SAKE-BASICS)
```

```
  protecting (NAT)
  protecting (INT)

  signature {
    op stock : Container Brand Int Nat -> Stock
    op container : Stock -> Container
    op brand     : Stock -> Brand
    op quantity  : Stock -> Int
    op id        : Stock -> Nat
    op decr-stock-quantity : Stock Int -> Stock
  }

  axioms {
  vars N D : Int
  var  I : Nat
  var  C : Container
  var B  : Brand

  eq container(stock(C,B,N,I)) = C .
  eq brand(stock(C,B,N,I)) = B .
  eq quantity(stock(C,B,N,I)) = N .
  eq id(stock(C,B,N,I)) = I .
  ceq decr-stock-quantity(stock(C,B,N,I),D)
  =   stock(C,B,(N - D),I)  if (((N - D) > 0) or (N == D)) .
  }
}
```

The module STOCK-DB is the common vocabulary STOCK_DB.

```
mod! STOCK-DB {
  protecting (DB[IDTRIV2STOCK]
           *{ sort NeDB -> NeStockDB,
              sort DB  -> StockDB,
              op   nilDB -> nilSDB })

  protecting (LIST-OF-REQUESTS)
  protecting (LIST-OF-STOCKS)
  protecting (LIST-OF-CONTAINERS)
```

```
signature {
  op add-db : StockDB Stocks Nat -> StockDB
  op stock-to-db : Stocks Nat -> StockDB

  op update-stock : StockDB Stocks -> StockDB
  op update-aux : StockDB Stocks StockDB -> StockDB
  op collect-vacant : StockDB -> Containers
  op cv-aux         : StockDB Containers -> Containers

  op check : StockDB Brand Int -> Stocks
  op collect-brand : StockDB Brand Stocks -> Stocks
  op check-quantity : Stocks Int Stocks -> Stocks
  op check-quantity-aux : Stocks Int Stocks -> Stocks

  op first-come-first-serve : StockDB Requests -> Requests
}

axioms {
vars D D1 : StockDB
vars LS LS1 LD : Stocks
vars S X : Stock
var  Q : Int
var C : Container
var B : Brand
vars LC LC1 : Containers
vars I J : Nat

eq add-db(D,LS,I) = append(stock-to-db(LS,I), D) .
eq stock-to-db(nilS, I) = nilSDB .
eq stock-to-db((stock(C,B,Q,J) LS),I)
=  stock(C,B,Q,I) ! stock-to-db(LS,(I + 1)) .

eq check(D,B,Q) = check-quantity(collect-brand(D,B,nilS),Q,nilS) .
eq collect-brand(nilSDB,B,LS) = LS .
eq collect-brand((X ! D),B,LS)
=  if (brand(X) == B) then collect-brand(D,B,(X LS))
                      else collect-brand(D,B,LS)     fi .

eq check-quantity(nilS,Q,LS)   = if (Q > 0) then nilS else LS fi .
```

```
  eq check-quantity((X  LD),Q,LS)
  =  check-quantity-aux((X  LD),(Q - quantity(X)),LS) .

  eq check-quantity-aux((X  LD),Q,LS)
  =  if (Q > 0) then check-quantity(LD,Q,(X LS))
                else (decr-stock-quantity(X,(- Q)) LS) fi .

  eq update-stock(D,LS) = update-aux(D,LS,nilSDB) .
  eq update-aux(D,nilS,D1) = append(reverse(D1), D) .
  eq update-aux((X ! D),(S LS),D1)
  =  if (id(X) == id(S))
        then update-aux(D,LS, (decr-stock-quantity(X,quantity(S)) ! D1))
        else update-aux(D,(S LS),(X ! D1))
     fi .

  eq collect-vacant(D) = cv-aux(D,nilC) .

  eq cv-aux(nilSDB,LC)  = LC .
  eq cv-aux((X ! D),LC)
  =  if (quantity(X) == 0)
        then (if not(container(X) in LC) then cv-aux(D, (container(X) LC))
                                 else cv-aux(D,LC)      fi)
        else cv-aux(D,LC)
     fi .
  }
}
```

4.4.5 Summary

The resultant CafeOBJ descriptions for the Sake Warehouse Problem consist of 47 modules that have some 1,100 lines of CafeOBJ code. Table 4.1 summarizes the figures.

The entry GILO Mechanism includes both Object System Kernel and Collaboration System Kernel that basically defines the module `MACHINE` (section 4.3.3). Of fifteen Common Vocabulary modules, four are general-purpose such as `YES-NO` and `DB`, while the rest eleven modules are specific to the present problem. The latter includes `STOCK` and `STOCK-DB` and can be considered as *domain-specific vocabulary*.

The specified class and collaboration for the case is four and one respectively.

	Category	CafeOBJ	
		total	direct
1	GILO Mechanism	13	–
2	Common Vocabulary	15	15
3	Class	15	12
4	Collaboration	3	3
5	Main	1	–
	Total	47	30

Table 4.1: Module Summary

For describing a GILO class, three CafeOBJ modules (a `mod!` module for the class body, two `mod*` modules for the names of attributes and messages) are introduced according to the guideline in Section 3.2.2. Thus, of the fifteen modules in the category Class, twelve ($12 = 3 \times 4$) have direct correspondence with GILO descriptions. The rest three are `mod!` modules to provide concrete representation for the attribute names, the attribute value type, and the message terms. A GILO collaboration, in turn, needs three CafeOBJ modules. It implies that the traceability of the collaboration is quite clear.

4.5 Discussions

This chapter proposed a new object-oriented modeling method for obtaining executable CafeOBJ specifications. The approach is an integration of conventional object-oriented modeling method (scenario-based modeling) and FDT (CafeOBJ). The key idea is an intermediate design notation GILO that bridges a gap between the concepts found in the scenario-based object-oriented design method and those of algebraic specifications in CafeOBJ. With a case study on a standard benchmark problem (the SAKE Warehouse problem), we showed concretely how the proposed method was applied.

One thing to note is that the GILO notation is multiparadigm. The whole GILO specification has a global state consisting of (1) the states in collaboration and (2) the states in all the participant objects. One might argue that both components are state-based and thus the multiparadigm element is minimal. Although both class and collaboration are state-based, the roles of states are different in each component.

More importantly, each has its own syntax that expresses the essential aspects of the computational model in a very concise manner. Thus GILO can be thought of being multiparadigm from the viewpoint of the notational suitability.

The contributions of the present chapter to the area of the algebraic specification technologies are summarized below.

1. Expressibility

   The proposal includes a critical analysis of scenario-based modeling method, which results in identifying three key elements to express the specifications: collaboration, class and common vocabulary. Collaboration keeps track of a global control flow, class defines entities that have local states, and common vocabulary provides entities that have purely functional properties.

   With a suitable encoding method based on the Maude concurrent object model (Chapter 3), collaboration as well as class can be represented in CafeOBJ. This illustrates that CafeOBJ is expressive enough to cover the richness of the concepts found in the scenario-based object-oriented modeling method.

2. Module Decomposition

   According to the proposed modeling method with GILO, the resultant CafeOBJ modules are either collaboration, class, or common vocabulary. Since each module has a specific role, dividing the whole specification into a set of such modules facilitates finding and defining CafeOBJ module specifications. Further, as shown in Table 4.1, each module in the resultant CafeOBJ specification can easily be traced from an entity in the problem domain space. Thus, traceability is good.

3. Use in Development Process

   As shown in Figure 4.2, a whole development process is an integration of informal (scenario-based object-oriented modeling), semi-formal (GILO), and formal (CafeOBJ) approaches. And the intermediate design notation GILO helps realizing a smooth (seamless) integration of the informal and the formal notations. The resultant CafeOBJ descriptions are executable and thus adequate for validating functional aspects of the *model* descriptions. A typical use of CafeOBJ in the present approach would be a rapid prototyping tool for analysis *models* in an early stage of software development.

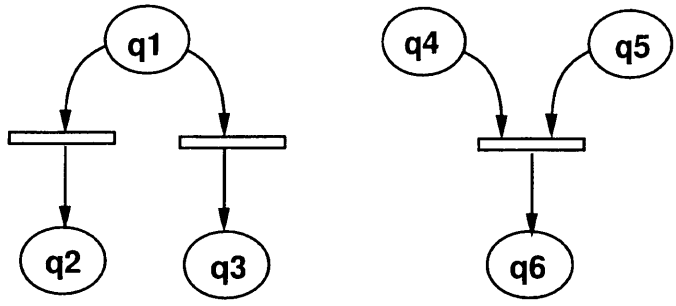


Figure 4.10: Fork and Join

Although it is not covered in the body of this chapter, multithreaded collaboration is easily encoded in CafeOBJ. A transition in the whole system is encoded in a rewriting rule that changes the marking data, where the marking is a multiset of state markers and each marker represents an execution snapshot of one thread of execution. Figure 4.10 shows a CPN version and the following CafeOBJ descriptions simulate the same transition. For the fork, marking on the RHS contains all the states that the execution forks to (q2 and q3). For the join, all the states that should be synchronized are specified explicitly in the marking on the LHS.

```
var REST : Marking .
trans on(O) <O:Machine | marking = (q1,REST)>
=> <O:Machine | marking = (q2,q3,REST)> .

ctrans on(O) <O:Machine | marking = (q4,q5,REST) >
=> <O:Machine | marking = (q6,REST) > .
```

The GILO-based modeling method to derive executable object-oriented algebraic specifications in CafeOBJ is unique compared with related works. Comparison is made in two areas: (1) scenario-based object-oriented modeling methods, and (2) modeling methods integrated with FDTs.

OOSE [65] and RDD [121] are two pioneers that proposed scenario-based object-oriented methodology. Booch and Rumbaugh's unified method also supports the scenario concept [15]. The interaction diagram of OOSE and the event trace diagram

of the unified method correspond to the collaboration of GILO. OOSE and RDD are diagram-based notations having less rigorous semantics.

In order to express collaboration, Fusion [23] uses the object interaction graph, which is basically the same as the event trace diagram of OMT. In addition, Fusion promotes the use of an operation model and a life-cycle model. The former corresponds to method behavior of the GILO class and the latter to the GILO collaboration. The life-cycle model uses a regular expression whose alphabet represents a set of events. The operation model offers guidelines for representing behavioral aspects of a method or an operation in terms of the pre- and post-conditions. Unfortunately the conditions are described informally in natural language. No formal semantics between the life-cycle and operation models is established.

Catalysis [30] is a new modeling method that focuses on the object interactions and formal description techniques. The central idea is to treat objects and actions equally, and thus to place interactions between objects from the start. Catalysis introduces a joint action, which is a series of related actions, as a common modeling tool for use-cases and collaborations, which is in accordance with the idea of GILO. As for expressing the pre- and post-conditions or other forms of behavior description, Catalysis uses semi-formal notations OCL of UML [135]. Although activities on formalizing OCL is underway [136], OCL of Catalysis itself is not based on rigorous semantics. Catalysis, however, has an important notion of refinement, which provides a systematic guideline to transform an abstract design artifact into concrete ones in a stepwise fashion. Unfortunately, GILO does not provide any guideline for the refinement.

The second area is on the integration of the informal object-oriented modeling and FDTs. Giovanni and Iachini [43] and Hall [43] use object-oriented modeling method as a guideline for finding *objects* in the analysis phase and then obtain descriptions in the Z notation. The structural aspect is a primary concern and does not have a concept of scenario. Further, descriptions in the Z notation is hardly mechanically analyzable.

Wirsing [124] proposes a formal object-oriented design based on Jacobson's OOSE [65] and in which a Maude-based formal object model is encoded in an algebraic specification language Spectrum, and later in CafeOBJ instead [125]. The emphasis is put on the importance of the stepwise refinement with discussions on the role of proof checking in the refinement process. Since the development process uses the interaction diagram (a scenario) as a guiding tool just for obtaining object specifications, scenario diagrams do not have rigorous semantics. On the other hand, collaboration of GILO has its translation into CafeOBJ and thus is rigorous.

Last, GILO still suffers from the problem that any general-purpose modeling method has. The problem is clearly summarized by Jackson [64] in that methods cannot be panaceas (medicines that cure all diseases) and that methods should be related to a particular class of problem and thus give a sharply focused help in reaching a solution. Chapter 6 deals with a CafeOBJ-based design method for a particular problem of the ODP trader, and discusses an approach to establishing the methodological principles of selecting and applying CafeOBJ in the software development process.

Chapter 5

Understanding the ODP Trader with CafeOBJ

5.1 Introduction

The advancement of distributed software technology demands the establishment of common services for distributed computing environments. Examples of the common services include naming services, trading services, and security services [86][133].

ODP (Open Distributed Processing), a joint effort of ISO and ITU-T, aims to provide a general architectural framework for distributed systems in a multi-vendor environment [99]. Their activities involve the use and integration of FDTs (Formal Description Techniques) into the ODP from the early stages of defining RM-ODP (Reference Model of Open Distributed Processing) [34]. As a concrete service following RM-ODP, the ODP trader has been defined [8][131].

The ODP trader is an important standard specification because it has recently become IS (the International Standard) and also it is technically aligned with the OMG specification for the trading object service [134]. The framework presents five separate viewpoints corresponding to different abstraction levels. The five viewpoints are enterprise, information, computational, engineering, and technology. The standard document uses the Z notation [110] to describe the information viewpoint specification and uses IDL [86] for documentation of the computational viewpoint specification, and it supplements explanations in a natural language (English).

Although the current specification provides valuable information on the ODP trading function from various viewpoints, it is too detailed when we want to know

the overall features or functionalities that the service provides. Since more than one technique is used to specify essentially one functionality (the trading function), there are some gaps between descriptions of various viewpoints. The Z specification and the IDL specification define one concept through the use of respective language constructs. Since each language has different underlying computational assumptions, the descriptions are not easy to compare to see how they are really related. Sometimes each specification employs a lot of *idioms* specific to the respective language, which makes the comparison difficult.

The specification would be much more accessible if we could write the two viewpoints in one multiparadigm specification language that has a consistent computational semantics. Since CafeOBJ is an algebraic specification language, it allows specifications to be written in a property-oriented specification style [119]. CafeOBJ can be used to represent different abstract levels for different aspects of the specificand while providing the same specification fragments for a set of the common concepts. Further, the rewriting machinery of CafeOBJ can be used for two different purposes when mechanically analyzing the specifications: specification execution and proof checking. By fully using the capabilities of CafeOBJ, we can obtain the specifications for the ODP trader that are clear, consistent and checked mechanically [94][95].

Section 5.2 gives an overview of the ODP trader that is based on the ITU-T recommendation document, and summarizes current activities on the application of FDT to the ODP trader. Section 5.3 and Section 5.4 present CafeOBJ descriptions of the information and computational viewpoints respectively. Section 5.5 presents discussions on the pros and cons of CafeOBJ-based approach to the ODP trader viewpoint specifications.

5.2 The ODP Trader Specification

5.2.1 Trading Function

Figure 5.1 shows the ODP trader and the participants in a trading scenario [8][131]. A service server (Exporter) exports a service offer. A service client (Importer) imports the service offer and then becomes a client of the service. Trader mediates between the two by using the exported service offers stored in its own repository which is ready for import requests.

Every service offer has a service type which has the interface type of the object

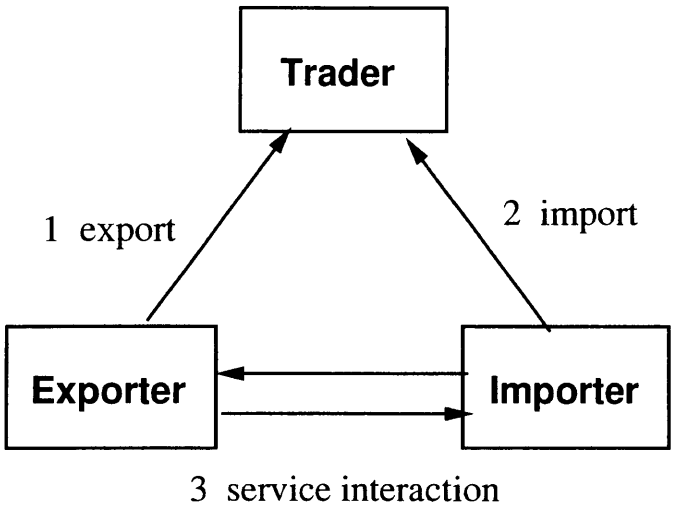


Figure 5.1: The ODP Trader and its Context

being advertised and a list of property definitions. A property definition consists of a property name, the type of property value, and a mode. The mode indicates whether the property value is mandatory, optional, or readonly. A service offer is a value which is consistent with the type and mode information in the property definitions of its service type.

The importer specifies a list of pairs of the property name and value for service offers that it tries to import. Then, the trader searches its repository to find service offers whose property values match the importer's request. The importer can further specify preference information to specify that the matched offers are sorted according to the preference rule. Last, the sorted offers are returned to the importer to be ready for initiating service interactions.

A concrete example might be a help in understanding a trading scenario. The example scenario, importing service offers of a printer service, is taken from [116]. Some of the properties that the offer for the printer service has are the location of the printer server, a resolution, color or black/white, the printer's name, and a length of the printer queue. Of these properties, the length of the printer queue is dynamic and its exact value is obtained at the time of the request.

The importer tries to obtain the service offers of the printer by a request that has a set of property values and a preference value. For example, the importer

specifies the location of a floor lower than the third in Building A, and true color properties in addition to a rule saying that a printer with a short queue is preferred. The trader searches its repository for offers that match the importer request. For values of properties that are not specified, the trader either uses default values that the trader has or just ignores them if the properties are optional. After the trader successfully finds a set of matched offers, it sorts the set according to the preference rule. Thus, the trader returns the list of offers for the color printers with the shortest printer queue first. The importer receives the list of printer offers, and initiates a printing service on the desired printer in the list.

The ODP trader defines a standard constraint language for specifying both the property and the preference. Further, the ODP trading function is a complex specification since it has the concept of service subtyping and federation between traders. The subtyping rule is used to find offers that, in a sense, partially match the request. The trader can be a member of a federated trader group that interworks to manage and handle service offers.

5.2.2 The Standard Document and FDTs

ITU-T has defined a reference model for open distributed processing RM-ODP [99], which introduces the concept of viewpoints for describing a system from various concerns. The viewpoints are enterprise (requirement capture and early design of distributed systems), information (conceptual design and information modeling), computational (software design and development), engineering (system design and development), and technology (technology identification, procurement and installation). Of the five viewpoints, the information and computational are the most important ones in view of the application of FDTs and the standardization activities. The others are intrinsically informal (enterprise) or implementation-dependent (engineering and technology).

The standard document of the ODP trading function [131] follows RM-ODP and it describes three viewpoints: enterprise, information and computational. In particular, the information viewpoint sees the trading system from the viewpoint of a centralized system. It uses the Z notation [110] to define the basic concepts, the trader state, and the set of the top-level operations that is visible from the outside. The computational viewpoint describes the functional behavior of the trading system, and provides a decomposition of the overall functionality into several components and their interactions. This viewpoint uses IDL [86] to describe the basic datatypes and the top-level operation interfaces, and it supplements all the behav-

ioral aspects in terms of natural language explanations.

Since the Z notation is a model-oriented formal specification language [110] that is based on axiomatic set theory (Zermelo set theory), the required mathematical background is not so advanced. And thus software engineers who have adequate training in mathematics do not find it difficult to understand Z specifications. Some of specification fragments, however, employ a lot of *idioms* specific to the Z notation. The description is almost a result of *hacking* at the worst. Further, it is hard to mechanically analyze specifications written in the full Z notation because of the richness of its background mathematics. For naive software engineers who are not familiar with mathematics, specifications written in an executable or operational manner are much more accessible.

Some literature discusses that the FDTs to be used in the standard specification documents such as the ODP trader should also be standardized and thus that Z, SDL and LOTOS are good candidates [16] [34]. However, these FDTs are based on research long before RM-ODP and the ODP trader appeared, and there have been many new advances in the FDT research since then. For example, the abstract data part of LOTOS is many-sorted while CafeOBJ has order-sorted algebra.[1] Moreover, the Z notation itself has not been approved to be IS (the International Standard) yet.[2] One may think that it is not necessary to stick to old FDTs. Instead, a new FDT with advanced features like CafeOBJ is sometimes better suited for specifying complex specifications such as the ODP trader.

Some researchers have proposed new FDTs for RM-ODP [12][17][31]. Concerns here are (1) consistency between the information and computational viewpoints and (2) the possibility of mechanical analysis for the specification descriptions. Most of the work has concentrated on the first aspect only. However, we see that mechanical analysis is valuable in allowing us to understand the specificand as well as providing consistent specifications. Although designing a new specification language for RM-ODP is valuable, it would also be worth investigating to study how existing formal specification languages are employed. It is because we can make use of existing techniques and tools for analyzing specifications mechanically.

Apart from the Z specification of the standard document, [33], [34] and [75] provide formal specifications for the ODP trader. Fischer et al. [34] is one early work applying FDTs to the specifications of the ODP application. They investigate Z, LOTOS and SDL in writing the specifications and conclude that no single FDT

[1] Actually, these advances have motivated a new addition to the LOTOS family, E-LOTOS [75].

[2] As of December 1999.

can specify the richness of the ODP trader. The work has had much impact on the current standard document [131]. Fischbeck et al. [33] employ a combination of IDL and SDL to describe the computational viewpoint specification. Their tool generates executable programs of either C++ or Java. Thus, functional aspects of the computational viewpoint specifications can be validated by executions. Since the approach is based on generating programs, the emphasis is on the relationship between the computational and engineering viewpoints. Lecero and Quemada [75] present the E-LOTOS specification for the computational viewpoint of the ODP trader. Their purpose is to show how new language constructs of E-LOTOS are used to specify ODP applications. Since the language belongs to the LOTOS family, the specification style is process-oriented (the ODP trader consists of a set of E-LOTOS processes). It presents specifications that are organized differently from those in the standard document.

5.3 The Information Viewpoint

The core part of the information viewpoint specification is descriptions written in the Z notation, which is based on axiomatic set theory (Zermelo set theory). Because of the nature of the theory, it is not always easy to translate Z specifications into CafeOBJ directly [128]. The Z notation allows higher-order specifications in that expressions can be evaluated to be types (sets), while CafeOBJ, being an algebraic specification language, is essentially first-order. Type (sort) and value are clearly distinct. Further, the Z notation can specify an infinite set by using either an existential quantifier or a set comprehension that employs predicates to characterize the elements of the defining set. Contrarily it is difficult to encode infinite sets in CafeOBJ. Therefore, it is necessary to give some *interpretation* to each specification fragment in regard to how the fragment is used in the rest of the specification. Since the specific translation will be discussed below with detailed explanation on each decision, we only give some general rules for the translation in Table 5.1.

5.3.1 Some Specification Fragments

In this section, we will show some specification fragments in the Z notation drawn from the standard document [131] and the CafeOBJ counterparts. Our goals are (1) to write executable specifications so as to help us understand the functionality of the ODP trader at an abstract level, and (2) to write modular specifications so

	Z Notation	CafeOBJ
1	given name	sort symbol
2	state schema	sort symbol, constructor function symbol
3	operation schema	function symbol, equational axiom
4	property part of schema	function symbol, equational axiom

Table 5.1: General Rules for Translation

as to make it clear the correspondence of the CafeOBJ modules with the elements in the standard document and at the same time to make it clear the role that each module has.

Basic Concepts

A given name introduces a new basic concept. *InterfaceSignatureType* and *Name* are examples of this. A free type definition specifies a set with elements in the defining set. The set *Mode* has four constants.

$$[InterfaceSignatureType,\ Name]$$

$$Mode ::= normal \mid readonly \mid mandatory \mid readonlymandatory$$

A given name is encoded to be a sort in CafeOBJ, and a free type is a sort with appropriate (constant) constructors.

```
mod! INTERFACE-SIGNATURE-TYPE {   [ InterfaceSignatureType ] }

mod! MODE {
  [ Mode ]
  signature {
    ops normal readonly mandatory  : -> Mode
    op readonlymandatory : -> Mode
  }
}
```

Some schema represents a basic concept that is an aggregate of known concepts. *ServiceType* has two named components, *signature* of type *InterfaceSignatureType* and *prop_defs* of type $Name \pfun (ValueType \times Mode)$.

ServiceType

$signature : InterfaceSignatureType$
$prop\_defs : Name \pfun (ValueType \times Mode)$

Since *ServiceType* can be considered as a structured data, its CafeOBJ specification employs a style similar to a record structure. The following module `SERVICE-TYPE` introduces a new sort `ServiceType` and one constructor which provides a record-like syntax, and two accessor functions. The `axioms` part gives definitions for the accessors by using equations. The module also imports two modules `INTERFACE-SIGNATURE-TYPE` and `PROPERTY-DEFINITIONS` that provide definitions for the symbols used.

```
mod! SERVICE-TYPE {
  [ ServiceType ]
  protecting (INTERFACE-SIGNATURE-TYPE)
  protecting (PROPERTY-DEFINITIONS)
  signature {
   op [signature=_, prop-defs=_] :
      InterfaceSignatureType PropertyDefinitions -> ServiceType
   op _.signature  : ServiceType -> InterfaceSignatureType
   op _.prop-defs : ServiceType -> PropertyDefinitions
  }
  axioms {
    var I : InterfaceSignatureType    var P : PropertyDefinitions

    eq ([signature=(I), prop-defs=(P)]).signature = I .
    eq ([signature=(I), prop-defs=(P)]).prop-defs = P .
  }
}
```

Basic Library for Executable Specifications

Any executable specification in CafeOBJ should have an initial algebra model [29] [38]. A rule of thumb is that the definition of basic data structures is to fol-

low recursive structures. They are based on recursively defined List (see Section 3.2.1) and they add appropriate utility functions to simulate other high-level functionalities such as the set-like collection of data. The parameterized module `COLLECTION[X :: TRIV]` is the one we provide. By using this library module, we have an executable CafeOBJ module for a type $\mathbb{P}$ *ElementData* in the Z notation. The module `SERVICE-OFFER-S` is such an example, which is $\mathbb{P}$ *ServiceOffer* in the Z specification.

```
mod! SERVICE-OFFER-S {
  protecting (COLLECTION[SERVICE-OFFER]
    *{ sort Collection -> ServiceOffers })
}
```

The module `COLLECTION` introduces a sort `Collection` to represent (homogeneous) collection of some values. In defining the module `SERVICE-OFFER-S` with instantiating `COLLECTION`, we have *renamed* `Collection` to be `ServiceOffers` by using the *view* mechanism of CafeOBJ [29].

In order to construct a basic library for the original Z specification having function types ($\pfun$), we have introduced another parameterized module `ENVIRONMENT`. This follows the observation that the function types in Z are basically power sets of some relation, although not strictly so.[3]

$$X \pfun Y \Rightarrow \mathbb{P}(X \times Y)$$

The `ENVIRONMENT[X :: TRIV, Y :: TRIV]` employs `COLLECTION` and `2TUPLE` [46] as its internal representation.

State Schema

The schema *TradingSystem* is the main state schema and it maintains the global state space of a series of federated traders. The schema defines data structure representing the state space with a set of well-formedness conditions as its properties. The schema shows that the state space consists of five components and that four predicates be asserted for the state space to be well-formed. The last line starting $\forall$ defines equality of *ServiceOffer* instances.

[3]The definition of the partial function in terms of the relationship is given in [110].

TradingSystem

$$
\begin{array}{l}
offers : \mathbb{P}\ ServiceOffer \\
nodes : \mathbb{P}\ Node \\
partition : ServiceOffer \twoheadrightarrow Node \\
edges : Node \leftrightarrow Node \\
edge\_properties : (Node \times Node) \twoheadrightarrow \mathbb{P}\ Property \\
\hline
\mathrm{dom}\ partition = offers \\
\mathrm{ran}\ partition \subseteq nodes \\
\mathrm{dom}\ edges \cup \mathrm{ran}\ edges \subseteq nodes \\
\mathrm{dom}\ edge\_properties = edges \\
\forall p, q : ServiceOffer \bullet p.service\_offer\_identifier = q.service\_offer\_identifier \\
\quad \Leftrightarrow p = q
\end{array}
$$

The declaration part is directly translated to a record style structure, which is in the same manner as the *ServiceType*. We defined `TRADING-SYSTEM` module for the *TradingSystem*.

When we stand at the specification execution side, one way of using the property part of the *TradingSystem* is just to check the well-formedness of the *TradingSystem* record structure after invocation of an operation, say an *export*. In other words, the property part is interpreted as a boolean-valued function that accepts a *TradingSystem* record instance as its argument.

Here is a CafeOBJ module definition in accordance with this idea. The function `trading-system-axiom` is a conjunction of the four predicates of the Z specification. In addition, all the symbols such as `dom-partition` are defined by using either `COLLECTION` or `ENVIRONMENT`. Since the symbols can, in principle, be used in a variety of modules, they should be defined in one place. The `TRADING-SYSTEM-LIBRARY` imported in the `TRADING-SYSTEM-AXIOMS` has all the definitions.

```
mod! TRADING-SYSTEM-AXIOMS {
  protecting (TRADING-SYSTEM)
  protecting (TRADING-SYSTEM-LIBRARY)
  signature {
    op trading-system-axiom : TradingSystem -> Bool
    op service-offer-equality : ServiceOffer ServiceOffer -> Bool
  }
  axioms {
  var T : TradingSystem   vars V1 V2 : ServiceOffer
```

```
  eq trading-system-axiom(T)
  =     ((dom-partition((T).partition)) equal-to ((T).offers))
    and ((ran-partition((T).partition)) is-subsumed-by ((T).nodes))
    and ((all-nodes-of ((T).edges)) is-subsumed-by ((T).nodes))
    and ((dom-edge-prop((T).edge-properties)) equal-to ((T).edges))  .

  eq service-offer-equality(V1,V2)  =  ((V1).offer-id == (V2).offer-id) .
  }
}
```

Operation Schema

The main trading functions are implemented as operations on the state schema *TradingSystem*. *Export* is an example function that registers a new service offer. The offer is subsequently stored in a repository the trader maintains, and it is actually stored in some components of the schema *TradingSystem*.

The following schema *ExportOK* defines a part of the definition of the export function.[4] It accepts two input parameters and returns a value of *ServiceOfferIdentifier*, and it also leaves some changes in the *TradingSystem* state space. Further, the first two predicates are preconditions and the rest are postconditions. *Offers* and *partitions* are updated accordingly while the others are left unchanged.

ExportOK

$\Delta TradingSystem$
$new\_offer? : ServiceOffer$
$node? : Node$
$service\_offer\_identifier! : ServiceOfferIdentifier$

$\forall s : offers \bullet$
$\quad s.service\_offer\_identifier \neq new\_offer?.service\_offer\_identifier$
$node? \in nodes$
$offers' = offers \cup \{new\_offer?\}$
$partition' = partition \cup \{new\_offer? \mapsto node?\}$
$service\_offer\_identifier! = new\_offer?.service\_offer\_identifier$
$nodes' = nodes$
$edge\_properties' = edge\_properties$
$edges' = edges$

[4] A case where the precondition becomes false is also documented [131].

The fact that the properties can be divided into preconditions and postconditions is the basis for writing the CafeOBJ specification.

Since the *ExportOK* and other operation schemata not only return some specific values, but also leave changes in the state space, we could not model the operation in a purely functional manner. We introduce the module `BASIC-STATE`, which has functionalities whereby the operation can return values and update the state space at the same time. The function to represent a top-level operation will take the following form;

$$\texttt{TradingSystem} \times \text{InputArgs} \rightarrow \text{ReturnValue} \times \texttt{TradingSystem} \ .$$

The sort `ValueState` is defined as a tuple of ReturnValue and `TradingSystem`.

The module `EXPORT` defines the top-level operation `export`, which uses two auxiliary functions and returns a new `ValueState` value; `value` computes a return value and `state` gives a new state space value.

```
mod! EXPORT {
  protecting (BASIC-STATE)
  protecting (EXPORT-BEHAVIOR)
  signature {
    op export : TradingSystem ServiceOffer Node -> ValueState
  }
  axioms {
    var S : TradingSystem
    var O : ServiceOffer     var N : Node
  eq export(S,O,N)  = new-state(value(S,O,N), state(S,O,N)) .
  }
}
```

The module `EXPORT-BEHAVIOR` defines behavior for the two functions. Since the original Z specification for the *Export* schema handles an exceptional case as well as a normal one, the behavior of the two functions can be defined in terms of a set of conditional equations where the condition corresponds to the precondition of the *ExportOK* schema.

```
mod! EXPORT-BEHAVIOR {
  protecting (TRADING-SYSTEM)
```

```
  signature {
    op value    : TradingSystem ServiceOffer Node -> ServiceOfferIdentifier
    op state    : TradingSystem ServiceOffer Node -> TradingSystem
    op pre-cond : TradingSystem ServiceOffer Node -> Bool
  }
  axioms {
    var S : TradingSystem     var O : ServiceOffer     var N : Node

  ceq value(S,O,N) = (O).offer-id if pre-cond(S,O,N) .
  ceq value(S,O,N) = void         if not pre-cond(S,O,N) .

  ceq state(S,O,N) = [offers=(add((S).offers,O)), nodes=((S).nodes),
                      partition=(add((S).partition,O,N)), edges=((S).edges),
                      edge-properties=((S).edge-properties)]
                                          if pre-cond(S,O,N) .
  ceq state(S,O,N) = S                    if not pre-cond(S,O,N) .

  eq pre-cond(S,O,N) = ((N) is-member-of ((S).nodes))
                       and (new-id ((S).offers, (O).offer-id)) .
  }
}
```

For example, the member `offers` is updated to be `add((S).offers,O)` which is a CafeOBJ representation of the following Z specification fragment in the *ExportOK* schema:

$$offers' = offers \cup \{new\_offer?\}.$$

Further, in order for the above specification to be executable, `add((S).offers,O)` should be evaluated to have a "natural" normal form of the specified sort `ServiceOffers`. This requires that the module `SERVICE-OFFER-S` defining the sort `ServiceOffers` should provide executable data structures, which we have done.

Relation Schema as Function

The Z specification includes a schema which introduces a relation on some particular set(s). The relation *is_subtype_of* is one such example which defines the subtype relationship between two *ServiceType* instances.

$$
\begin{array}{|l}
\_\ is\_subtype\_of\ \_ : ServiceType \leftrightarrow ServiceType \\
\hline
\forall\, a, b : ServiceType \bullet\ b\ is\_subtype\_of\ a \Leftrightarrow \\
\quad b.signature\ is\_sig\_subtype\_of\ a.signature \\
\quad \wedge\ \mathrm{dom}\, a.prop\_defs \subseteq \mathrm{dom}\, b.prop\_defs \\
\quad \wedge\ (\forall\, n : \mathrm{dom}\, a.prop\_defs \bullet \\
\qquad first(a.prop\_defs\ n)\ is\_value\_supertype\_of\ first(b.prop\_defs\ n) \wedge \\
\qquad second(a.prop\_defs\ n)\ is\_mode\_supertype\_of\ second(b.prop\_defs\ n)
\end{array}
$$

In order to make the relation *is_subtype_of* executable, we model it as a function to ensure that the two arguments of the relation *is_subtype_of* satisfy the subtype relationship. According to the Z specification, the relation is further decomposed into *is_sig_subtype_of*, $\subseteq$ and the $\forall$ part that also checks the two conditions by using *is_value_supertype_of* and *is_mode_supertype_of*.

```
mod! SUBTYPING-RULE {
  protecting (SERVICE-TYPE)
  protecting (SIGNATURE-SUBTYPING)
  protecting (VALUE-MODE-SUPERTYPE-FLATTEN[PRED2SUBTYPING]
              *{ op andalso -> check-vm })
  signature {
    op _is-subtype-of_ : ServiceType ServiceType -> Bool
  }
  axioms {
    vars S1 S2 : ServiceType

    eq (S1) is-subtype-of (S2)
    =     ((S1).signature) is-sig-subtype-of ((S2).signature)
      and ((names((S1).prop-defs)) is-subsumed-by (names((S2).prop-defs)))
      and check-vm(names((S1).prop-defs), (S1).prop-defs, (S2).prop-defs) .
  }
}
```

Since the last condition concerns $\forall\, n : \mathrm{dom}\, a.prop\_defs$, we will model it as a function with n as its input parameter. More concretely, a new function **check-vm** is defined to have three arguments, the first runs through n in $\mathrm{dom}\, a.prop\_defs$, and the second and the third are the *prop_defs* components of the *ServiceType* instances. Since enumeration over n in $\mathrm{dom}\, a.prop\_defs$ requires some auxiliary

functions in CafeOBJ, we factor the definition of `check-vm` into another module `VALUE-MODE-SUPERTYPE-FLATTEN` that in turn uses a parameterized module to simulate the higher-order specification style. The following fragment illustrates the body of the function `check-vm`.

```
var N : Name  var L : Names
vars P1 P2 : PropertyDefinitions

eq andalso(nil,P1,P2) = true .
eq andalso((N L),P1,P2)
=  if p(N,P1,P2) then andalso(L,P1,P2) else false fi .
```

The function `andalso` is basically a procedural implementation of the following Z specification fragment.

$$\forall n : \mathrm{dom}\, a.prop\_defs \bullet p(n, ad(n), bd(n))$$
$$\Rightarrow \quad \wedge_{(n \in \mathrm{dom}\, a.prop\_defs)}\ p(n, ad(n), bd(n))$$

The predicate p (or `p`) is actually given by `pred-vm` below.

```
var N : Name   vars W1 W2 : ValueMode  vars P1 P2 : PropertyDefinitions

eq pred-vm (N,P1,P2)  =  pvm(lookup(P1,N), lookup(P2,N)) .
eq pvm(W1,W2)  = (get-value-type(W1) is-value-supertype-of get-value-type(W2))
                 and (get-mode(W1) is-mode-supertype-of get-mode(W2)) .
```

5.3.2 Test Execution Trace

Below we show an example reduction for the case of `EXPORT`. After loading all the necessary CafeOBJ specifications, we can *execute* the specification with appropriate input terms (test data). [5]

```
TEST> let a1 = export((t), new-s, node) .
TEST> let a2 = export((t), new-s, node2) .
```

[5] We used cafeobj (1.4b5), which can be obtained from the following URL: http://www.ldl.jaist.ac.jp/cafeobj/index.html.en

```
TEST> red 1*(a1) .
oid3 : ServiceOfferIdentifier

TEST> red trading-system-axiom(2*(a1)) .
true : Bool

TEST> red 1*(a2) .
void : ReturnValue

TEST> red trading-system-axiom(2*(a2)) .
true : Bool
```

The construct `let` temporarily binds its right-hand side term to the left-hand side identifier. The label `t` refers to a `TradingSystem` value. The label `new-s` is a `ServiceOffer` value defined as a term similar to record structure in the module `SERVICE-OFFER`. The labels `node` and `node2` are `Node` values defined in the module `NODE`. Since the value of `export` is a tuple (see Section4.1.4), we can obtain the return value of the `export` function by `1*(a1)` and the resultant new `TradingSystem` state value by `2*(a1)` where `1*` and `2*` access the first and second element of the tuple respectively [46].

In the above session, `a1` is a case of success in *Export* operation while `a2` corresponds to a case where the precondition of *Export* is violated and thus leaves no change in the *TradingSystem*. For the first case, the return value is a new `ServiceOfferIdentifier` value `oid3`, which is generated by `export`. The second case returns `void` which means that no significant value is returned. In both cases, `trading-system-axiom` returns `true` as expected.

Last, the CAFE environment also provides a compiler that translates terms and rewriting rules into virtual machine codes in order to realize fast execution [40]. Typically, we can obtain 3 to 10 times performance gain by using the compiler.

5.3.3 Search and Select

The Z specification of the standard document defines *Search* and *Select* schemata for importing. The basic idea is first to use *Search* for collecting all the service offers that match both the importer's request and the conditions that the trader has, and second to invoke *Select* for sorting the offers according to the importer's preference. As shown below, the original Z specification is very declarative and hard to understand its operational meaning at a glance.

$$MatchingCriteria == \mathbb{P}\ ServiceOffer$$
$$ScopeCriteria == \mathbb{P}\ ServiceOffer$$

TradingSystemConstraints

$$trader\_matching : MatchingCriteria$$
$$trader\_scope : ScopeCriteria$$
$$\vdots$$

SearchOK

$$\Xi TradingSystem$$
$$\Xi TradingSystemConstraints$$
$$SearchRequest?$$
$$starting\_point? : Node$$
$$search\_result! : \mathbb{P}\ ServiceOffer$$

$$starting\_point? \in nodes$$
$$partition(\mid search\_result! \mid) \subseteq \{x : Node \mid (starting\_point?, x) \in edges^{+}\}$$
$$search\_result! \subseteq importer\_matching? \cap trader\_matching \cap importer\_scope? \cap trader\_scope$$

SearchOK is meant to return an appropriate set of *ServiceOffer*s bound to the variable *search_result*!. The appropriateness is expressed in terms of the logical condition given to the variable. The returned *ServiceOffer*s is a subset of intersection of the four *ServiceOffer* sets. Each set is a subset of the managed *ServiceOffer*s that matches a particular partitioning condition. For example *importer_matching*? is for those that match the condition in the importer request, while *trader_matching* refers to those that reflect the condition imposed by the trader. Thus, taking the intersection produces a set of *ServiceOffer*s that match all the conditions.

However, since all four sets are defined as a value of $\mathbb{P}$ *ServiceOffer* and are not mentioned further, it is not easy to grasp what the specification really means. This is partly because the modeling method that *ServiceOffer*s are partitioned *a priori* has a large gap with an intuitive specification such as one for the computational viewpoint. The latter viewpoint uses a constraint language to specify such conditions

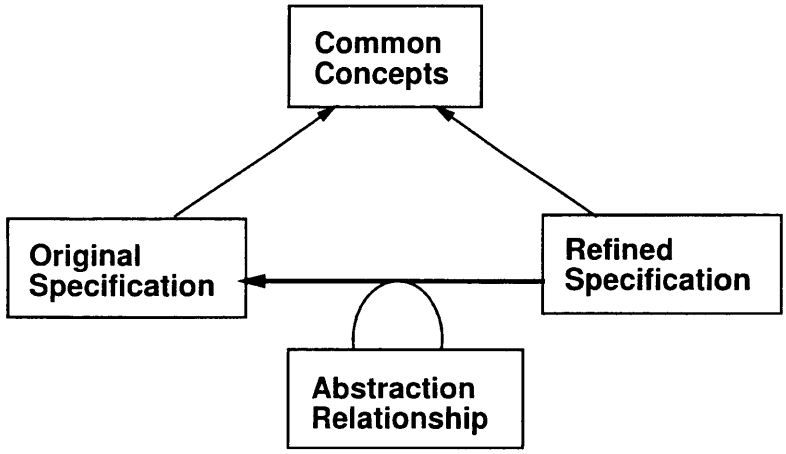


Figure 5.2: Refinement

[131] explicitly. In summary, the original Z specification is abstract and thus further design decision is necessary to have an executable specification. Since introducing such design decisions is beyond the scope of this chapter, our decision was not to provide executable specifications.[6]

Actually we study two styles of CafeOBJ specifications for this part. Our decision for the first approach is only to provide signatures (declarations of symbols) and minimum axioms saying that some relationships exist between some of the symbols. In a word, the specification is a direct syntactical transcription of the Z specification. However, the CafeOBJ specification can be mechanically checked in view of syntax and sort (type), while the Z specification is hard to mechanically analyze. Namely, the specification fragment together with the rest of the CafeOBJ descriptions can be shown consistent in view of sort. Our second approach is to study how CAFE/CafeOBJ provides mechanical support for specification refinement, which we will discuss below.

As the CafeOBJ specifications, we define two versions for *SearchOK* as shown in Figure 5.2. One is what is called the original specification, and the other is a refined one that employs modeling with a hypothetical retrieval language. The retrieval language can be regarded as an abstraction of the constraint language

[6]Chapter 6 discusses some design decisions.

of the computational viewpoint [131]. In addition, we introduce an abstraction relationship which maps elements in the refined specification to elements in the original if possible. The idea has commonly been used in verifying refinement of the Z specifications [110].

Here, we will show that the search result expressed as `search(T,X,S)` is a subset of the intersection of the two predefined sets.[7] The relationship is shown as a part of the original specification.

```
var T : TradingSystem  var S : SearchRequest
var X : TradingSystemConstraints

eq search(T,X,S) is-subset-of
    ((X).trader-matching cap (S).importer-matching) = true .
```

The refined specification below indicates how `search` is obtained in terms of executing the search procedure which employs the hypothetical retrieval language. The condition that the resulting search offers should meet the requirements of both *trader-matching* and *importer-matching* is expressed as a conjunction of the two conditions.

```
var T : TradingSystem  var S : BSearchRequest
var X : BTradingSystemConstraints

eq search(T,X,S)
= exec((T).offers, (S).bimporter-type,
       ((X).btrader-matching and (S).bimporter-matching)) .
```

We also have a set of abstraction functions which map elements in the refined specification to their counterparts in the original specification, though the detailed equations are omitted.

```
op abs : BSearchRequest -> SearchRequest
op abs : BTradingSystemConstraints -> TradingSystemConstraints
```

With appropriate lemmas, we can mechanically verify the relationship for the terms in the refined specification. The search result is a subset of the intersection of the two specified sets.

[7]For simplicity, we dropped the condition on the scoping ($importer\_scope? \cap trader\_scope$) of the Z specification.

```
CafeOBJ> start abs(search(T*,XB*,SB*))
 is-subset-of ((XA*).trader-matching cap (SA*).importer-matching) .
CafeOBJ> apply red at term .
result true : Bool
```

The identifiers ended with * are all constants of appropriate sort which act as universally quantified variables.[8] Actually, T* denotes a TraderSystem. SB* and XB* are BSearchRequest and BTradingSystemConstraint respectively, while SA* and XA* are the counterparts of the *original* specification in Figure 5.2.

In addition to those relating to some general properties of set, lemmas include definitions for the hypothetical retrieval language. In particular, the equation for exec is the basis of mechanical checking above because it reflects the compositionality of retrieval conditions.

```
mod* B-RETRIEVAL-LANGUAGE {
  [ RetrievalLanguage ]
  protecting (A-SERVICE-OFFER-S)
  signature {
    op _and_ : RetrievalLanguage RetrievalLanguage
                -> RetrievalLanguage [comm]
    op exec : ServiceOffers ServiceType RetrievalLanguage
                -> ServiceOffers
  }
  axioms {
    var P : ServiceOffers
    var T : ServiceType  var O : ServiceOffer
    vars L L1 L2 : RetrievalLanguage

  ceq ((O).service-type) is-subtype-of T = true  if member(O,exec(P,T,L)) .
  eq exec(P,T,(L1 and L2)) = exec(P,T,L1) cap exec(P,T,L2) .
  }
}
```

Here, in the equation for the is-subtype-of relation, the conditional part has two variables (P,L) not appeared in the LHS of the equation, which is not allowed in *executable* specifications in CafeOBJ. However, the purpose of the B-RETRIEVAL-LANGUAGE module is just to provide a *relation* on the symbols that is not meant for executability.

[8] Using a constant as a representative value for an universally quantified variable follows the Theorem of Constants [48].

	Category	Z Notation	CafeOBJ	
			total	direct
1	*Basic Concepts*	13	23	13
2	*State Schema*	9	43	15
3	*Main API*	31	26	10
4	*Library*	–	5	0
5	(total)	53	97	38

Table 5.2: Some Metrics

5.3.4 Summary

The CafeOBJ specification we have written consists of 97 modules, and the number of equations is 290.[9] Table 5.2 summarizes some metrics of the specification. The column *Z Notation* shows the number of Z specification components (the standard document [131]) while the column *CafeOBJ* shows the number of CafeOBJ modules. The number of CafeOBJ modules that have direct correspondence with the Z counterpart is also shown. It depicts how many of the CafeOBJ modules are traceable from the Z specification, and hints how many are necessary for obtaining executable specifications.

The row *Basic Concepts* is for primitive concepts defined by using given names, free types, and abbreviation definition (==) in the Z specifications. The row *State Schema* is for structured data, relationship, and the main state schema *TradingSystem* defined by using schema in the Z specifications. The row *Main API* is for operations visible outside, actually operation schema. The row *Library* is for the common library modules that implement executable specifications and thus only for CafeOBJ exist.

In Table 5.2, we can see about 40% of the CafeOBJ modules are directly traceable from the Z counterparts. A primary reason that the CafeOBJ specification is larger than the original Z specification follows from the fact that we use a property-oriented specification style for CafeOBJ while a model-oriented style is used in the Z notation. In the property-oriented style, we have to provide all the necessary definitions [119]. On the contrary, we can assume or use predefined set of primitives in the Z notation. We can say that some of the CafeOBJ modules constitute the

[9]For the specification of *Search* and *Select*, the number includes the first approach briefly mentioned in Section 5.3.3.

"model" that provides appropriate definitions which are considered to be equivalent to the predefined definitions of the Z notation, although the "model" is quite different since our aim is to have executability. Further, detailed descriptions are necessary to have executable specifications (see Section 4.1.2). This also makes the CafeOBJ specification large.

As the most of *Basic Concepts* is just introducing set in the Z notation or sort in the CafeOBJ, about 60 % (13 out of 23) of the modules can be directly traced from the Z specification fragments. The main body of the rest of the modules is necessary for calculating membership of data in a certain type. In the Z notation, membership is very easy to state. Since a type of value is defined in terms of a set in the Z notation, such membership is expressed in terms of $\in$ predicate.

$$ValueType == \mathbb{P}\ Value$$
$$a : ValueType \land b : Value \land b \in a$$

For the CafeOBJ specification, both type and data are ordinary terms belonging to certain sorts, the CafeOBJ version requires some auxiliary functions to calculate the membership.

Most of *hacking* for CafeOBJ to have executability is in *State Schema* because functions and/or relationships are fallen in this category and thus lots of auxiliary modules are necessary. For *Main API*, the Z specification has three or five schemata for each operation, while in the CafeOBJ specification we write one module for one operation and other auxiliary modules as necessary. The traceability of the top-level operations is good.

Section 5.3.3 has seen that CAFE/CafeOBJ provides adequate supports for mechanical verification of specification refinement. The verification activity usually involves using lemmas, which often requires further proof sessions. As it is always the case, finding appropriate lemmas is a difficult task that human should take care of. However, we think that studying one specification from various concerns helps us understand its functionality. Writing specifications at two different abstraction levels and establishing refinement relationships between them is one such approach when writing executable specifications is not adequate for various reasons.

	IDL	CafeOBJ
1	datatype	abstract datatypes
2	exception	abstract datatypes
3	interface	concurrent object
4	operation	method (concurrent object)
5	attribute	method (concurrent object)

Table 5.3: General Rules for Translation

5.4 The Computational Viewpoint

The computational viewpoint specifies the trading function as a set of operations in terms of the object-oriented paradigm [99][131]. The IDL module `CosTrading` has all the operations which are subsequently grouped into eleven IDL interfaces. An IDL interface fits the object-oriented paradigm well, since it packs attributes, operations and exceptions, accompanied with related data definitions such as struct, union, enum, in a modular manner. Therefore, our strategy for writing the CafeOBJ specification is to represent each IDL interface as a Maude concurrent object (Section 3.3).

Since the specific translation will be discussed below with detailed explanation on each decision, we give some general rules for the translation in Table 5.3.

5.4.1 IDL Datatypes in CafeOBJ

As was the case for the information viewpoint, we will show concrete examples from the standard document to explain our idea. First, we present IDL descriptions and CafeOBJ counterparts for definitions of various data structures.

Typedef and Struct

IDL `typedef` introduces a new type name, but it is just an alias of another predefined type. IDL `struct` introduces a named structured type that has more than one member of the types already defined.

```
typedef Istring Preference;
```

```
  struct OfferInfo {
    Object reference;
    ServiceTypeName type;
    PropertySeq properties;
  };
```

Since aliasing is a programming level concept, we explicitly introduce a new sort. The module PREFERENCE defines a sort Preference. In order to represent structured data, the module OFFER-INFO needs to have a constructor and appropriate accessor(s), where the module ROOT provides the sort OId.

```
mod! PREFERENCE { [ Preference ] }

mod! OFFER-INFO {
  [ OfferInfo ]
  protecting (ROOT)
  protecting (NAME)
  protecting (PROPERTY-VALUES)

  signature {
    op OfferInfo : OId Name PropertyValues -> OfferInfo
    op _.reference : OfferInfo -> OId
    op _.type : OfferInfo -> Name
    op _.properties : OfferInfo -> PropertyValues
  }
  axioms {
    var O : OId
    var N : Name
    var P : PropertyValues

    eq (OfferInfo(O,N,P)).reference  = O .
    eq (OfferInfo(O,N,P)).type       = N .
    eq (OfferInfo(O,N,P)).properties = P .
  }
}
```

Enum and Union

IDL enum introduces constant symbols belonging to a user-defined datatype. The type HowManyProps has three constants. The example below also shows a way

of using HowManyProps together with IDL union SpecifiedProps. A data of SpecifiedProps has a PropertyNameSeq data only when the switch tag is equal to some of HowManyProps, and it does not have component data for the switch tag other than some.

```
enum HowManyProps { none, some, all };

union SpecifiedProps switch ( HowManyProps ) {
  case some: PropertyNameSeq prop_name;
}
```

Since enum and union are often used together as the above example illustrates, we put everything in one module HOW-MANY-PROPS. The encoding is somewhat complicated. We first define an accessor function label which reads out the switch tag that the current data has. The accessor _.some returns the data of sort Names only when the switch tag is the same.

In defining the module, we identified that PropertyNameSeq is basically the same as Names which is defined as one of the basic concepts for the information viewpoint. Thus, this example also shows that the same module, NAMES, can be used in both viewpoints, which clarifies the correspondence between the two viewpoints.

```
mod! HOW-MANY-PROPS {
  [ HowManyProps, HowManyProps', HowManyProps'', SpecifiedProps ]
  protecting (NAMES)
  [ HowManyProps' < HowManyProps, HowManyProps'' < HowManyProps ]
  signature {
   op SpecifiedProps : HowManyProps' Names -> SpecifiedProps
   op SpecifiedProps : HowManyProps'' -> SpecifiedProps
   op _.some : SpecifiedProps -> Names
   op label  : SpecifiedProps -> HowManyProps
   op some   : -> HowManyProps'
   ops none all  : -> HowManyProps''
  }
  axioms {
   var H : HowManyProps
   var N : Names

   eq (SpecifiedProps(some,N)).some = N .
   eq label(SpecifiedProps(H,N)) = H .
```

```
    eq label(SpecifiedProps(H)) = H .
  }
}
```

In the definition of the HOW-MANY-PROPS module, we have to take into account the fact that HowManyProps has three constants only and that some shows distinctive behavior in that only some takes a parameter. Since representing such *conditions* compactly in CafeOBJ (algebraic specification languages in general) is difficult, our approach is only to *simulate* the facts and not to pursue a complete mathematically elegant answer. The approach needs to introduce auxiliary sorts and establish subsort relations between them.

```
HowManyProps'  < HowManyProps
HowManyProps'' < HowManyProps
```

The HowManyProps is a main sort representing the set introduced by the enum declaration. The HowManyProps' has a unique constant some, while the HowManyProps'' has two (none and all). The accessor .some is defined only for the case that the first parameter of the constructor SpecifiedProps is some. This implies that .some causes an error if applied to other situations.

Exceptions

IDL has the concept of exceptions, which is invoked when an operation detects some anomalies. IDL exception can take parameters that will be of help in inspecting the reason.

```
exception InvalidObjectRef {
  Object ref;
};
```

Since an exception has parameter(s), it can be modeled similar to the IDL struct. However, we have to take into account the computational aspects of the exception that it is invoked from operations and also it is subsequently consumed. We make the sort Exception to be a subsort of Message (Section 3.3.2) in order that a Configuration data can take the exception to be its constituents. Then, an object can consume appropriate exceptions by regarding them as messages of a special kind.

```
mod! INVALID-OBJECT-REF {
  extending (MESSAGE)
  protecting (ROOT)
  [ InvalidObjectRef < Exception ]
  signature {
    op InvalidObjectRef : OId -> InvalidObjectRef
    op _.ref : InvalidObjectRef -> OId
  }
  axioms {
    var O : OId

    eq (InvalidObjectRef(O)).ref = O .
  }
}
```

5.4.2 IDL Interfaces in CafeOBJ

An IDL interface definition can be considered to introduce an object. Attributes and operations are primary components of IDL interfaces.

Attributes

Some IDL interfaces have `readonly attributes`. `TraderComponents` has several attributes, each having a reference to an object interface.

```
interface TraderComponents {
  readonly attribute Lookup lookup_if;
  readonly attribute Register register_if;
    ...
}
```

Since accessing attribute values involves some computation, it is natural to model the interface in terms of a concurrent object. The module `TRADER-COMPONENTS` defines one such object that accepts messages to access attribute values.

```
mod! TRADER-COMPONENTS[X :: TH-TRADER-COMP-AID, Y :: TH-TRADER-COMP-MSG] {
  extending (ROOT)
  [ TraderComponentsTerm < ObjectTerm ]
  [ CIdTraderComponents < CId ]
```

```
  signature {
    op <(_:_)|_> : OId CIdTraderComponents Attributes -> TraderComponentsTerm
    op TraderComponents : -> CIdTraderComponents
  }
  axioms {
    vars O L R : OId
    var REST : Attributes

    trans lookup-if(O,R) <(O : TraderComponents)|(lookup-if = L), (REST)>
    =>  <(O : TraderComponents)|(lookup-if = L), (REST)>  return(R, L) .
    ...
}
```

Operations

IDL `interface` specifies *interface signature* of operation only that is basically an invocation pattern at the programming level concept. Below is an example IDL interface `Lookup` that provides an operation `query`. The operation `query` corresponds to the *Search* and *Select* schemata of the information viewpoint, and implements the *importing* function.

```
interface Lookup : TraderComponents, SupportAttributes, ImportAttributes {
  ...

  void query {
    in ServiceTypeName type,
    in Constraint constr,
    in Preference pref,
    in PolicySeq policies,
    in SpecifiedProps desired_props,
    in unsigned long how_many,
    out OfferSeq offers,
    out OfferIterator offer_itr,
    out PolicyNameSeq limits_applied
  } raises {
    IllegalServiceType,
    ...
  };
```

```
  ...
}
```

IDL `interface` is modeled in terms of a CafeOBJ module that defines a concurrent object. However, the resultant concurrent object does not have attributes because IDL `interface` defines its interface signature only. It requires further design decisions if the object is defined so as to have appropriate attributes (Chapter 6 discusses this issue by refining and elaborating design of the trader server.).

The module imports all the modules that provide the necessary definitions for structs, exceptions and others, and then it has the body of the object definition. The `query` operation is then a method that is implemented as a family of `ctrans` rules. Each rule corresponds to either a normal execution flow or exceptional case. An exceptional case generates an appropriate IDL exception.

```
mod! LOOKUP-TEMPLATE[Y :: TH-LOOKUP-MSG] {
  extending (ROOT)
  protecting (LOOKUP-LIBRARY)
  [ LookupTerm < ObjectTerm ]
  [ CIdLookup < CId ]
  signature {
    op <(_:_)|_> : OId CIdLookup Attributes -> LookupTerm
    op Lookup : -> CIdLookup
  }
  axioms {
  vars O R : OId
  var REST : Attributes
  var N : Name
  var C : Constraint
  var P : Policies
  var D : SpecifiedProps
  var I : Nat

  ctrans query(O,N,C,P,D,I,R) <(O : Lookup)| (REST)>
  =>  <(O : Lookup)| (REST)>
      return(R, outArgs(R,lookup(N,C,P,D))) if normal(N,C,P,D,I) .

  ctrans query(O,N,C,P,D,I,R) <(O : Lookup)| (REST)>
  =>  <(O : Lookup)| (REST)>
      IllegalServiceType(N) if serviceTypeError(N,C,P,D,I) .
```

```
  ...
  }
}
```

The module LOOKUP-LIBRARY may define all the symbols appeared in the module LOOKUP-TEMPLATE. The definitions may have **signature** only because giving appropriate interpretation to each symbol requires further design decisions, which is not written in the standard document.

```
mod! LOOKUP-LIBRARY {
  protecting (ILLEGAL-SERVICE-TYPE)
  ...

  signature {
    op normal : Name Constraint Policies SpecifiedProps Nat -> Bool
    op serviceTypeError
              : Name Constraint Policies SpecifiedProps Nat -> Bool
    ...
  }
}
```

An IDL `interface` can have declaration similar to the multiple inheritance. For example, the `Lookup` interface specifies three pre-existing ones `TraderComponents`, `SupportAttributes`, and `ImportAttributes`. The interpretation of the IDL inheritance is just an importation. It means that one can access the data introduced by the three interfaces through the `Lookup` interface. The `Lookup` interface is supposed to define a concrete CORBA object that has its own object identifier referenced by client programs.

One can employ the specification style in (b) of Figure 3.3. The style allows to define an object that has multiple bodies, each of which is introduced in a different module. Since the object terms have the same object identifier, the technique can simulate the situation where one can access, for example, an attribute of `TraderComponents` through the object identifier given to the `Lookup` object.

5.4.3 Summary

From the translation method illustrated with examples in the preceding subsections, it is apparent that the number of CafeOBJ modules is almost identical with that

of IDL components. Each IDL `interface` have one auxiliary module that imports and introduces all the symbols that are used in the definition of the corresponding CafeOBJ object module.

The total number of CafeOBJ modules is some 80; 33 for the modules each introducing one exception, 8 for the concurrent objects corresponding to IDL `interface`, 5 for IDL `struct` and 3 for its sequence version, 11 for IDL `typedef` and 4 for its sequence version, 2 for IDL `enum` and 1 for IDL `union`, and some auxiliary modules necessary to define the concurrent object modules compactly.

Among the eighty CafeOBJ modules, eleven corresponds to datatypes introduced as the IDL `typedef` and these may have clear correspondence with some of the sort symbols introduced in the information viewpoint descriptions. Concretely, five of them are used to represent *names* of some kind such as `ServiceTypeName` or `TraderName`. These datatypes can be directly related to the sort `Name`. Further, the IDL `typedef`'s, `PropertyValue` and `PolicyValue`, being aliased to the IDL `any`, corresponds to the sort `Value`.

5.5 Discussions

By presenting concrete example descriptions, this chapter has shown that CafeOBJ is expressive enough to represent both the information and computational viewpoint specifications of the ODP trader. The information viewpoint is mostly transcriptions of the Z version of the descriptions in the standard document. It, however, requires some *operational* interpretation on the Z descriptions. As a result, the CafeOBJ descriptions are executable and thus will be of much help in understanding the functionality at an abstract level. The computational viewpoint, on the other hand, is a syntactic translation of the IDL. The IDL is basically a data definition language and is not suitable for describing functional behavior of the specificand. Thus, the resultant CafeOBJ descriptions do not include functional behavior in a significant manner either. The descriptions, however, have all the definitions necessary to enable syntax and sort checking in the CAFE environment.

The contribution of the above exercise is twofold; one area of contribution is in a practice of the algebraic specification technologies and another is in an algebraic approach to the ODP specifications. As for the first area, the contributions are summarized as follows.

1. Expressibility

The information viewpoint is mostly concerned with information modeling, and thus abstract datatypes, functions, predicates and constraint relations are the main computational entities to be modeled. The computational viewpoint needs dynamic, concurrent aspects. The exercise in this chapter shows that CafeOBJ can encode all the computational entities compactly, and that CafeOBJ is expressive enough to describe the richness of the ODP trader.

2. Module Decomposition

   Respecting the organization of the recommendation document, we have identified CafeOBJ modules that reflect the structure of the original presentation. The structure acts as a good guideline to derive a unit of modules. In a word, we have used the structure of the recommendation document as domain specific heuristics to decompose the problem (the ODP trader).

Regarding to the algebraic approach to the ODP specifications, we will discuss the contributions by comparing it with related works on FDTs for ODP.

The CafeOBJ description of the ODP trader is the first attempt of algebraic approach to describing both the information and computational viewpoints of the ODP trader. In addition, writing both viewpoints in a single language (CafeOBJ) is unique and the result is a counter example of the folklore in [34], which says that no single FDT can specify the richness of the ODP trader.

Bowman et al. [16] analyze the characteristics of the ODP standard in general (not restricted to the ODP trader), and discuss the pros and cons of the existing FDTs for ODP. They identify seven requirements for FDTs: (1) object-orientation, (2) dynamic reconfiguration or concurrency, (3) non-functional properties, (4) co-existence with multiple FDTs, (5) support for formal reasoning, (6) abstraction, and (7) standardized FDTs. Of the seven, CafeOBJ satisfies four requirements (1, 2, 5, 6) and falls short of (3) and (7). However, since non-functional requirement (3) is generally hard to capture in any languages, existing FDTs have more or less disadvantage in this respect.

Among the seven requirements, co-existence with multiple FDTs (4) is not a mandatory. Bowman et al. [16] discuss that a central requirement of FDTs arising from ODP is a need to support multiparadigm specification. With a FDT allowing multiparadigm specification, one can describe various viewpoint specifications in a single language and then the requirement (4) can be thrown away. The exercise in this chapter is the first published work to concretely show that CafeOBJ can specify both the information and computational viewpoints.

Another interesting research area relating to the ODP is to find consistent semantics for the information and computational models [12][17]. Bernardeschi et al. [12] define two new languages, one for the information viewpoint and the other for the computational viewpoint, and establish relationship to translate the former language construct to the latter. However, their language for the computational viewpoint is a somewhat hypothetical concurrent object-based language and is independent of the computational viewpoint of the ODP trader that is written using the OMG IDL. Bowman et al. [17] uses the Z notation for the information viewpoint and LOTOS for the computational one. The hard part is to have mapping LOTOS language construct into the Z notation.

Our work uses a single multiparadigm algebraic logic language CafeOBJ to represent different abstract levels for different aspects of the specificand while providing the same specification fragments for a set of the common concepts. By sharing some of the basic concept modules such as `NAME` or `VALUE` in the two viewpoints, the correspondence between the two aspects is clearer than those in the standard document. Thus, the CafeOBJ descriptions can be concise. Establishing a mapping relation between the two different viewpoint specifications is a future direction, which is not addressed in this work.

In closing, we point out how this chapter relates to the next; it needs further design decisions to complete the functional behavior of the computational viewpoint specification. In the recommendation document, informal descriptions of the functional behavior are scattered over many pages with illustrative figures only. Importantly no *actual* specification that is necessary for constructing program is presented because the document is a standard recommendation, but not a design document. The descriptions are meant only for the explanatory purpose. Thus, it requires that further design activities should be involved even to complete the functional behavior of the computational viewpoint. For example, the CafeOBJ module `ITERATOR` (Section 3.2.2) basically provides a functional specification of the IDL `OfferIterator` (below) and the `OfferIdIterator`.

```
interface OfferIterator {
  unsigned long max_left ( ) raises ( UnknownMaxLeft);
  boolean next_n ( in unsigned long n, out OfferSeq offers);
  void destroy ();
};
```

To write the CafeOBJ module `ITERATOR`, one has to make several design decisions such as using `List` as the internal representation of managed data. The next chapter

deals with further design issues in developing object-oriented frameworks for the ODP trader server.

Chapter 6

Constructing a Trading Server with CafeOBJ

6.1 Introduction

The ODP trader, discussed in the previous chapter, is an important service in a distributed computing environment. Vendors need an implementation of a trading server that strictly confirms to the recommendation. The trader server is expected to have well-organized architecture to be tolerable in a long term maintenance. Thus, the software development process of constructing a trading server should take care of two divergent facets: (1) proper understanding of the problem (the ODP trader recommendation), and (2) realizing well-organized architecture. And the situation is typical in the industrial settings.

For the second facet, one can use the object-oriented framework technology in order to establish a highly modular architecture that aims to allow future customizations and ease of maintenance. In addition, one may in principle use the GILO-based modeling method in Chapter 4. Although one can get benefit of using CafeOBJ as a design validation checker, the method is general-purpose and does not talk about the problem concretely. Therefore, the method is not powerful enough to be a help for understanding the problem (the first facet). A new design method that is focussed more on the problem itself and also that can benefit from CafeOBJ is desirable.

This chapter includes a proposal of a problem-oriented design method for the development of an ODP trading server. The development process employs CafeOBJ, a multiparadigm algebraic specification language, and the target system is constructed

as a collection of object-oriented frameworks. The development process starts with the ODP/OMG document [131][134] as an input source.[1] One can use the outcome of the last chapter on the CafeOBJ descriptions of the computational viewpoint specification as a start. The process aims to provide a problem-oriented method that employs a technique of lightweight use of CafeOBJ [96].

The core part of the development process is a new analysis method performed at early stages of the development [97]. The method concentrates on *separation of concerns*, which helps us understand the complex real world problem of the ODP/OMG trader specification. And the method *talks* about the intrinsic nature of the problem at hand.

This chapter reports the experience in using CafeOBJ to develop the frameworks in the industrial setting. The discussion covers a whole development process including both the problem analysis and the Java program construction phases. In the case study, CafeOBJ is used as a validation checker for a design artifact of the trading server.

6.2 Problem-Oriented Development

6.2.1 Early Stages of Framework Development

In real world software system, structural properties as well as functional behavior are important with regard to the ease of customization or maintenance. Systems, without clean structural organization architecture, are difficult to customize because a slight change in the functional requirement may be scattered over several program modules and thus is hard to be traced. The object-oriented framework is a promising technology to resolve the issue by allowing appropriate mapping of functionality to the constituent object structure. The importance of the technology can be said to shed light on the structural organization. Although some methods have been proposed to be applied to framework development, designing well-organized frameworks is still an art.

The reason why the proposed methods for framework development are inadequate may be related to Jackson's influential remark on two important issues on method and problem in general [64]; (1) methods cannot be panaceas (medicines

[1] Because the computational viewpoint of the ODP trader is in accordance with the OMG trading service, both the ODP trader and the OMG trading service are treated one single standard specification in this chapter.

that cure all diseases), and (2) very few problems can be decomposed into *homogeneous* structures. Because real world system is complex, the problem is decomposed into a set of subproblems. The subproblem is *heterogeneous* in the sense that it needs a different problem frame (a kind of structural pattern to solve the subproblem). In addition, methods should be related to a particular class of problem and thus give a sharply focused help in reaching a solution.

A new design method is required for developing object-oriented frameworks for a real world complex system. The presumed method bridges the gap between the complex problem and existing object-oriented methods; the given problem is one such that is decomposed into a set of heterogeneous subproblems, but the object-oriented methods can handle only homogeneous world of objects. The new design method focuses on decomposing the whole problem into a set of simple subproblems. Each subproblem needs not to be formalized as object-oriented, but is associated with specification technique best fitted for the intrinsic nature of the subproblem. The design process continues to produce homogeneous object solutions by using existing object-oriented design methods.

6.2.2 Aspect-Centered Design Method

Existing design methods for developing object-oriented frameworks, such as the collaboration-based design [9][20][92] and design pattern [42][67][98], are effective in general. However, the methods have several drawbacks. (1) The design methods have their basis on the object-orientation, and explicitly assume that *object* is sole constituent of the system. (2) The methods provide only general guidelines of decomposing a whole problem into constituent objects, and mention no concrete hint for the decomposition. (3) The design pattern is a catalog of useful design idioms that provide concrete solutions, but most of them are at a programming level and are thus not suitable for use at an early stage of the development.

Real world software such as the ODP trading server is a complex system. As will be discussed in Section 6.3.1, the target problem has various aspects that are not amenable to *homogeneous* object-oriented modeling. Analyzing the target problem and decomposing it into a set of subproblems is the most important task.

The first step of the process (the aspect design phase[2]) is identifying a set of distinct aspects in the problem to obtain a semi-formal description. The phase

[2]The terminology, *aspect* is borrowed from [70] because viewing a software system consisting of many *aspects* is the common idea.

starts with analyzing both the ODP document and the system requirement. Using a specification technique best fitted for the characteristics of each aspect reaches aspect solutions. The solutions form a whole design artifact that is the input to the next phase. The phase (object-oriented design) makes use of existing methods such as collaboration-based design or design patterns. In the course of preparing the design document, some part of the framework is implemented incrementally in Java [5].

The following two characteristics of the aspect design seems a well-known common practice: (1) decomposing a large complex problem into a set of manageable subproblems to solve individually, and (2) seeing a target system from various viewpoints. For example, a top-down functional design approach deals with decomposition into procedures or processes. Its decomposition, however, is homogeneous and hierarchical only. OMT provides three *models* (object, dynamic, and functional), and promotes a method to describe the system behavior by using the three different models [102]. The model, however, represents only different viewpoints of a same entity, *object*.

On the other hand, the important characteristic of the aspect-centered design method is *heterogeneity*. *Aspect* is related to a subproblem that is further refined and elaborated to reach solution description individually. The subproblem needs not to be object-oriented, but is *heterogeneous* in the sense that each subproblem is associated with specification technique best fitted for its intrinsic nature.

A simple example on Composite pattern [42] may illustrate the above discussion. Composite pattern is a design pattern to represent tree structures. When a target system has an aspect of language processing such as a kind of query language, Composite pattern together with Visitor pattern is quite useful in implementing language processing subsystem.[3] However, BNF (Backus Normal Form) is a better notation to discuss the grammatical aspects of the problem such as abstract syntax of the language. BNF is more concise than class diagrams representing the abstract syntax tree that follows the Composite pattern. Usually BNF is used in comparing various designs, and then the patterns are employed to instantiate appropriate classes. Because other part of the given problem may use notation different from BNF, the problem can be said to be decomposed into a set of *heterogeneous* subproblems. And, the resultant classes are *homogeneous* representations.

Since identifying aspects in the target problem is not a well-established method-

[3]Section 6.3.2 illustrates how to use Composite and Visitor patterns in implementing an aspect of the trading server.

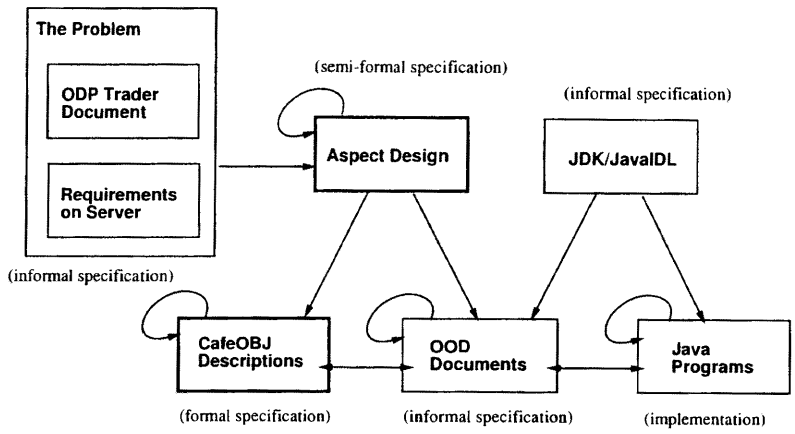


Figure 6.1: Development Process

ology, the present chapter relies on a case study. The presented result may not be generally applicable, but provides a useful insight on the application of the method because the case study talks about the problem in a concrete manner. The rest of this chapter presents experience of a case study in applying the aspect design method to the development of object-oriented frameworks to implement the ODP trading server and discusses the pros and cons of the proposed approach.

6.2.3 Overview of Development Process

Figure 6.1 summarizes the development process, which is one adapted from a process based on the parallel-iterative model of software development [120].

The aspect-centered design phase starts with the analysis of the ODP document and of what is required on the target system. Then the specification techniques that best describe the characteristics of each of the aspects are selected (Table 6.1 in Section 6.3.1). From the semi-formal description of the aspect design, informal specification descriptions are obtained with the help of standard techniques using collaboration-based design [9][20][92] and design patterns [42][67]. In particular, the collaboration-based design, sometimes called scenario-based design, focuses on the analysis of interaction patterns between participant objects, and promotes to use notations such as MSC (Message Sequence Chart). During this phase, the specifications of the JDK library and JavaIDL are referred to. While the informal specifications and the implementation are being prepared, CafeOBJ descriptions of

the aspects are also being prepared as a formal specification document.

The aspect-centered design is quite useful because the ODP/OMG trading service server is a complicated specification. Identifying six primary aspects and refining and elaborating each one with an appropriate design technique individually facilitates the design and implementation of the object-oriented frameworks. Aspect design alone, however, comes with one drawback.

The basic idea of the aspect design is decomposition of a complicated problem into a set of various aspects. Conversely, completing the system requires integration of the solutions of each aspect. Without the formal specification descriptions that can be mechanically analyzable, we only have a combination of mostly analyzable functional programs and unanalyzable graphical notations such as MSC. What could be done is human design review only. Instead we use a specification language to describe solution descriptions of each aspect, and thus make it easy to check integrity when all the aspect descriptions are put together.

Our choice is a multiparadigm algebraic specification language CafeOBJ, which has clear semantics based on hidden order-sorted rewriting logic [29][40]. The logic subsumes order-sorted equational logic [35][48], concurrent rewriting logic [78], and hidden algebra [49].[4] Being an algebraic specification language, CafeOBJ promotes a property-oriented specification style; the target system is modeled as *algebra* by describing a set of properties to be satisfied. By introducing suitable specification modules (*algebra*), various computational models ranging from MSC[5] to functional programming and concurrent objects can be encoded in CafeOBJ. Further because CafeOBJ has clear operational semantics, specifications written in it are executable. It helps much improve the design by validating functionality of the system.

6.3 Trading Function and Design Aspects

This section first summarizes the ODP trading function and then presents the identified design aspects in details. The summary, following Section 5.2.1, aims to provide the explanation of the core part of the computational viewpoint, which is needed to understand the way to identify the design aspects.

[4] We will not consider hidden algebra.

[5] Basically a transition system. The use of MSC in object-oriented modeling and its encoding method in CafeOBJ are described in Section 4.3.

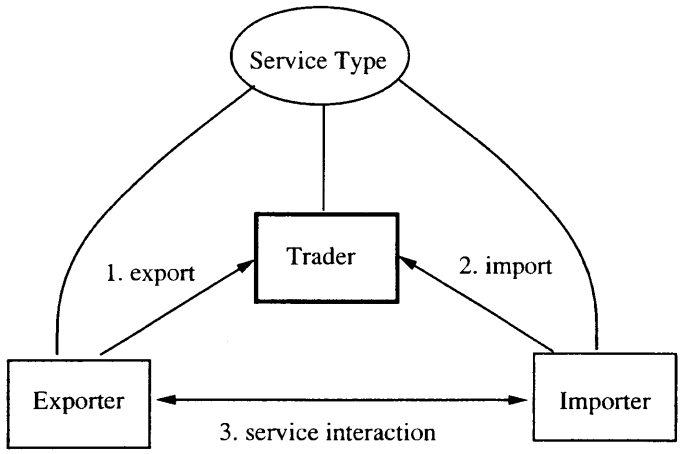


Figure 6.2: Trading Scenario

6.3.1 Trading Function

Figure 6.2 shows a trader and participants in a trading scenario. The `Exporter` exports a service offer. The `Importer` imports the service offer and then becomes a client of the service. The `Trader` mediates between the two by using the exported service offers stored in its own repository which is ready for import requests. Every service offer has a service type and is considered to be its instance. The service type holds the interface type of the object being advertised and a list of property definitions. A property is a triple of name, type of the value and mode which indicates whether the property is mandatory or optional. Further, a subtype relation is defined between service types. The relation is useful both in importing offers that do not exactly match the request and in defining a new derived service type from the existing ones. The common concept defined in the ODP/OMG trader specification document, such as the service type, the service offer or the property definition, plays a central role in the trading scenario, and thus proper understanding of the concepts is important in designing the trading server.

Because importing, implemented as the `Lookup` interface, is the most complex and interesting function, we focus on its design and implementation. The following IDL fragment shows a portion of a `query` operation of the `Lookup` interface. It is the operation for importing.[6]

[6]Parameters not relevant here are omitted for brevity.

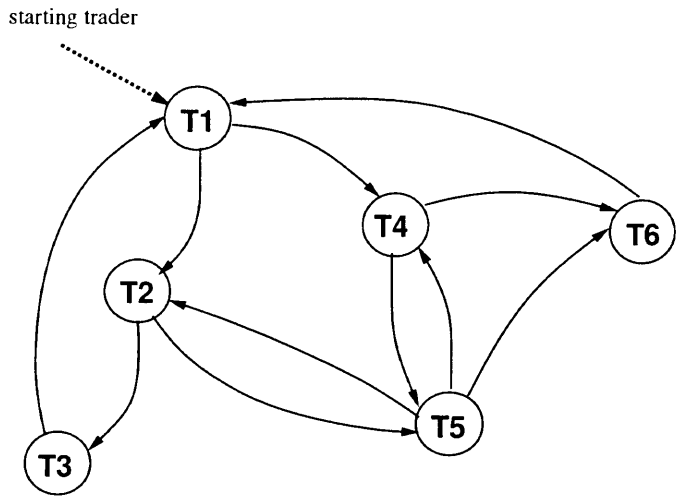


Figure 6.3: Federated Trader Group

```
typedef Istring ServiceTypeName;
typedef Istring Constraint;
typedef Istring Preference;

void query(
  in ServiceTypeName type,
  in Constraint constr,
  in Preference pref,
  ...
  out OfferSeq offers,
  out OfferIterator offer_itr,
  ...
) raises ( ... )
```

The first parameter `type` specifies the service type name of requested offers. The parameter `constr` is a condition that the offers should satisfy and is a *constraint language* expression that specifies the condition in a concise manner. The expression describes semantically a set of property values of service offers that the client tries to import. The trader searches its repository to find service offers whose property values satisfy the constraint expression. Understanding the search process requires

an explicit formulation of the constraint language, and a formal definition of the language would be valuable.

The parameter `pref` is preference information that specifies that the matched offers are sorted according to the preference rule. The sorted offers are returned to the importer in the `out` parameters: `offers` and `offer_itr`. The ODP/OMG standard specification also defines a set of *scoping policies* to provide the upper bounds (cardinalities) of offers to be searched at various stages of the search process. Actual values of the cardinalities are determined by a combination of the importer's policies and the trader's policies. Understanding the role of each scoping policy requires grasping the global flow of the base query algorithm.

The ODP/OMG trader defines the specification for interworking or federation of traders to realize scalability. The use of a federated trader group enables a large number of service offers to be partitioned into a set of small offer sets of manageable size. One trader is responsible for each partition and works with the other traders when necessary. Figure 6.3 shows an example of a federated trader group. The traders T1 to T6 are linked as indicated by the curved arrows. When a `query` is issued on the starting trader T1 and a federated search is requested, the traders T2 to T6 also initiate local searches. All the matching offers are collected and returned to the client importer.

The federation process uses a set of policies controlling the graph traversal. A simple one is the `request_id` that cuts out unnecessary visits to the same trader, and another is the `hop_count` that restricts the number of traders to visit. A set of policies called the *FollowOption* controls the traversal semantically. For example, a link marked with `if_no_local` is followed only if no matched offer is found in a trader at the source of the link. Again, the role of each policy is hard to understand without referring to the global flow of the base federation algorithm.

The ODP/OMG standard describes the query and federation algorithm and the role of each policy by using illustrative examples. In particular, the explanation adopts a stream processing style of selecting appropriate offers from an initial candidate set. The overall picture, however, is hard to grasp because the descriptions are informal and scattered over several pages of the document [131][134]. A concise description is needed to prepare a precise design description of the algorithm, and functional programming style is a good candidate.

After the analysis of the trader specification mentioned above, the six aspects in Table 6.1 are identified together to form the whole system. Table 6.1 shows the aspects and the accompanying specification techniques.

In summary, the ODP/OMG trader is a medium scale, non-trivial problem that

Design Aspect (example)	Specification Technique
common concept (service type)	abstract datatype
policy (scoping policy)	functional programming
algorithm (query, federation)	stream-style programming
language (constraint language)	denotational semantics style
functional object (Lookup)	concurrent object
architecture	concurrent object, ambient

Table 6.1: Design Aspects and Specification Techniques

has six heterogeneous subproblems. Since the aspects are quite distinct in nature, no general-purpose methodology is adequate. Using specification techniques suitable for each aspect is a better approach to a systematically and sharply focused help in reaching the solution.

6.3.2 Semi-formal Descriptions of Aspect Solutions

This subsection will discuss some of the important aspects including language, policy and algorithm (see Table 6.1).

Query Algorithm and Policy

The *policy* of the trading service is just a parameter that modifies behavior of both local and federated query algorithm. It is hard to understand the meaning of policies without referring to basic algorithm. In addition, in order to grasp the algorithm at a glance, a concise notation is needed. The notation adopted is the one borrowed from a functional programming language StandardML [82] augmented with some symbols to describe and handle set-like collections of data. Another important decision here is a choice of a stream-style functional programming for the query algorithm. This viewpoint is in accordance with the informal presentation in the ODP/OMG document [134].

Local Query

The following describes the query algorithm that is executed locally in one trader.

The top-level function `IDLquery(T,I)` (♯1), which is invoked as an IDL request, takes the form below. All the function definitions are supposed to come in the lexical context (as **fun** $\cdots$) of the `IDLquery(T,I)`. The functions can use `T` and `I` freely as global constants, where `T` refers to the trader state or trader's policy and `I` denotes the importer's request and policy.

```
(♯1) fun IDLquery(T,I) =
       fun query() = if valid-trader()
         then if valid_id() then
           (select ∘ federation ∘ search)(T.offers)
           else φ
         else IDLquery(remote_trader(T),I)
       fun ···
       in
         query()
       end
```

First, it checks whether the request is on the trader itself. Then, it invokes the body of the **query** function, which is described as a stream-style processing consisting of **search**, **federation**, and **select**.

The function **search** (♯2) is responsible for collecting candidate offers. The candidate space is then truncated by appropriate policies on cardinality. The **search** uses two such cardinality filters.

```
(♯2) fun search(R)
       = (match_cardinality_filter ∘ match ∘ search_cardinality_filter ∘ gather)(R)
```

The function **gather** (♯3) collects offers that have the specified service type. If the importer policy has a false `I.exact_type_match`, offers of all the subtypes of the specified one should be collected (♯4). In the definition of (♯5), the content of TypeRepository is a directed acyclic graph (`G`) whose node (`n`) is a service type and edge is a service subtype relationship ($\prec$).

```
(♯3) fun gather(R) = { s ∈ R | s.ServiceType ∈ requested_types() }
(♯4) fun requested_types()
       = if I.exact_type_match then { type(I.ServiceTypeName) }
         else collect_subtypes(T.TypeRepository,type(I.ServiceTypeName))
(♯5) fun collect_subtypes(G,N) = { n ∈ node(G) | n ≺* N }
```

The function `match` returns offers that satisfy the importer's requirement expressed as a constraint language expression. Its representation will be discussed afterward.

The next two functions (♯6 and ♯7) implement filtering on cardinality mentioned above. Both uses the `truncate` function to filter out unnecessary offers. The two functions represent how to compute each cardinality in a concise manner. The role of the policy concerning the cardinality is thus clear.

```
(♯6) fun search_cardinality_filter(R)
     = truncate((if exist(I.search_card)
                    then min(I.search_card, T.max_search_card)
                    else T.def_search_card), R)
(♯7) fun match_cardinality_filter(R)
     = truncate((if exist(I.match_card)
                    then min(I.match_card, T.max_match_card)
                    else T.def_match_card), R)
```

When a federated query is not in use, the function invoked after the `search` is the `select` (♯8). With a help of `order` (♯9), it uses the importer's preference expression to make the collected offers into a sequence, the sequence of which reflects the preference order.

```
(♯8) fun select(R) = (return_cardinality_filter ∘ order)(R)
(♯9) fun order(R) = order_on_preference(R,I.preference)
(♯10) fun return_cardinality_filter(R)
      = truncate((if exist(I.return_card)
                     then min(I.return_card, T.max_return_card)
                     else T.def_return_card), R)
```

Federated Query

Next deals with the federated query algorithm that involves more than one trader.

The function `federation(R)` (♯11) is responsible for controlling a federated query. It first checks whether further IDL query requests are necessary to linked traders by consulting the trader's policy on `hop_count`. The auxiliary function `new_hop_count()` (♯12) demonstrates the role of policies involved in the federation control.

```
(♯11) fun federation(R)
      = let val new_count = new_hop_count()
        in
            if new_count ≥ 0 then traversal((I with new_count),R) else R
        end
(♯12) fun new_hop_count()
      = (if exist(I.hop_count)
         then min(I.hop_count, T.max_hop_count) else T.def_hop_count) - 1
```

The function `traversal` (♯13) is invoked with a modified importer policy (`J`) and the offers obtained locally (`R`). It controls invocations on the target trader located at the far end of the specified link. The control again requires a scoping policy calculation, which involves the link policies as well as the trader's policies and the importer's. The two functions `new_importer_follow_rule(L,J)` (♯14) and `current_link_follow_rule(L,J)` (♯15) show the definitions of `FollowOption` rule. The function `dispatch` (♯16) concisely gives specifications of the use of the `FollowOption` rule of the specified link; the rule defines three cases, `local_only`, `if_no_local`, and `always`. How to construct the final offers differs in each case.

```
(♯13) fun traversal(J,R)
      = ⋃∀L∈T.links dispatch_on(current_link_follow_rule(L,J), L,
                                (I with new_importer_follow_rule(L,J)),R)
(♯14) fun new_importer_follow_rule(L,J)
      = if exist(J.link_follow_rule)
        then min(J.link_follow_rule, L.limiting_follow_rule, T.max_follow_policy)
        else min(L.def_pass_on_follow_rule, T.max_follow_policy)
(♯15) fun current_link_follow_rule(L,J)
      = if exist(J.link_follow_rule)
        then min(J.link_follow_rule, L.limiting_follow_rule, T.max_follow_policy)
        else min(L.limiting_follow_rule, T.max_follow_policy, T.def_follow_policy)
(♯16) fun dispatch_on(local_only,L,J,R) = R
        | dispatch_on(if_no_local,L,J,R) = if empty(R) then follow(L,J) else R
        | dispatch_on(always,L,J,R) = follow(L,J) ∪ R
(♯17) fun follow(L,J) = IDLquery(L.trader,J)
```

Constraint Language Processing

```
CExp ::= Pred
Pred ::=  L
        | Exp == Exp
        | exist L
        | not Pred
        | Pred and Pred
        | Pred or Pred
        | ...
```

Figure 6.4: Abstract Syntax (a part)

What follows deals with the aspect of the constraint language processing. The accompanying specification technique is a denotational style of language definition.

Two functions (`order_on_preference` and `match`) used in the main algorithm involve evaluation of a constraint expression and a preference expression. Each function is defined to call an evaluation function (either $\mathcal{CE}$ or $\mathcal{PE}$).

fun match(R) = $\mathcal{CE}$ ⟦ I.constraint ⟧ R
fun order_on_preference(R,X) = $\mathcal{PE}$ ⟦ X ⟧ R

The specification technique follows a standard way of rigorous language definition. First, the abstract syntax of the language is introduced. A portion is shown in Figure 6.4. Second, a valuation function is defined for each syntax category; $\mathcal{CE}$ is an example function for constraint expressions (`CExp`) and it further calls $\mathcal{LE}$ of the valuation function for predicates (`Pred`). `R` stands for a set of offers and `O` is an offer.

$\mathcal{CE}$: CExp → R → R
$\mathcal{LE}$: Pred → O → Bool

Then, the specifications of the constraint language interpreter or evaluator are best seen by the definitions of the valuation function. The definitions can be formulated systematically by studying the meaning of each abstract syntax element.

$\mathcal{CE}$ ⟦ E ⟧ R = { O ∈ R | $\mathcal{LE}$⟦ E ⟧ O }
$\mathcal{LE}$⟦ L ⟧ O = prop-val(O,L)$\downarrow_{Bool}$
$\mathcal{LE}$⟦ E1 == E2 ⟧ O = $\mathcal{AE}$⟦ E1 ⟧ O == $\mathcal{AE}$⟦ E2 ⟧ O
...

A set of definitions of the valuation functions form the rigorous design descriptions of the constraint language interpreter.

6.3.3 CafeOBJ Descriptions of Aspect Solutions

This subsection deals with some example CafeOBJ descriptions of the aspect solution.

Common Concepts

Of the entries in Table 6.1, the common concepts such as `ServiceType` and `PropertyDefinition` are easily translated into CafeOBJ modules because abstract datatype technique provides a concise way to model such basic vocabularies. Further, the common concept modules can be the same as those for the information viewpoint. Thus one calls them the *common concepts*.

The module `SERVICE-TYPE-NAME` introduces a new sort `ServiceTypeName` that specifies a set of service type name.

```
mod! SERVICE-TYPE-NAME { [ ServiceTypeName ] }
```

`ServiceTypeName` may be a subsort of `Name`.

Service type is defined as an abstract datatype. The module `SERVICE-TYPE` provides a concise notation to represent record-like terms and accessor functions. The module `SERVICE-SUBTYPING` defines the service subtyping relationship. Actually `_is-subtype-of_` is a predicate used to calculate whether the operand service types satisfy the subtyping relationship.

```
mod! SERVICE-TYPE {
  [ ServiceType ]
  protecting (PROPERTY-DEFINITION)
  protecting (INTERFACE)
  protecting (SERVICE-TYPE-NAME)
  signature {
   op [name=_, interface=_, properties=_] :
       ServiceTypeName Interface PropertyDefinition -> ServiceType
   op _.name : ServiceType -> ServiceTypeName
   op _.interface : ServiceType -> Interface
   op _.properties : ServiceType -> PropertyDefinition
```

```
    op _.property(_) : ServiceType PropertyName -> ModeAndType
    op _.names : ServiceType -> Seq<PropertyName>
  }
  axioms {
    var S : ServiceTypeName  var T : ServiceType
    var N : PropertyName  var I : Interface  var PS : PropertyDefinition

  eq ([name=(S), interface=(I), properties=(PS)]).name = S .
  eq ([name=(S), interface=(I), properties=(PS)]).interface = I .
  eq ([name=(S), interface=(I), properties=(PS)]).properties = PS .
  eq ([name=(S), interface=(I), properties=(PS)]).property(N) = lookup(PS,N) .
  eq (T).names = names((T).properties) .
  }
}

mod! SERVICE-SUBTYPING {
  protecting (SERVICE-TYPE)
  signature { op _is-subtype-of_ : ServiceType ServiceType -> Bool }
  axioms {
    vars T1 T2 : ServiceType

  eq (T1) is-subtype-of (T2)
  =   (((T1).interface is-subinterface-of (T2).interface)
  and ((T1).names includes (T2).names))
  and (mode-strength((T2).names,(T1).properties,(T2).properties)) .
  }
}
```

Because SERVICE-TYPE is one of the basic domain specific vocabulary in the ODP trader, one may expect that the definition in the computational viewpoint is the same as that in the information viewpoint. The definition above, however, is slightly different from the one in Section 5.3. It is because SERVICE-TYPE in the computational viewpoint is elaborated one that has an additional member.

Query Algorithm and Policy

Below CafeOBJ modules are explained with referring to the pseudo StandardML descriptions in Section 6.3.2.

The first example is the top-level function `IDLquery(T,I)` (♯1), which is invoked as an IDL request takes the following form.

```
fun IDLquery(T,I) =
  fun query() = if valid-trader()
    then if valid_id() then (select ∘ federation ∘ search)(T.offers) else φ
    else IDLquery(remote_trader(T),I)
  fun ···
  in
    query()
  end
```

`IDLquery(T,I)` first checks to see whether the request is on the trader itself. Then, it invokes the body of `query` function, which is a stream-style processing consisting of `search`, `federation`, and `select`. The module `QUERY-ALGORITHM` is a CafeOBJ description of `IDLquery(T,I)`.

```
mod! QUERY-ALGORITHM [X :: TH-TRADER-STATE, Y :: TH-IMPORTER-REQUEST ] {
  signature {
    op query : TraderState Request Set<Offer> -> Seq<Offer>
    op valid-trader : TraderState TraderName  -> Bool
    op valid-request-id : TraderState RequestId -> Bool
    op remote-invoke : TraderState Request -> Seq<Offer>
  }
  axioms {
    var T : TraderState  var I : Request var S : Set<Offer>

    eq query(T,I,S) = if valid-trader(T,(I).starting-trader)
                      then (if valid-request-id(T,(I).request-id)
                            then select(T,I,federation(T,I,search(T,I,S)))
                            else empty<Offer> fi)
                       else remote-invoke(T.remote,I) fi .
    ...
  }
}
```

The function `search` (♯2) collects candidate offers. The candidate space is then truncated according to appropriate policies on cardinality. The `search` uses two such cardinality filters. One is `search_cardinality_filter(R)` (♯6) below.

```
fun search(R) = (match_cardinality_filter ∘ match
                 ∘ search_cardinality_filter ∘ gather)(R)

fun search_cardinality_filter(R)
= truncate((if exist(I.search_card)
            then min(I.search_card, T.max_search_card)
            else T.def_search_card), R)
```

The module SEARCH-CARDINALITY defines the way that the *search cardinality* is calculated by using a trader's policy and an importer's policy.

```
mod! SEARCH-CARDINALITY [X :: TH-TRADER-POLICY, Y :: TH-IMPORTER-POLICY] {
  signature {
    op search-cardinality : TraderPolicy ImporterPolicy -> Cardinality
  }
  axioms {
    var T : TraderPolicy   var I : ImporterPolicy

    eq search-cardinality(T,I)
    = if exist((I).search-card)
         then min((I).search-card,(T).max-search-card)
         else (T).def-search-card fi .
  }
}
```

Because `search_cardinality_filter(R)` is defined as a pure function, the CafeOBJ description is a literal translation of (♯6).

The function `federation(R)` (♯11) controls a federated query process. One important thing to note is the use of various federation policies. Actually, the function `traversal` (♯13), called from the `federation(R)`, controls invocations on the target trader located at the far end of the specified link by consulting appropriate policy values.

The two functions `new_importer_follow_rule(L,J)` and `current_link_follow_rule(L,J)` show how to use the `FollowOption` policy. Finally, the function `dispatch` shows the use of the `FollowOption` rule. The rule defines three cases – `local_only`, `if_no_local`, and `always` –, and how the final offers are constructed depends on the case.

```
fun traversal(J,R)
```

$= \bigcup_{\forall L \in T.links}$ dispatch_on(current_link_follow_rule(L,J), L,
(I with new_importer_follow_rule(L,J)),R)
fun current_link_follow_rule(L,J)
= if exist(J.link_follow_rule)
then min(J.link_follow_rule, L.limiting_follow_rule, T.max_follow_policy)
else min(L.limiting_follow_rule, T.max_follow_policy, T.def_follow_policy)
fun dispatch_on(local_only,L,J,R) = R
| dispatch_on(if_no_local,L,J,R) = if empty(R) then follow(L,J) else R
| dispatch_on(always,L,J,R) = follow(L,J) $\cup$ R

The specification of the policy calculation is important as well as the base algorithm because the federation process is dependent on the combination of the policy values. It means that the trader behavior is different for different policy values if one views the trader from the outside.

The module `FOLLOW-OPTION` describes the *FollowOption* policy. Basically it provides `min` functions, and other functions are omitted for brevity. The module `NEW-LINK-OPTION` shows an example of calculating the *FollowOption* policy, which is used in the federation process.

```
mod! FOLLOW-OPTION {
  [ FollowOption ]
  signature {
    ops local-only if-no-local always : -> FollowOption
    op min : FollowOption FollowOption -> FollowOption
    op min : FollowOption FollowOption FollowOption -> FollowOption
    op _<_ : FollowOption FollowOption -> Bool
    ... (omitted) ...
  }
  axioms {
    vars F1 F2 F3 : FollowOption

    eq (F1) < (F2) =  (((F1 == local-only) and (not (F2 == local-only)))
                       or ((F1 == if-no-local) and (F2 == always))) .
    ceq min(F1,F2) = F1  if (F1)<(F2) .
    ... (omitted) ...
  }
}
```

```
mod! NEW-LINK-OPTION [X :: TH-TRADER-POLICY, Y :: TH-IMPORTER-POLICY,
                      Z :: TH-LINK-POLICY ] {
  protecting (FOLLOW-OPTION)
  signature {
    op current-link-follow-rule :
       TraderPolicy ImporterPolicy LinkPolicy -> FollowOption
  }
  axioms {
    var T : TraderPolicy   var I : ImporterPolicy  var L : LinkPolicy

    eq current-link-follow-rule(T,I,L)
    = if exist((I).link-follow-rule)
      then min((I).link-follow-rule, (L).limiting-follow-rule,
               (T).max-follow-policy)
      else min((L).limiting-follow-rule, (T).max-follow-policy,
               (T).def-follow-policy) fi .
  }
}
```

Constraint Language

The CafeOBJ descriptions of the constraint language interpreter are obtained directly from the denotational descriptions of the language. One can use a standard technique for writing denotational descriptions of language in CafeOBJ.

First, the abstract syntax of the language, a portion of which is shown below, is translated into the module PRED-SYNTAX.

```
  CExp ::= Pred
  Pred ::=  L  | Exp == Exp  | exist L         | not Pred
         | Pred and Pred     | Pred or Pred    | ...
```

```
mod! PRED-SYNTAX {
  protecting (EXP-SYNTAX)
  [ Pred,  Exp < Pred ]
  signature {
    op  _==_  : Exp Exp -> Pred
    op  exist : Exp -> Pred
    op  not   : Pred -> Pred
    ... (omitted) ...
```

```
  }
}
```

Then, from the descriptions of the valuation function for each syntax category, the corresponding CafeOBJ description is easily obtained. For example, the module PRED-EVAL provides the valuation function $\mathcal{LE}$ for predicates (Pred).

$\mathcal{LE}$: Pred $\rightarrow$ O $\rightarrow$ Bool

$\mathcal{LE}[\![$ L $]\!]$ O = prop-val(O,L)$\downarrow_{Bool}$
$\mathcal{LE}[\![$ E1 == E2 $]\!]$ O = $\mathcal{AE}[\![$ E1 $]\!]$ O == $\mathcal{AE}[\![$ E2 $]\!]$ O
...

```
mod! PRED-EVAL {
  protecting (PRED-SYNTAX)
  protecting (EXP-EVAL)
  signature { op EP(_)_ : Pred ServiceOffer -> Bool }
  axioms {
    vars E1 E2 : Exp  var P : Pred  var N : PropertyName
    var  O : ServiceOffer

  eq EP(label(N)) O = EE(label(N)) O .
  eq EP(E1 == E2) O = (EE(E1) O) == (EE(E2) O) .
  eq EP(exist E) O  = exist-property(O,(EE(E) O)) .
  eq EP(not P) O    = not(EP(P) O) .
  ... (omitted) ...
  }
}
```

The signature part (EP(_)_) of the module has the type of the valuation function $\mathcal{LE}$, and an equation in the axioms part provides definition of the valuation function for each syntax element of the predicates (Pred).

Functional Objects and Architecture

A functional object describes the behavior of interfaces such as the Lookup and Register, while the architecture here refers to a global organization of functional objects. We use collaboration-based design methods to refine and elaborate these

aspects in order to identify the responsibilities of constituent objects, each of which is then translated into a Maude concurrent object [78] (Section 3.3).

The module IDL-LOOKUP represents the CORBA object implementing the Lookup interface with its behavioral specification written in CafeOBJ. Other functional objects are IDL-REGISTER, IDL-LINK, IDL-ADMIN, TRADER-STATE, TYPE-REPOSITORY, and OFFER-REPOSITORY. In this division of labour, the architecture configuration is just a collection of the concurrent objects (Configuration). Each object is described by the corresponding module such as IDL-LOOKUP.

A Lookup object receiving a query(O,N,C,P,Q,D,H,R) message converts the input parameters into the representation that the part of the algorithm assumes, then invokes the body of the query algorithm (query'), and finally translates the result to match the IDL interface specifications.

```
mod! IDL-LOOKUP[X :: TH-LOOKUP-AID, Y :: TH-LOOKUP-MSG] {
  extending (ROOT)
  protecting (IDL-LOOKUP-LIBRARY)
  [ LookupTerm < ObjectTerm ,  CIdLookup < CId ]
  signature {
    op <(_:_)|_> : OId CIdLookup Attributes -> LookupTerm
    op Lookup : -> CIdLookup
    op invoke-query : TraderState ServiceTypeName Constraint Preference
                Seq<Policy> Seq<PolicyName> Nat Set<Offer> -> Seq<Offer>
   }
  axioms {
    vars O R R' : OId          var REST : Attributes   vars T U : OId
    var N : ServiceTypeName    var C : Constraint      var H : Nat
    var P : Preference         var Q : Seq<Policy>
    var D : Seq<PolicyName>    var X : TraderState     var S : Set<Offer>

  ctrans query(O,N,C,P,Q,D,H,R)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R')),(REST)>
  => trader-state(T,O) initial-offers(U,O) m-wait(O,U,T)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R)),(REST)>
  if valid-trader'(T,Q) and valid-request-id'(T,Q) .

  ctrans query(O,N,C,P,Q,D,H,R)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R')),(REST)>
  => void(R) outArgs(no-offers)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R)),(REST)>
```

```
  if valid-trader'(T,Q) and (not valid-request-id'(T,Q)) .

  ctrans query(O,N,C,P,Q,D,H,R)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R')),(REST)>
  => query(remote-trader(T),N,C,P,Q,D,H,R)
       <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R)),(REST)>
  if not valid-trader'(T,Q) .

  trans m-wait(O,U,T) return(O,U,S) return(O,T,X)
        <(O : Lookup)|(client=(R)),(REST)>
  => void(R) outArgs(invoke-query(X,N,C,P,Q,D,H,S))
        <(O : Lookup)|(client=(R)),(REST)> .

  eq invoke-query(X,N,C,P,Q,D,H,S) = query'(X,request(N,C,P,Q,D,H),S) .
  }
}
```

On receiving a query message, the Lookup object sends messages to the OfferRepository object (U) and the TraderState object (T) to obtain a set of potential offers and the trader state. The Lookup object then invokes query'. The function query' is almost identical to the one defined algorithmically in the module QUERY-ALGORITHM. In order to get compatible with the modeling method based on the concurrent object model, the original algorithm in the module QUERY-ALGORITHM is slightly modified. Actually, the checks on valid-trader and valid-request-id are factored out of the query' algorithm. The above module assumes that the module IDL-LOOKUP-LIBRARY imports all the *library* modules.

The description of the module IDL-LOOKUP takes the same form as the module LOOKUP-TEMPLATE (Section 5.4.2). The former can be said to be an elaboration of the latter one in that IDL-LOOKUP uses the elaborated query algorithm derived as as result of the aspect design solution.

Ambient-based Architecture Configuration

An alternative approach to modeling the architecture configuration is to use the ambient-based technique. The approach is suitable when one explicitly specifies which object is shared. Figure 6.5 illustrates the approach, which is an instantiation of the *pattern* in Section 3.4.

An ambient with its Handle W is introduced to enclose all the shared objects. The

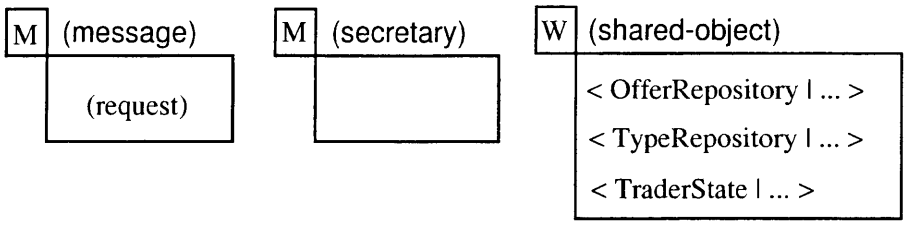


Figure 6.5: Shared Resources in Ambient

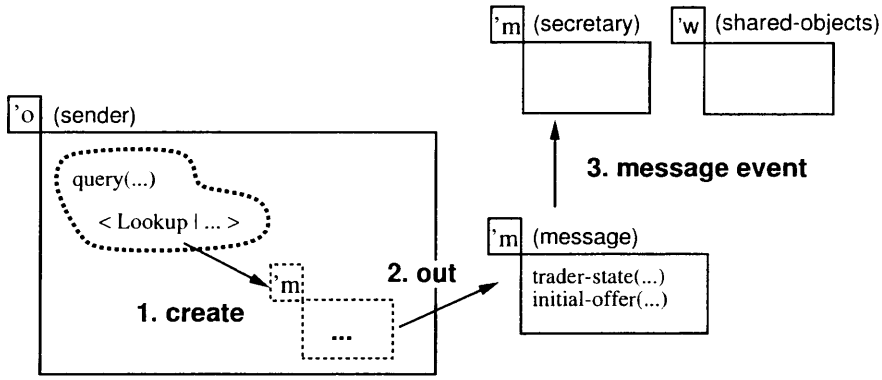


Figure 6.6: Lookup Object in Ambient

objects are `OfferRepository`, `TypeRepository`, and `TraderState`. The ambient annotated with `(secretary)` is the secretary of Section 3.4. It is responsible for transferring request message to the shared ambient one at a time. Thus, multiple simultaneous requests are serialized. The third one named with `(message)` carries the actual request message at the object level. It means that the enclosing message term invokes actions on the objects in the shared ambient. Since the message is a transient ambient, which simulates a message invocation event, a message sender object should create an appropriate message ambient at each invocation time. The three shared objects (`OfferRepository`, `TypeRepository`, and `TraderState`) can be grouped together to be separated from the rest.

In order to complete the formalization using the ambient, however, it also requires some changes in the definition of the sender object. According to the pattern in

Section 3.4, the result of the action by the shared object also takes a form of ambient (another message ambient carrying a result term). Thus, the sender object should have a way to accept such a message ambient, which is done in a similar way as shown in Figure 6.5. Each sender object, such as Lookup object, is enclosed in an ambient similar to the one with Handle W, having an accompanying secretary ambient for receiving an incoming message ambient that embraces the expected return value. Below is a portion in the definition of query method in the module LOOKUP-TEMPLATE.

```
ctrans query(O,N,C,P,Q,D,H,R)
     <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R')),(REST)>
=> query-request(U,T,request(N,C,P,Q,D,H)) m-wait(O,U,T)
     <(O : Lookup)|(offers=(U)),(state=(T)),(client=(R)),(REST)> .

eq query-request(U,T,I)
= send-message('m, (trader-state(T,O) initial-offer(U,O))) .
eq send-message(X) = ('m)[ out('o)@(open('m)@( X )) ] .
```

In the above definition, the ambient enclosing the Lookup object has a Handle of 'o. The send-message(X) creates an ambient with Handle 'm that have two messages trader-state(T,O) and initial-offer(U,O), both of which are sent to appropriate shared objects in the other ambient ('m). Figure 6.6 illustrates the situation.

6.4 Resultant Frameworks Written in Java

The aspect solutions are not the design descriptions of the object-oriented frameworks that are the final artifact. Some refinement or elaboration is necessary to obtain the framework design. Unfortunately the refinement or elaboration process is mostly based on heuristics and does not have a systematic basis.

This section focuses on design of two subsystems of the trader. Each subsystem corresponds to an object-oriented framework. Figures 6.7 and 6.8 show the resultant frameworks written in Java.

Based on the formal language definition presented early, designing the framework for constraint-language processing is straightforward (Figure 6.7). This framework is a representative of using the design patterns [42]: Composite pattern for representing the abstract syntax tree (AST) and Visitor pattern for representing the tree walkers

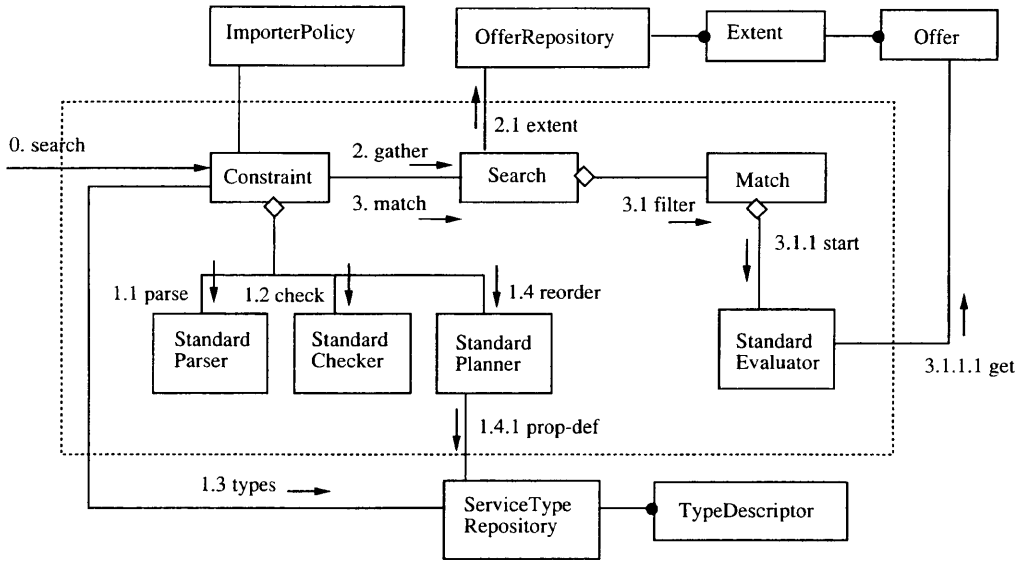


Figure 6.7: Query Processing Framework

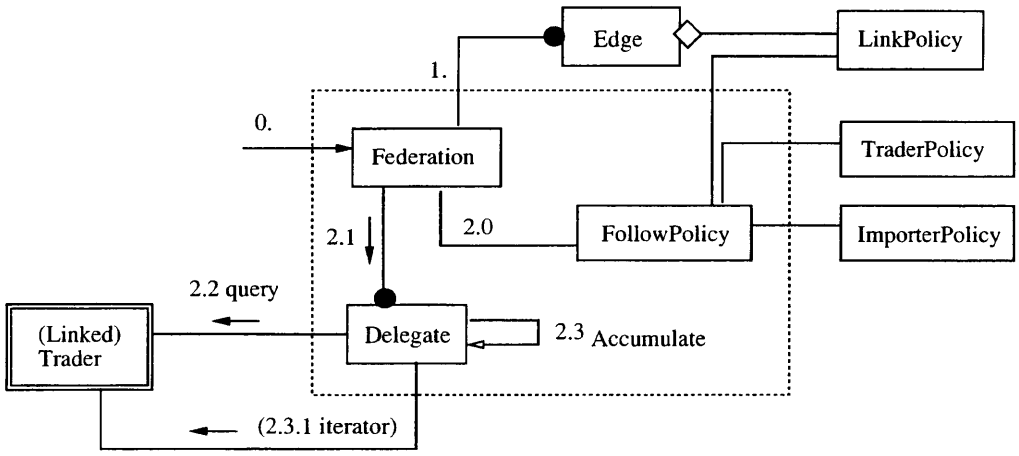


Figure 6.8: Federation Framework

such as a light semantic checker (`StandardChecker`), a filtering condition reorder planner (`StandardPlanner`), and an interpreter (`StandardEvaluator`).

Adopting the above two design patterns, which is the same to use Interpreter pattern, provides a guideline to translate descriptions in a functional programming style to a set of classes in object-oriented programming style. Each datatype definition for the element of the tree structure becomes either Composite or Leaf class of Composite pattern. The valuation function is mapped to a visitor, each method of which is responsible for manipulating a particular Composite or Leaf and corresponds to a clause of the valuation function.

In order to facilitate further program development, two off-line support tools are used in implementing the constraint language processor: JavaCC [137](a public domain Java-based parser generator) and ASTG (a visitor-skeleton generator). ASTG accepts BNF (Bachus Normal Form) descriptions of abstract syntax similar to the one in [117], and generates Java class definitions implementing both the AST node objects and skeleton codes for tree-walking. The skeleton code follows a convention of Visitor design pattern. Completing the program is not difficult because the program code fragment that need to be written in the body part of the skeleton corresponds to the clauses of the valuation functions.

Figure 6.8 shows the object-oriented framework that implements the federation process. This subsystem is an example of using the collaboration-based design technique with an algorithmic representation of the aspect design solution as its input specification. The collaboration-based design focuses on the analysis of the interaction patterns or a sequence of messages between objects, and the algorithm expressed in the functional programming style can be considered to provide an abstract view of a sequence of messages between (potential) participant objects.

The design step involves (1) identifying participant objects in a heuristic manner, and (2) determining responsibility of each object [9][20] to be consistent with the algorithm description. The algorithm description is homogeneous in that all the constituents are functions (sub-functions). What needs done is to make some of the functions to be class, while others to be methods of the identified class. A question arises what is a general guideline to make a function to be either class or method. Unfortunately, non-systematic way is found up to now. It, however, is related to identifying *hot-spot* [98], which is a portion of the program to be customized later. To put it differently, this process involves an important human design decisions on the customizability of the resultant framework.

The algorithm described by the function `federation(R)` is divided into two classes `Federation` and `Delegate` because the two have distinctive roles in view of

structural organization. Class `Federation` is responsible for controlling the whole federation process and thus plays the role of Facade [42], which decouples the federation subsystem from the rest of the program and thus makes it easy for testing. Class `Delegate` corresponds to the body of the function `traversal(J,R)` and thus implements details of the algorithm. Further, class `Delegate` encapsulates the implementation of the function `dispatch_on`, which implies that detailed implementation for invocating remote traders and collecting offers from them is localized in one class. Another example of the important design decision is encapsulating FollowOption calculation functions in class `FollowOption`, which aims to allow future customization of the policy.

6.5 An Alternative Design

6.5.1 Motivation

In order to discuss the role of the aspect-centered design method using CafeOBJ in Section 6.3, this section outlines how one conducts a design task for the case of the ODP/OMG trader with a conventional object-oriented modeling method. As the conventional method, one may use Catalysis [30], which is one of the latest object-oriented modeling methods that has the scenario concept. Since the specification of the ODP/OMG trader is large, the discussion is restricted to a portion only. It, however, covers all the important aspects.

As discussed in Section 6.2.2, the concept of *objects* plays a central role in any object-oriented modeling method. Object is a sole constituent of the system, and thus the modeling method is homogeneous. One can represent a complex system as a composite of well-defined small objects, which helps clarify a structural organization of the target system and leads to a situation where functions or constraints are attached to a particular object. It may, however, sometimes result in an awkward design that sees a lot of fragmentations. One cannot easily grasp a whole functionality at a glance. In the exercise, the following three points are the primary foci.

1. Expressibility

   The question is whether Catalysis together with UML provides notations for what to be represented.

2. Fragmentation

The question is whether one can easily grasp a whole functionality of the system.

3. Abstraction Level

   The question is whether Catalysis provides adequate abstraction levels so that one can represent his/her design without going into too much details.

6.5.2 Catalysis and the Recommendation Document

Catalysis addresses three levels of modeling and provides several *process patterns* as guidelines to proceed the modeling activities. The three levels are;

1. The Problem Domain or Business Process

   The activity at the level is to identify the problem outside of the system. It covers all concepts of the environment in which the system will be deployed.

2. The Component or System Specification

   The level primarily concerns with determining externally visible behavior of the system or component (a constituent of the system). It includes specification of component operations.

3. The Internal Design of the Component or System

   The last level is to focus on elaborating the internal organization of the component. It needs decisions for a specific implementation.

The ODP trader recommendation document [131] covers the first and the second level. The enterprise viewpoint specification corresponds to the the first level. The ODP computational viewpoint and the OMG specification [134] presents functionalities of the trading server as seen from clients of the trader. The document is an informal presentation of the interface of the component (the trading server), and thus can be said to provide informal descriptions for the second level. The ODP information viewpoint specification, however, is situated between the two and provides mostly declarative descriptions of the trader as a centralized system. It presents an alternative view of the trader specification but has little relationship with actual systems.

The main task for the modeling with Catalysis may start with the ODP enterprise specification for the first level, and then go to the second level by formalizing what

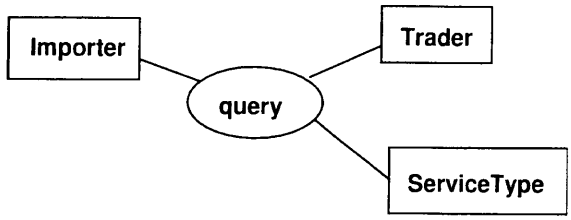


Figure 6.9: Trader in Business Model

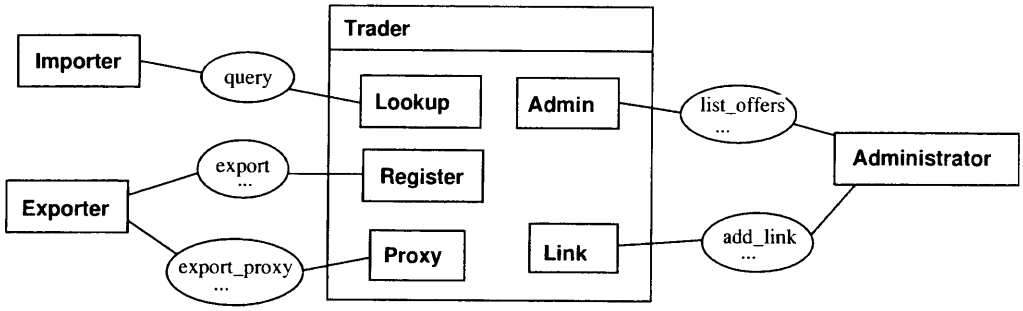


Figure 6.10: Type Model of Trader

the computational specification explains informally. The result of the activities is a set of design documents written in UML.

6.5.3 Designs

One may follow a process pattern for a case of building a design starting from scratch (Catalysis: Pattern 13.1), which is used when no reusable components or legacy systems available.

According to the ODP enterprise specification, one may describe an environment in which the trader is in play (Catalysis: Pattern 14.2). The result is Figure 6.9 to illustrate that an action (`query`) relates to three objects (`Trader`, `Importer`, and `ServiceType`). One may read this diagram as a `Trader` responds to a `query` action invoked by an `Importer` and they share `ServiceType` as a common data.

The initial model (Figure 6.9) is then enhanced and refined to Figure 6.10 by taking into account all the IDL interface specification of the computational view-

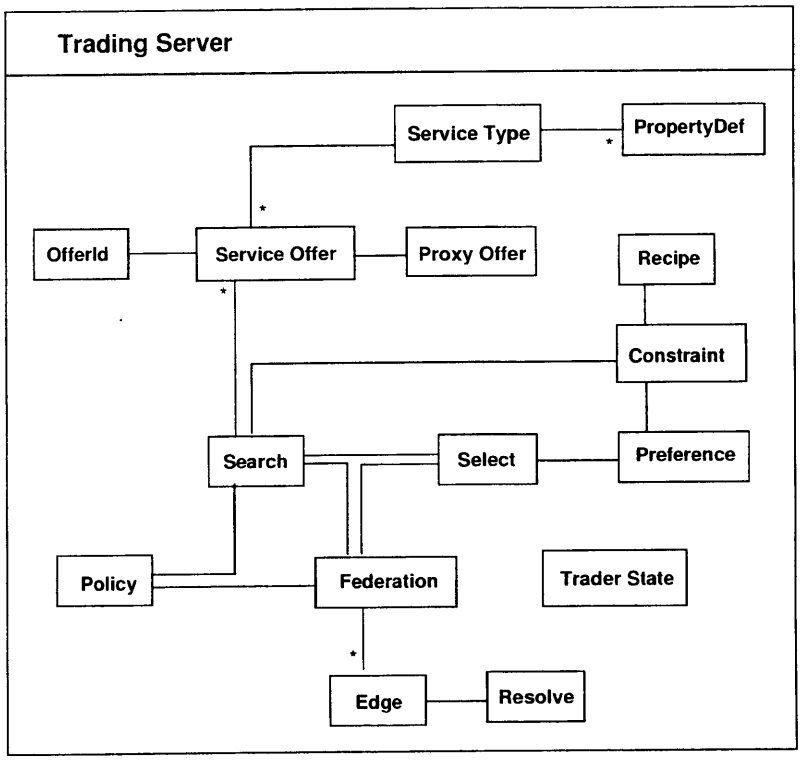


Figure 6.11: Trading Server Type Model

point. The figure shows that the trader consists of five functional objects, each of which corresponds to an IDL interface definition. One introduces three roles for clients of the trader (`Importer`, `Exporter`, and `Administrator`) in order to clarify the objectives of each interface operation. Figure 6.10 may complete the first level descriptions and bridge to the second level.

One proceeds to the second level and constructs a system behavior specification (Catalysis: Pattern 15.7). A main activity is to define a type specification of the trading server.[7] What needs to describe is the externally visible behavior of the trading server, of which informal specifications can be found in the ODP computational viewpoint and OMG trading service specification documents.

[7]The `Trader` in the first level is changed to `Trading Server` in order to emphasize that the latter is a system to be designed while the former is a term in the business model.

Figure 6.11 illustrates an initial type model of the trading server and static type models that are necessary for describing *externally* visible behavior of the server. The type model shows that the `Trading Server` consists of fifteen type models. Each type model is extracted from the recommendation document as a *logical* object necessary to understand the externally visible behavior of the `Trading Server`. The model also illustrates static relationships between the constituent type models. For example, every `Service Offer` belongs to a `Service Type`, and the `Service Offers` are fed into the `Search` which collects `Service Offers` that satisfy the `Constraint` condition given by a client of the `Trading Server`.

The next activity is to refine the initial specification of Figure 6.11 (Catalysis: Pattern 15.12). Refinement is conducted in several directions. First, one may focus on a refinement of operation specifications. The `query` operation is used as an example here.

Figure 6.12 illustrates a static type model necessary for explaining the behavior of the `query` operation. The diagram is obtained by *superposing* Figure 6.10 on Figure 6.11 with a slight modification in order to be consistent with an event trace diagram in Figure 6.13. The event trace diagram shows how the participant objects exchange messages, and describes the behavior of the `query` operation in terms of the identified objects. For example, a `Lookup` object accepts a `query` request from an `Importer`, and then invokes other objects to conduct validity checks on the input information by sending appropriate `check` messages. After invoking several functional objects such as `Search`, `Federation`, and `Select`, the `Lookup` sends back the final result (`Service Offers`) to the `Importer`. In the course of accomplishing the functionality of the `query` operation, the constituent objects send messages to each other as depicted in Figure 6.13. One needs to construct similar collaboration diagrams for all the sixteen operations defined in the IDL interface of the trader. Note that the event trace diagram shows a global flow of control, but is restricted to a single instance of execution trace only. It is not a rigorous description of an algorithmic aspect.

Second important area in regard to the refinement is to describe a static type model for concepts specific to the trading server. The basic concepts include `Service Type` and `Offer`, which the recommendation document explains in enough details. Thus, one can readily obtain a static type model as shown in Figure 6.14 that illustrates static relationships between the primary data definitions. Of the type model in Figure 6.14, `Mode` needs further refinement. The `Mode` is declared as an IDL `enum` that has four constants. And the recommendation document defines a strength relationship (a partial order) among them. One can represent these two

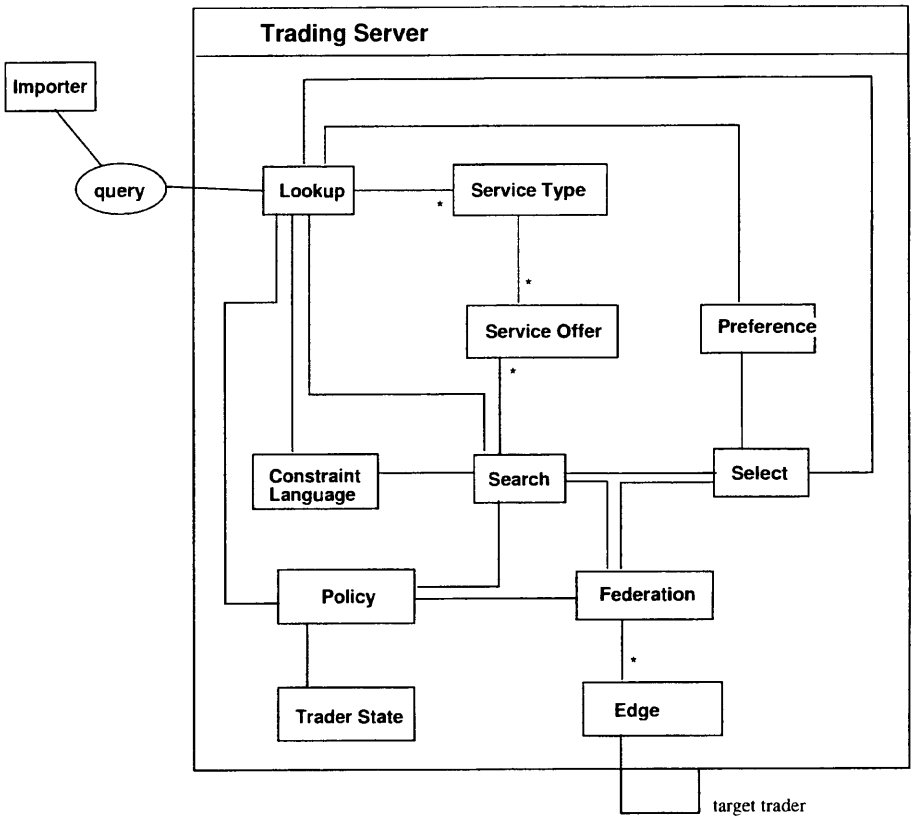


Figure 6.12: Collaboration for Query : Static Type Model

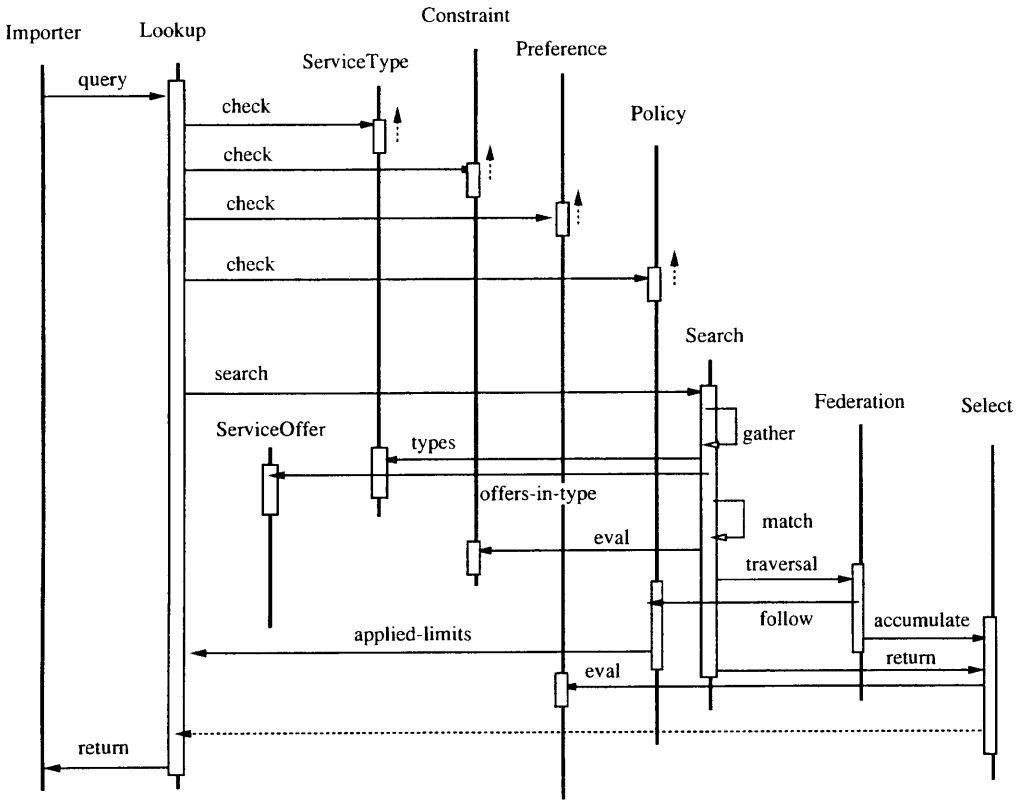


Figure 6.13: Collaboration for Query : Event Sequence

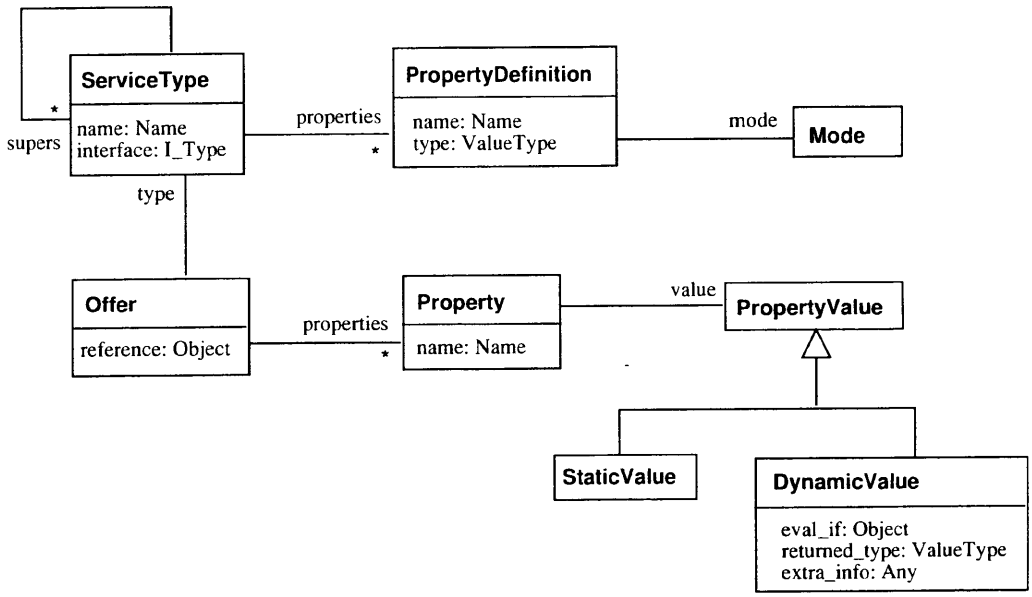


Figure 6.14: Static Type Model for Basic Concepts

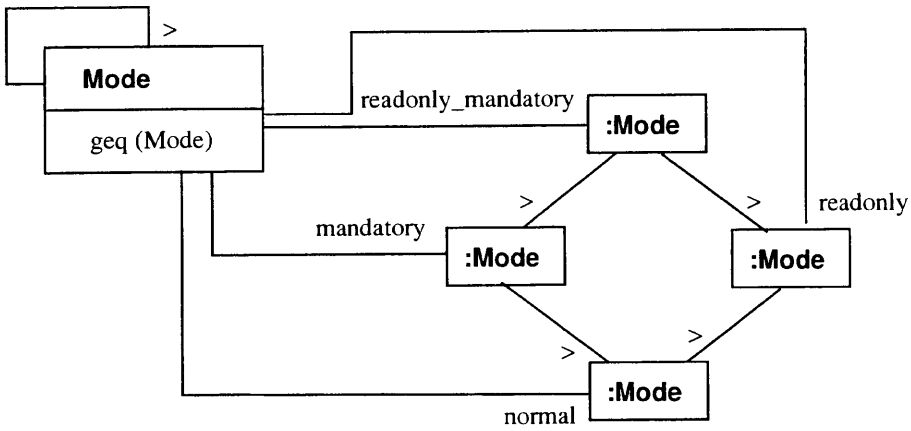


Figure 6.15: Mode Type Model

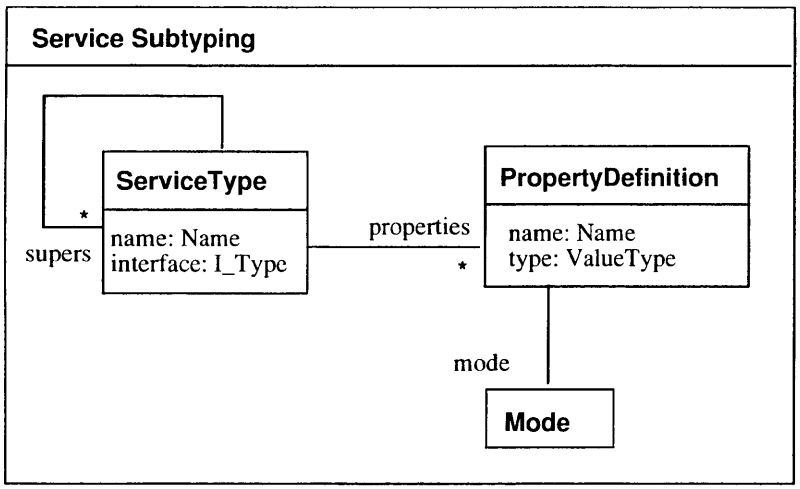


Figure 6.16: Subtyping Type Model

aspects of `Mode` type model by using *mixing* model of Catalysis (Figure 6.15).

Not everything can be represented in diagram notations only. A typical example is a subtyping relationship between service type instances that the recommendation document defines. One needs OCL to express such constraint conditions and the OCL expression should be accompanied by a related type model. In the case of subtyping relationship, a static type model in Figure 6.16 provides a context for the OCL expressions below.

The service subtyping relationship is placed on a service type instance (`self`) and its super-types kept in the `supers` attribute.

```
inv
---
self.supers->forAll(super : ServiceType
  |    self.interface->subtypeOf(super.interface)
   and pnames(self.properties)->includeAll(pnames(super.properties))
   and pnames(super.properties)->forAll(n : Name ;
          ap : PropertyDefinition = lookup_name(super.properties, n) ;
          bp : PropertyDefinition = lookup_name(self.properties, n)
        | bp.mode->geq(ap.mode) and ap.type = bp.type)
)
```

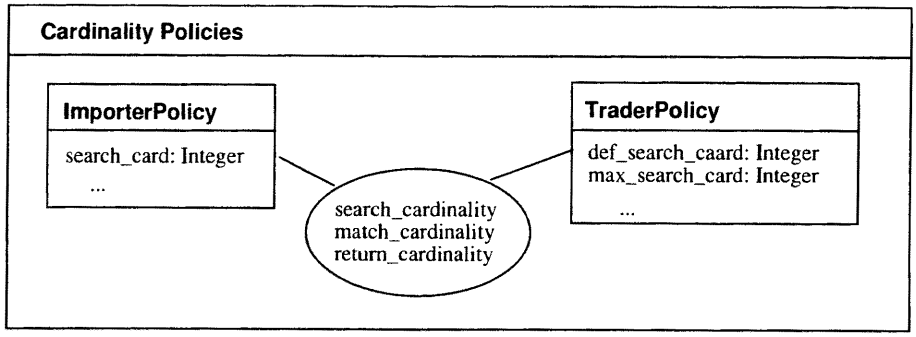


Figure 6.17: Policy Type Model

In order that the above invariant relationship is represented in a compact manner, one introduces two auxiliary functions. The division of labour between them is in accordance with the presentation of the recommendation document. The first one `pnames` collects all the `Names` appeared in the `properties`. The function `lookup_name` returns a `PropertyDefinition` that has a specified `Name` value. The function is accompanied with an invariant condition stating that there is no duplicate of `PropertyDefinition` instance in the set.

```
function pnames (ps : Set(PropertyDefinition)) : Set(Name)
-------------------------------------------------
post: result = (ps->collect(p | p.name))->asSet

function lookup_name (ps : Set(PropertyDefinition), n : Name)
         : PropertyDefinition
-------------------------------------------------------------
post: ((ps->Collect(p | p.name = n))->includes(result)

inv   (ps->Collect(p | p.name = n))->size = 1
---
```

As discussed in Section 6.3, *policy* is an important as well as an interesting concept to customize externally visible behavior of the trading server. One needs to have design descriptions for policies, which is again not adequate for diagram notations. Figure 6.17 defines a context for the cardinality policies. The constituents are `ImporterPolicy` and `TraderPolicy`, both of which are just data definitions. The

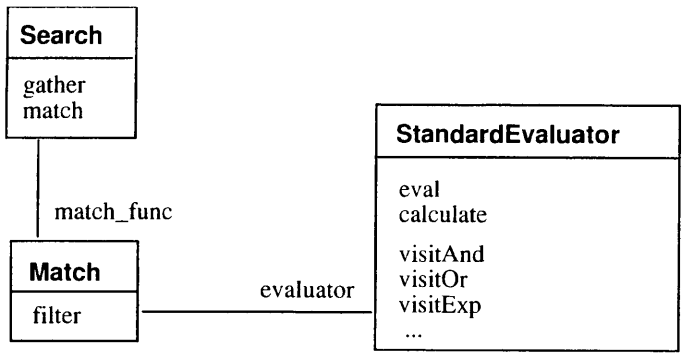


Figure 6.18: Static Type Model for Constraint Language Evaluator

OCL expression below is a direct transcription of one appeared in the recommendation document.

```
function search_cardinality (ip : ImporterPolicy, tp : TraderPolicy)
         : Integer
---------------------------
post: result
    = if exist(ip.search_card)
         then ip.search_card.min(tp.max_search_card)
         else tp.def_search_card
      endif
```

In regard to component construction or programming activities, the component for manipulating the query language (the constraint language) requires serious elaboration. The primary component is `StandardEvaluator` in Figure 6.18. The basic algorithm to obtain offers that satisfy the required condition can, in principle, be represented in OCL expression.

The method `filter` of class `Match` is a main routine for controlling the evaluation process. The algorithm is to eliminate such offers from an initial candidate set (`Set(Offer)`) that do not satisfy the given constraint (`And`). And the control is an iteration over the input sequence of constraint conditions (`Seq(And)`).

```
filter ( i : Set(Offer), c : Seq(And) ) : Boolean
-------------------------------------
```

```
c->iterate( a : And ; os : Set(Offer) = i
       | os = os->forAll( o : Offer | evaluator->eval(o, a)))
```

The class `StandardEvaluator` provides an interpreter body. The `eval` method is used in `filter` mentioned above, and the `calculate` method is an evaluator for a preference calculation (see (♯9) in Section 6.3.2). Their internal structure is to follow a style of the Visitor design pattern. And their body is just a call to an appropriate visitor method (see Section 6.4).

```
eval( o : Offer, a : And ) : Boolean
-------------------------------------
post:  result = self->visitAnd(a, o)

calculate( o : Offer, e : Exp ) : ValueType
-------------------------------------------
post:  result = self->visitExp(e, o)
```

Therefore, the design of the visitor method is a major concern, which may involve definitions of grammatical aspects (abstract syntax) and semantics of each abstract syntax element.

As presented in Section 6.3, however, this aspect is best discussed in terms of a standard language definition technique. Since an important characteristic of the grammatical aspect is a tree-like structure, one can represent, for example, the abstract syntax as a collection of related classes by using Composite pattern. However, the resultant static type model is inevitably large (the number of object definition is 25). Further the type model corresponds to Java class following Composite pattern and is not appropriately abstract. In fact, in Catalysis, consideration on design pattern is done only at the third level (Catalysis: Section 16.2). Design descriptions for the constraint language is somewhat less abstract than what is desirable. Catalysis/UML does not provide adequate notations for compactly and abstractly describing the design of the constraint language.

6.5.4 Summary

The above exercise leads to the following comparison between the proposed aspect-centered design method and a conventional modeling method following Catalysis.

The objective of the modeling activity is to have rigorous design descriptions of what the ODP/OMG recommendation document informally presents. The resultant

design should be abstract and compact so that one can grasp a whole picture at a glance. It also acts as a basis for further development activities. What follows is an examination of the design in regard to the three points raised in Section 6.5.1.

1. Expressibility

   One finds that Catalysis/UML provides various notations so that almost all the design can be represented. However, a global flow of control is bizarre in Calalysis/UML, while an algorithmic representation is appropriately abstract and compact. A notable example is a use of an event trace diagram to represent collaborations such as one for `query` (Figures 6.12 and 6.13). The flow shows a single instance of execution trace only.

2. Fragmentation

   In order to have rigorous descriptions written in OCL, one has to specify a context in which the OCL expression is evaluated. It requires that (1) one has to identify necessary objects and their static type models, and that (2) one writes an OCL expression in regard to the identified objects (Figures 6.16 and 6.17). The resultant design descriptions tend to be separated, and the whole specification becomes fragmented.

3. Abstraction Level

   Owing to various diagram notations and a textual one (OCL), one can represent designs at various abstraction levels in Catalysis/UML. Figure 6.15, using a mixing model, is intuitively understandable. Figure 6.16 with the invariant for the subtyping relationship is an example of using both a diagram notation and an OCL expression. However, as for the case of the constraint language, Catalysis/UML does not provide adequate notations for compactly and abstractly describing the essential design.

In summary, the proposed modeling method puts emphasis on grasping a whole picture at a glance and uses a single language CafeOBJ so that various aspects can be reasoned about. On the other hand, Catalysis/UML promotes to construct designs centered around *objects* and the resultant descriptions tend to be fragmented. From an alternative viewpoint, one can use Catalysis/UML after the proposed modeling method in such a way that one follows the Catalysis/UML method to bridge the gap between the aspect solutions (Section 6.3) and the final object-oriented frameworks written in Java (Section 6.4). In the development process of Section 6.2, we did

not follow such a systematic process but jumped to the object-oriented frameworks heuristically, which utilized accumulated *know-how*.

6.6 Discussions

This chapter presented a case that used CafeOBJ in the development of object-oriented frameworks. The work is unique in that none has been published that extensively addressed the two technology: the algebraic specification technology and object-oriented frameworks. A key idea is a problem-oriented modeling method (the aspect design), which helps decompose the the ODP/OMG trading server specifications into a set of design aspects. The aspect solutions form a clear input specification to the framework design phase where existing methods such as collaboration-base design and design patterns are effective.

Some quantitative summary follows. The aspect solution description is concise and thus expected to ease future maintenance.[8] For example, the seventeen functions in Section 6.3.2 abstractly describe the design of about 30 Java classes. And about thirty lines of the denotational style language description becomes more than 4K lines of Java code including AST classes and visitor skeletons that ASTG automatically generated. The current prototype implementation of the trading service server consists of some 380 Java classes, and its code size is about 25K lines.[9]

Further, Section 6.5 presents an alternative design of the trading server by following the Catalysis method. The aspect-centered design method puts emphasis on grasping a whole picture of the design at a glance and represents design descriptions in an abstract and compact manner.

The contribution of this work is twofold although two are strongly related. One area is in a practice of algebraic specification language and another is in a problem-oriented design method for object-oriented frameworks. The contribution for the first area is summarized below.

1. Expressibility

   The six identified aspects in Table 6.1 have intrinsically varied computational models. As shown in Section 6.3.3, one can encode all the aspect solutions in

[8]The size is not constant because we maintain and update the program code periodically. The information here is only meant to illustrate the system size.

[9]The current prototype implements a linked trader which consists of **Lookup**, **Register**, **Admin**, and **Link** interfaces. It omits the **Proxy** interface.

CafeOBJ, which confirms that CafeOBJ is expressive enough to represent all the interesting design specifications used in the proposed modeling method.

2. Module Decomposition

   The basic idea of the aspect-centered design is to break a whole problem into a set of manageable subproblems that can be solved individually. The resultant aspect solutions consist of CafeOBJ modules, each of which has a well-defined role in the overall specification. And the design method helps identify the modules. In a word, the proposed design method provides a problem-oriented guideline for CafeOBJ module decomposition.

3. Use in Development Process

   CafeOBJ/CAFE is used as a design checker in an iterative style of object-oriented framework development (Figure 6.1). The parallel iterative development model [120] is adequate in an integration of CafeOBJ with conventional modeling methods because various design activities can be done in any meaningful order.

On Aspect-Centered Design

A main contribution of this work to the area of object-oriented design method is the idea of the aspect-centered design. Its basic idea is to break a whole problem into a set of manageable subproblems that can be solved individually.

The aspect solution establishes clear relationships between the design descriptions at the different levels. First, the stream-style description does not have a gap with the ODP/OMG document, and, thus, both descriptions are quite traceable. It is easy to perform conformance checking during the design review. Second, collaboration, which is the most important view of frameworks, is basically a set of global interaction patterns and requires a concise notation for grasping the global flow of control. Algorithm description using the functional programming style is a good candidate for such a representation.

Regarding to the proposed design method, three research areas are identified: (1) discovering appropriate design aspects, (2) checking integrity of all aspect solutions, and (3) mapping the aspect solutions to object-oriented frameworks.

A hard part of the aspect-centered design method is lack of systematic methodology to discover appropriate aspects in a given problem. Since each aspect is accompanied with a specific specification technique, knowing specification techniques

as many as possible is helpful in identifying aspects in the problem. Accumulating various specification techniques and experience with their application to system development is important. The experience is supposed to derive Problem Frame of Jackson [64].

The idea of aspect-centered design is essentially decomposing a whole problem into a set of manageable subproblems to solve individually. Each aspect has its own notation such as functional-style descriptions or message sequence charts, and is amenable to validate separately. On the other hand, the design of the overall system requires to integrate all the solution descriptions, and this is difficult when each aspect solution uses a different notation. It requires a homogeneous notation. In the present work, CafeOBJ is employed because the language is powerful enough to cover all the aspect solution descriptions, from functional programming descriptions and abstract datatypes to message sequence charts.

The third point, mapping the aspect solutions to object-oriented frameworks, also needs some form of systematic method. Unfortunately this work did not follow such a systematic process but jumped to the object-oriented frameworks heuristically, which utilized accumulated *know-how*. As discussed in Section 6.5.4, one may use Catalysis to go into the object-oriented framework design stage by using the aspect solutions as the input design descriptions.

Role of CafeOBJ Descriptions

As explained above, CafeOBJ is used to write each aspect solution. And thus one can use CafeOBJ/CAFE as a design validation checker. First, one can get benefits from syntax and sort checking. This helps to identify what is missing and what are syntactically inconsistent. Second, with appropriate input test terms, one can validate the design by test execution (specification simulation). It also helps uncover logical inconsistency spreading over different aspect solutions.

The CafeOBJ descriptions themselves form an artifact. It is an *analogic* model [64] in the sense that the artifact does not represent the Java program faithfully, but instead elucidates essential design in an abstract manner. One may use the model as a reusable design artifact when developing a product line [76] (a family of systems having similar functionalities) in future.

One of methodological advantages of CafeOBJ is to provide hidden algebra or behavioral equations[29][40][49]. Specifications using hidden algebra are basically state-machines, where sequences of allowable *events* describe the properties of the state-machine without defining the constituting states explicitly. The technique is

adequate when certain properties are verified by using observational equivalence. Contrarily, writing specifications in a constructive and operational manner is significant in the present development process. The CafeOBJ descriptions focus on the design aspects and the solutions such as the query and federation algorithm in detail. Such a detailed description acts as a starting point for the further refinement and elaboration. In summary, the CafeOBJ description is an *abstract* implementation in the present approach. It is worth investigating to compare the pros and cons of the two approaches (the present one and the hidden algebra approach) and to study their roles in the development process of object-oriented frameworks.

Management Issues

The main development process consisted of three steps: the aspect design, the object-oriented design, and the coding and testing. The aspect design step started with the study of the system requirements and ended with the semi-formal description of each aspects. The object-oriented design step was one in which the collaboration-based object-oriented design method and the design pattern technique were used to produce design documents describing specifications of Java classes. It was followed by the coding and testing. We assigned one person (this author) to the aspect design and two engineers to the coding and testing. All three persons worked together to produce the design documents at the intermediate step. The engineers were not familiar with the collaboration-based object-oriented design technique and required *on the job* training. The object-oriented design step involved technology transfer, and took far longer than initially planned. The time needed for coding and testing, however, was short for a program of this size.

We started using CafeOBJ only when we almost reached a code complete stage of the first working prototype. It is partly because we had to finish most of the coding work as early as possible due to the time constraint of the financial support. We used the CafeOBJ descriptions to check the conformance of the informal design and implementation against the ODP document.

Additionally, we think that most of the engineers would not accept CafeOBJ descriptions as their input source information because the engineers resisted even the semi-formal descriptions of the aspect design. Proper division of labor between project members is thus important when formal methods are used in real world projects.

As for an integration of conventional design methods and formal methods, the experience is somewhat different from the *folklore* in the research community. The

folklore says that one may follow some conventional design methods such as scenario-based object-oriented modeling to understand the domain or the problem and then one proceeds to use formal methods for rigorous development of software system with the result of the modeling as the input specification.

The experience does not support the folklore, but supports the use of formal method (CafeOBJ) for analyzing and understanding the problem. And then the development proceeds to use conventional object-oriented design methods or design patterns to elaborate the object-oriented framework design. This may be due to the fact that an essence of designing object-oriented framework is to relate functional units to structural components. It requires first to fully understand the functional aspects and then to find appropriate mappings of each functional unit to a particular class construct. In searching the mappings, one has to concentrate on the analysis of the participant object collaborations and of forecasts on possible future customization (*hot spots*). The process requires expert insights and no systematic basis is yet found.

Putting emphasis on the structural aspect of the software system is important also from a viewpoint of managing the development process in the industrial settings. The management for the coding and testing phase is not so time-consuming if all the engineers and the managing person have common understanding on the structural organization of the program. It is because managerial decisions relating to finish a certain phase and to proceed to another are usually done based on the structural unit.

In summary, the experience elucidates some interesting issues in relation to using formal methods (CafeOBJ) in developing object-oriented frameworks in the industrial settings. One has to consider realistic constraints such as the followings; (1) few engineers can understand formal specification, (2) time constraint on the project is a major issue (which has a great impact on the development process), and (3) structural organization is as important as functional properties in view of future customization and maintenance.

Related Work

As stated early, the present work is unique in that one can find no other published research in integration of the algebraic specification technology with object-oriented frameworks.

Wirsing integrates an algebraic specification technology with scenario-based object-oriented modeling [124][125]. He, however, does not mention an application to

object-oriented framework development.

Jackson discusses that methods should be related to a particular class of problem and thus give a sharply focused help in reaching a solution [64]. He suggests an importance of problem-oriented method, which leads to an idea of *Problem Frame*, but does not propose a specific modeling method in the usual sense. The aspect-centered design is a practice following his philosophy.

In using FDT tool as a design validation checker at an early stage of software development, the case studies at NASA [32] are similar to the present work. They use PVS [103] as a tool for validating properties of requirement models described in OMT. Our work uses CafeOBJ as a checker for design descriptions of the aspect solutions. The design is multiparadigm, not restricted to the object-orientation. Further, our work covers the whole life-cycle of object-oriented framework development.

Chapter 7

Contributions and Future Work

This chapter summarizes the main technical contributions of the dissertation and discusses some of further research themes.

7.1 Contributions

Object Technology (OT) and Algebraic Description Techniques (ADT) are generally accepted as two main streams for improving productivity and quality of software system. The convergence of the two, however, has not been so successful that few engineers in industry can use both technologies in their daily work. It is because OT consists of a wide variety of concepts and tools that a naive algebraic approach to OT cannot deal with properly. These include (1) object-oriented paradigm and programming, (2) high-level reuse concepts such as object-oriented framework, and (3) object-oriented modeling methods. In order to put the two technologies into work in industry, this dissertation showed a way of integrating CafeOBJ in the development process of object-oriented frameworks. In a word, the dissertation discussed a new algebraic approach to object-oriented software development method with CafeOBJ. The method follows the understanding of the object-oriented software technology, which is based on the author's own industrial experience.

A main contribution of the dissertation is in the area of a practical use of the algebraic specification technique. The dissertation shows that the technology, CafeOBJ in particular, is an industrial-strength formal description technique. With four medium scale case studies, the discussion addressed three issues with CafeOBJ (Section 1.2): (1) expressiveness, (2) module decomposition, and (3) use in develop-

ment process. Four cases are (1) Maude/Ambient (Chapter 3), (2) GILO (Chapter 4), (3) ODP Trader Recommendation (Chapter 5), and (4) OO-FWK Development (Chapter 6). These cover the most important four different aspects in relation to applying CafeOBJ to development of object-oriented frameworks.

1. Expressiveness

   Chapter 3 presents a way of encoding object-oriented algebraic specification in CafeOBJ. A novel contribution is an integration of the Maude concurrent object model with an idea of the Ambient calculus. The integration brings a new machinery to handle a set of strongly related objects as one unit.

   Chapter 4 proposes a new intermediate design notation GILO, which bridges a gap between scenario-based modeling method and object-oriented algebraic specification in CafeOBJ. The modeling method with GILO is effective in obtaining executable CafeOBJ specifications in accordance with the scenario-based modeling technique.

   Chapter 5 illustrates that CafeOBJ is expressive enough to represent both the information and computational viewpoint specifications of the ODP trader. It is a counter example of the folklore [34] saying that no single FDT language can specify the richness of the ODP trader.

   Chapter 6 shows that one can encode all the interesting design aspect solutions in CafeOBJ. Each of the design aspect sheds light on a particular characteristic of the whole design of the ODP trading server.

   From a viewpoint of representing various computational models, one sees, through concrete examples, that CafeOBJ can encode abstract datatypes, functional programming, concurrent objects, procedural methods (state-based model), transition systems, and constraint relationships. All the computational models are important in describing design artifacts of object-oriented software.

2. Module Decomposition

   The case studies present both problem-independent and problem-oriented guidelines for module decomposition. The experience is a basis for scaling up the CafeOBJ-based technology.

   The modeling method with GILO in Chapter 4 facilitates finding and defining CafeOBJ modules because each module has a specific role (either collabora-

tion, class, or common vocabulary). And the resultant CafeOBJ specification is clearly traceable from an entity in the problem domain space because the specification is in accordance with the scenario-based problem analysis method.

Contrary to the case of Chapter 4, which discusses a general-purpose (problem-independent) method for module decomposition, a primary characteristic of Chapters 5 and 6 is problem-oriented guideline. Focusing on a particular problem is important because one can enjoy a sharply focused help in reaching the solutions [64].

The case study in Chapter 5 shows that the structure of the ODP trader recommendation document acts as a good guideline to derive a unit of modules. One can identify and define CafeOBJ modules with the proposed method focusing on design aspects as a decomposition guideline in Chapter 6. Both guidelines can be considered as problem-oriented heuristics to decompose the problem.

3. Use in Development Process

   The issue relating to the development process is difficult to discuss in a general manner. It is because human- or project-specific problems are sometimes the main concern and thus one finds it hard to discuss objectively. The issue, however, has much impact on using a new technology in industry, and thus cannot be neglected.

   The modeling method with GILO in Chapter 4 is an integration of CafeOBJ with the conventional scenario-based object-oriented modeling methods. One assumes a development process such that one uses the scenario-based modeling method to analyze a given problem and then obtains executable CafeOBJ descriptions. Thus, the use of CafeOBJ comes at an object-oriented analysis phase. That the resultant CafeOBJ specifications are executable is unique compared with related works.

   The discussion in Chapter 6 covers a whole development life-cycle, which is a novel contribution of this work to the field of FDT. The whole development process is the iterative one. The aspect-design stage comes at the first step, whose aim is to have a global design of the system before starting the iterative development cycle. Then, one makes use of existing technology such as design patterns to construct final object-oriented frameworks. CafeOBJ is employed

to encode the resultant global design (the aspect solutions), which is in principle validated with the CAFE environment. Thus, the case shows a way of using CafeOBJ/CAFE as a checker for a global design before proceeding to the program construction. In addition, discussions in Chapter 6 includes a viewpoint on managing the process, which is important in industry but often neglected in the academic community.

Chapter 5 makes a contribution to another area in which FDT plays a central role. The ITU-T recommendation document has been a promising area to which FTDs are applied, because unambiguous and consistent descriptions are necessary as a standard document. The CafeOBJ description of the ODP trader is a first attempt of the algebraic approach to describing both the information and computational viewpoints of the ODP trader. The work shows that CafeOBJ can be used as a FDT language for the recommendation document. The discussion also includes how one gets benefit from CafeOBJ/CAFE in order to understand the constructed model.

7.2 Future Work

This section presents a list of future works in relation to applying the algebraic specification techniques to the object technology.

First, as discussed in Section 6.6, one of the methodological advantages of CafeOBJ is to provide hidden algebra or behavioral equations [29][40][49]. The present dissertation did not consider this feature, and focused on using CafeOBJ as a language for abstract design in a constructive and operational manner.

Specifications using hidden algebra are basically state-machines, where sequences of allowable *events* describe the properties of the state-machine without defining the constituting states explicitly. The technique is adequate when certain properties are verified by using observational equivalence. In addition, Mori and Futatsugi [85] show that an automated theorem proving technique is effective in verifying behavioral specifications. It is worth investigating to study their roles in the development process.

Second, *Component* gets a wide acceptance in industry as a new technology for object-oriented software reuse. Notable examples are Sun Microsystem's Enterprise Java Beans (EJB), Microsoft's COM, and a new component model of OMG CORBA [133]. One can implement a component that encapsulates application logic and make it run with other existing components on the pre-defined infrastructure.

The core of the technology is the infrastructure that supports the basic information flow among the components running on its top, and that defines a set of well-established interfaces referred to as API. Thus, as long as conforming to the pre-defined API, one can easily deploy his own component in the computing environment. From the control flow viewpoint, the infrastructure can be considered to act as a *frozen spot* in the sense of the object-oriented framework. In other word, the component is plugged into the *hot spot* to complete the behavior as a whole.

However, it is not easy to ensure that the user defined components behave as the infrastructure expects. The components are required to behave properly even in an error situation. Otherwise, the system as well as the problematic component may crash down as a whole. Thus, a verification prior to component deployment is important. The verification involves the overall control flow taking into account the behavioral properties of both the infrastructure and the component. Since the control flow can be considered as a collaboration of the participants, one can clearly abstract the flow in terms of the state transition model, which is discussed in this dissertation. Then, an interesting observation is that the abstract state-machine technique of hidden algebra may be adequate for modeling such information flow, and thus that the verification can be done in the algebraic logic framework.

Third, integrating the proposed GILO method with the stepwise refinement is an important direction. As discussed in the history of FDTs, proof-assisted stepwise refinement is an old but still new challenge to develop high quality and accountable programs. In the context of constructing object-oriented specifications with CafeOBJ, Wirsing [124][125] has studied the problem. An observation is that FDT (CafeOBJ) alone does not suffice, but that total support for a thorough development process is necessary. Thus, integrating informal modeling techniques with FDT is essential.

Catalysis is an object-oriented modeling method that already has some notion of the stepwise refinement. One may integrate Catalysis with GILO/CafeOBJ for a start. It needs an algebraic formulation of Catalysis, which involves establishing a rigorous semantic basis for UML and OCL in an algebraic manner. And then, the stepwise refinement process may adopt the techniques reported in [124][125].

In closing this dissertation, I note that the research included here demonstrates that CafeOBJ, or algebraic specification techniques in general, is adequate for use to understand the ill-understood world in order to construct software systems. The use of the language is similar to one in the exploratory programming style of software development [107]. Since CafeOBJ is a mathematically-based specification language that has semantics more precise than those languages historically used in

exploratory programming, we can say that our development style is a modern and rigorous version of exploratory programming. Finally, to advance the technology, it is important to use it in the process of daily software development. I hope that the reported results will be a help in conducting further exercise.

Bibliography

[1] Agha, G. : *Actors : A Model of Concurrent Computation in Distributed System*, The MIT Pess 1986.

[2] Agha, G. et al : Abstraction and Modularity Mechanisms for Concurrent Computing, in *Research Directions in Concurrent Object-Oriented Programming (Agha, Wegner and Yonezawa ed.)*, pp.3-21, MIT Press 1993.

[3] Alencar, A. and Goguen, J. : OOZE: An Object Oriented Z Environment, Proc. ECOOP'91, pp.180-199 (1991).

[4] Alexander, C. : *A Pattern Language*, Oxford 1977.

[5] Arnold, K. and Gosling, J. : *The Java™ Programming Language*, Addison-Wesley 1996.

[6] Barsow, D.R., Schrobe, H.E., and Sandewall, E. : *Interactive Programming Environments*, McGraw-Hill 1984.

[7] Bass, L., Clements, P., and Kazman, R. : *Software Architecture in Practice*, Addison-Wesley 1998.

[8] Bearman, M. : Tutorial on ODP Trading Function, University of Canberra (1997).

[9] Beck, K., and Cunningham, W.: A Laboratory for Teaching Object Oriented Thinking, Proc. OOPSLA'89, pp.1-6 (1989).

[10] Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. : Meteor: A Successful Application of B in a Large Project, Proc. FM'99, pp.369-387 (1999).

[11] Beppu, Y., Nakajima, S., Kumeno, F., Cho, K., Hasegawa, T., Ohsuga, A. : A Directory Server for Mobile Agents Interoperability, Proc. IEEE EDOC 2000 (2000).

[12] Bernardeschi, Dustzadhe, J., Fanttechi, A., Najm, E., Nimour, A., and Olsen, F. : Transformation and Consistent Semantics for ODP Viewpoints, Proc. FMOODS'97 (1997).

[13] Berry, G. and Boudol, G. : The Chemical Abstract Machine, Theor. Comp. Sci. no.96, pp. 217-248 (1992).

[14] Booch, G. : Object-Oriented Development, IEEE Trans. Soft. Engin., vol. SE-12, no. 2, pp.211-221 (1986).

[15] Booch, G., Rumbaugh, J. and Hopkins, J. : *The Evolution of Object Methods*, Handout, Rational Software Corporation, 1995.

[16] Bowman, H, Derrick, J., Linington, P., and Steen, M.W.A. : FDTs for ODP, Computer Standards and Interfaces (17) pp.457-479 (1995).

[17] Bowman, H, Boiten, E.A., Derrick, J., and Steen, M.W.A. : Viewpoint Consistency in ODP, a General Interpretation, Proc. FMOODS'96 (1996).

[18] Cardelli, L. and Gordon, A.D. : Mobile Ambients, DEC SRC (1997).

[19] Carrington, D. et al : Object-Z : An Object-Oriented Extension to Z, Proc. FORTE'89, pp.281-296 (1990).

[20] Carroll, J.M. (ed.) : *Scenario-Based Design*, John Wiley & Sons 1995.

[21] Clarke, E.M. and Wing, J.M. : Formal Methods: State of the Art and Future Directions, ACM Computing Surveys (1997).

[22] Clavel, M., Eker, S., Lincoln, P., and Meseguer, J. : Principles of Maude, Proc. 1st Workshop RWL (1996).

[23] Coleman, D., Arnols, P., Bodoff, S., Dollin, C., Gilcchrist, H., Hayes, F., and Jeremaes, P. : *Object-Oriented Development : The Fusion Approach*, Prentice-Hall 1994.

[24] Cook, W. : Object-Oriented Programming Versus Abstract Data Types, in *Foundations of Object-Oriented Languages*, LNCS 489, pp.151-178 (1991).

[25] Dahl, O.-J. aand Nygaard, K. : SIMULA – an ALGOL-Based Simulation Language, CACM vol. 9, no. 9, pp.671-678 (1966).

[26] Deutsch, P. : Design Reuse and Frameworks in the Smalltalk-80 System, in *Software Reusability II, (Biggerstaff, T. and Perlis, A. ed.)*, IEEE Press 1987.

[27] Diaconescu, R. : Foundations of Behavioural Specification in Rewriting Logic, Proc. 1st Int. Workshop of Rewriting Logic and its Applications (1996).

[28] Diaconescu, R. and Futatsugi, K. : Logical Semantics for CafeOBJ, JAIST Research Report IS-RR-96-22S (1996).

[29] Diaconescu, R. and Futatsugi, K. : *The CafeOBJ Report*, World Scientific 1998.

[30] D'Souza, D.F. and Wills, A.C. : *Objects, Components, and Frameworks with UML*, Addison-Wesley 1998.

[31] Dustzadeh, J. and Najm, E. : Consistent Semantics for ODP Information and Computational Models, Proc. FORTE/PSTV'97 (1997).

[32] Easterbrook, S., Lutz, R., Kelly, J., Ampo, Y., and Hamilton, D. : Experiences Using Lightweight Formal Methods for Requirements Modeling, IEEE Trans. Soft. Engin., vol. SE-24, no. 1, pp.4-14 (1998).

[33] Fischbeck, N., Fischer, J., Holz, E., Lowis, M., Kath, O., and Schroder, R. : Improving the Development and Validation of Viewpoint Specifications, Proc. FMOODS'97 (1997).

[34] Fischer, J., Prinz, A. and Vogel, A. : Different FDT's Confronted with Different ODP-Viewpoints of the Trader, Proc. FME'93 pp. 332-350 (1993).

[35] Futatsugi, K., Goguen, J., Jouannaud, J-P., and Meseguer, J. : Principles of OBJ2, Proc. 12th POPL, pp.52-66 (1985).

[36] Futatsugi, K., Goguen, J., Meseguer, J. and Okada, K. : Parameterized Programming in OBJ2, Proc. 9th ICSE, pp.51-60 (1987).

[37] Futatsugi,K. : Trends in Formal Specification Methods based on Algebraic Specification Techniques – from Abstract Data Types to Software Processes: A Personal Perspective –, Proc. Info Japan '90, pp.59-66 (1990).

[38] Futatsugi, K. and Sawada, T. : Design Consideration for **CAFE** Specification Environment, Proc. The 10th Anniversary of OBJ2 (1995).

[39] Futatsugi, K. : Fundamentals for Algebraic Modeling (in Japanese), Computer Software, vol.13, no.1 pp.4-22 (1996).

[40] Futatsugi, K. and Nakagawa, A.T. : An Overview of CAFE Specification Environment, Proc. 1st IEEE ICFEM (1997).

[41] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. : Design Patterns: Abstraction and Reuse of Object-Oriented Design, Proc. ECOOP'93, pp.406-431 (1993).

[42] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. : *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994.

[43] Giovanni, R. and Iachini, P. : HOOD and Z for the Development of Complex Software Systems, Proc. VDM'90, pp.262-289 (1990).

[44] Goguen, J. and Meseguer, J. : Unifying Functional, Object-Oriented and Relational Programming in Logical Semantics, in *Research Directions in Object-Oriented Programming (Shriver and Wegner ed.)*, pp.417-477, MIT Press 1987.

[45] Goguen, J. : Principles of Parameterized Programming, in *Software Reusability*, pp.159-225, ACM Press 1989.

[46] Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J-P.: Introducing OBJ, SRI-CSL-92-03 (1992).

[47] Goguen, J. and Diaconescu, R. : An Oxford Survey of Order Sorted Algebra, Math. Struct. in Comp. Science, vol. 4, pp.363-392 (1994).

[48] Goguen, J. and Malcolm, G. : *Algebraic Semantics of Imperative Programs*, The MIT Press 1996.

[49] Goguen, J. and Malcolm, G. : A Hidden Agenda, UCSD CS97-538 (1997).

[50] Goguen, J. and Rosu, G. : Hiding More of Hidden Algebra, Proc. FM'99 (1999).

[51] Goldberg, A. and Robson, D. : *Smalltalk-80 : the language and its implementation*, Addison-Wesley 1983.

[52] Guttag, J. and Horning, J. : The Algebraic Specification of Abstract Data Types, Acta Informatica 10, pp.27-52 (1978).

[53] Guttag, J. and Horning, J. : *Larch: Languages and Tools for Formal Specification*, Springer-Verlag 1993.

[54] Hall, J. : Using Z as a Specification Calculus for Object-Oriented System, Proc. VDM'90, pp.290-318 (1990).

[55] Hayes, I. : Applying Formal Specification to Software Development in Industry, IEEE Trans. Soft. Engin. SE-11 (2), pp.169-178 (1985).

[56] Hayes, I. (ed.) : *Specification Case Studies (2ed.)*, Prentice Hall 1993.

[57] Helm, R., Holland, I., and Gangopadhyay, D. : Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, Proc. OOPSLA/ ECOOP'90, pp.169-180 (1990).

[58] Hewitt, C. and Smith, B. : Towards a Programming Apprentice, IEEE Trans. Soft. Engin., vol.SE-1, no. 1, pp.26-45 (1975).

[59] Hutt, T.F. (ed.) : *Object Analysis and Design : description of methods*, John Wiley & Sons 1994.

[60] Ingalls, D. : The Smalltalk-76 Programming System Design and Implementation, Proc. POPL'78, (1978).

[61] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. : Back to the Future, Proc. OOPSLA'97, pp.318-326 (1997).

[62] Inverardi, P. and Wolf, A.L. : Formal Specification and Analysis of Software Architecture : Using the Chemical Abstract Machine Model, IEEE Trans. Soft. Engin., vol.21, no.4, pp.373-386 (1995).

[63] Inverardi, P. and Yankelevich, D. : Relating CHAM Descriptions of Software Architecture, Proc. 8th IWSSD, pp.66-74 (1996).

[64] Jackson, M. : *Software Requirements & Specifications*, Addison-Wesley 1995.

[65] Jacobson, I. et al : *Object-Oriented Software Engineering*, Addison-Wesley 1992.

[66] Jensen, K. : *Coloured Petri Nets 1*, Springer-Verlag 1992.

[67] Johnson, R. : Documenting Frameworks using Patterns, Proc. OOPSLA'92, pp.63-76 (1992).

[68] Jones, C.B. : Scientific Decisions which Characterized VDM, Proc. FM'99, pp.28-47 (1999).

[69] Kay, A. and Goldberg, A. : Personal Dynamic Media, IEEE Computer, pp.31-41 (1977)

[70] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. : Aspect-Oriented Programming, Proc. ECOOP'97, pp.220-242 (1997).

[71] Kobryn, C. : UML 2001: A Standardization Odyssey, CACM Vo.42, no.10, pp.29-37 (1999).

[72] Lano, K. and Haughton, H. (ed) : *Object-Oriented Specification Case Studies*, Prentice Hall 1994.

[73] Lano, K. : *Formal Object-Oriented Development*, Springer-Verlag 1995.

[74] Leavens, G. and Cheon, Y. : Preliminary Design of Larch/C++, Proc. 1st Workshop on Larch, pp.159-184 (1993).

[75] Lecero, G. F. and Quemada, J. : Specifying the ODP trader in E-LOTOS, Proc. FORTE/PSTV'97 (1997).

[76] van der Linden, F.J. and Muller, J.K. : Creating Architecture with Building Blocks, IEEE Software, November pp.51-60 (1994).

[77] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. : Specifying Distributed Software Architectures, Proc. ESEC'95 (1995).

[78] Meseguer, J. : A Logical Theory of Concurrent Objects and its Realization in the Maude Language, in *Research Directions in Concurrent Object-Oriented Programming (Agha, Wegner and Yonezawa ed.)*, pp.314-390, MIT Press 1993.

[79] Meyer, W. : Taligent's CommonPoint : The Promise of Objects, IEEE Computer, pp.78-83 (1995).

[80] Miki, M., Tanaka, M., Tomobe, M., and Nakajima, S. : A Scripting Language for Network Management Applications and its Related Tool, IEEE GLOBECOM'97, pp.1714-1718 (1997).

[81] Mikkonen, T. : Formalizing Design Patterns, Proc. ICSE'98, pp.115-124 (1998).

[82] Milner, R., Tofte, M., Harper, R., and MacQueen, D. : *The Definition of Standard ML (revised)*, The MIT Press 1997.

[83] Milner, R. : *Communicating and Mobile Systems: the π-Calculus*, Cambridge 1999.

[84] Moreira, A. and Clark, R. : Combining Object-Oriented Analysis and Formal Description Techniques, Proc. ECOOP'94, pp.344-364 (1994).

[85] Mori, A. and Futatsugi, K. : Verifying Behavioural Specifications in CafeOBJ Environment, Proc. FM'99 (1999).

[86] Mowbray, T.J. and Zahavi, R. : *The Essential CORBA*, John Wiley & Sons, 1995.

[87] Nakajima, S. : Formalizing Object-Oriented Software with Algebraic Specification Techniques, in *Understanding Object-Model Concepts*, Bringham Young University, BYU-CS-93-12 (1993).

[88] Nakajima, S. : GILO/Z: An Extension of Z Notation for Object-Oriented Specification (in Japanese), Trans. IPS Japan, vol.36, no.5, pp.1059-1069 (1995).

[89] Nakajima, S. : Formal Notation for Scenario-Based Object-Oriented Design, Proc. The 10th Anniversary of OBJ2 (1995).

[90] Nakajima, S. and Futatsugi, K. : Constructing OBJ Specifications with Object-Oriented Design Methodology, Talk at 7th NEC Research Symposium (1996).

[91] Nakajima, S. : An Implementation of HORB-based Network Management System, Talk at IC-DISC, Imperial College (1996).

[92] Nakajima, S. and Futatsugi, K. : An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ, Proc. 19th ICSE, pp.34-44 (1997).

[93] Nakajima, S. : Encoding Mobility in CafeOBJ, Talk at CafeOBJ Symposium (1998).

[94] Nakajima, S. and Futatsugi, K. : An Algebraic Approach to Specification and Analysis of the ODP Trader, Trans. IPS Japan, vol.40, no.4, pp.1861-1873 (1999).

[95] Nakajima, S. and Futatsugi, K. : CafeOBJ Specifications of the ODP Trader (in Japanese), Computer Software, vol.16, no.4, pp.76-79 (1999).

[96] Nakajima, S. : Using Algebraic Specification Techniques in Development of Object-Oriented Frameworks, Proc. FM'99 (1999), also in *OBJ/CafeOBJ/Maude at Formal Methods '99* (Futatsugi, Goguen, Meseguer (eds.)), THETA 1999.

[97] Nakajima, S : An Aspect-Centered Design of Object-Oriented Frameworks, Trans. IPS Japan, vol.41, no.3, pp.758-766 (2000).

[98] Pree, W. : Meta Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design, Proc. ECOOP'94, pp.150-162 (1994).

[99] Raymond, K. : Reference Model of Open Distributed Processing (RM-ODP) : Introduction, Proc. ICODP'95 (1995).

[100] Rich, C. and Waters, R. : *The Programmer's Apprentice*, Addison-Wesley 1990.

[101] Riehle, D. and Gross, T. : Role Model Based Framework Design and Integration, Proc. OOPSLA'98, pp.117-133 (1998).

[102] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice-Hall 1991.

[103] Rushby, J. : Mechanized Formal Methods: Where Next?, Proc. FM'99, pp.48-51 (1999).

[104] Sawada, T. and Futatsugi, K. : CAFE as an Extensible Specification Environment, Proc. Kunming International CASE Symposium '94, pp.6A.1-6A.21 (1994).

[105] Sharble, R. and Cohen, S. : The Object-Oriented Brewery : A Comparison of Two Object-Oriented Development Methods, ACM SIGSOFT Soft. Engin. Notice 18 (2), pp.60-73 (1993).

[106] Shaw, M. and Garlan, D. : *Software Architecture*, Prentice-Hall 1996.

[107] Sheil, B. A. : Power Tools for Programmers, Datamation Magazine (1983) also in [6].

[108] Shock, J.F. : An Overview of the Programming Language Smalltalk-72.

[109] Sousa, J.P. and Garlan, D. : Formal Modeling of the Enterprise JavaBeans^{TM} Component Integration Framework, Proc. FM'99, pp.1281–1300 (1999).

[110] Spivey, J. : *The Z Notation (2ed edition)*, Prentice Hall 1992.

[111] Stepney, S, Barden, R. and Cooper, D. (ed) : *Object Orientation in Z*, Springer-Verlag 1992.

[112] Tamai, T. : How Modeling Methods Affect the Process of Architectural Design Decisions: A Comparative Study, Proc. IWSSD-8, pp.125-134 (1996).

[113] Tomono, M., Yamanaka, A., Tonouchi, T., and Nakajima, S. : An Implementation of Customizable Services with Java/ORB Integration, Proc. GLOBECOM'97 (1997).

[114] Tonouchi, T. and Nakajima, S. : A GUI Library based on Object Composition (in Japanese), Computer Software, vol.12, no.3, pp.49–58 (1995).

[115] Tonouchi, T., Fukushima, H., Manki, A., and Nakajima, S. : An Implementation of OSI Management Q3 Agent Platform for Subscriber Networks, Proc. ICC'97 (1997).

[116] Vogel, A. and Duddy, K. : *Java Programming with CORBA*, Wiley 1997.

[117] Wang, D.C., Appel, A.W., Korn, J.L., and Serra, C.S. : The Zephyr Abstract Syntax Definition Language, Proc. DSL'97 (1997).

[118] Wegner, P. : Concepts and Paradigms of Object-Oriented Programming, ACM SIGPLAN OOPS Messenger, vol.1 no.1, pp.7-87 (1990).

[119] Wing, J. : A Specifier's Introduction to Formal Methods, IEEE Computer, pp.8-24 (1990).

[120] Wing, J. and Zaremski, A.M.: Unintrusive Ways to Integrate Formal Specifications in Practice, CMU-CS-91-113, CMU (1991).

[121] Wirfs-Brock, R., Wilkerson, B., and Wiener, L.: *Designing Object-Oriented Software*, Prentice-Hall 1990.

[122] Wirfs-Brock, R. and Johnson, R. : Surveying Current Research in Object-Oriented Design, CACM 33 (9), pp. 104 - 124 (Sept. 1990).

[123] Wirsing, M. : Algebraic Specification, in *Handbook of Theoretical Computer Science: volume B, (J. van Leeuwen ed.)*, MIT Press/Elsevier 1990.

[124] Wirsing, M. : A Formal Approach to Object-Oriented Design, Seminar Talk at NEC (1995).

[125] Wirsing, M. and Knapp, A. : A Formal Approach to Object-Oriented Software Engineering, Proc. 1st Workshop on Rewriting Logic and its Applications (1996).

[126] Yamanaka, A., Nakajima, S., Tomono, M., and Tonouchi, T. : A HORB-based Network Management System, Proc. ICODP/ICDP'97 (1997).

[127] Yamasaki, T. : Surveys of Program Design Methods Using a Common Example Problem (in Japanese), Journal of IPS Japan, vol. 25, no. 9, p.934 (1984).

[128] Yatsu, H. and Futatsugi, K. : Verification of Z Specifications using Algebraic Specifications (in Japanese), Computer Software vol.13, no.6, pp.26-42 (1996).

[129] Yonezawa, A. : On Object-Oriented Programming (in Japanese), Computer Software vol.1, no.1, pp.29-41 (1984).

[130] Zave, P. and Jackson, M. : Conjunction as Composition, ACM Trans. Soft. Engin. Meth. vol.2, no.4, pp.379-411 (1993).

[131] ITU-T Rec. X.950-1 : Information Technology - Open Distributed Processing - Trading Function - Part 1: Specification (1997).

[132] BYTE magazine August 1981.

[133] OMG : OMG CORBA (http://www.omg.org).

[134] OMG : CORBAservices, Trading Object Service Specification (1997).

[135] OMG : UML (http://www.omg.org/uml/).

[136] pUML : (http://www.cs.york.ac.uk/puml/).

[137] Sun Microsystems : JavaCC Documentation (http://www.suntest.com/JavaCC/).