Software Evolution

Architecture-Based Software Evolution: A Multi-Dimensional Approach

By

Huan Wang, B.ENG., M.ENG.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2007) McMaster University

(Computing and Software) Hamilton, Ontario

TITLE: Architecture-Based Software Evolution: A Multi-Dimensional Approach

AUTHOR: Huan Wang, B.ENG., M.ENG. (China Agricultural University)

SUPERVISOR: Professor Thomas S. E. Maibaum

NUMBER OF PAGES: xvi, 136

ABSTRACT

Software Evolution is unavoidable because software systems are subject to continuous change, continuing growth and increasing complexity. As software systems become mission-critical and large in size, the complexity in software development is now focused on software evolution rather than construction. In this work, we view a software system as an entity that is evolving throughout its lifetime, during development and maintenance. Based on a broad survey of software evolution approaches, we propose an architecture-based solution for software evolution, which is defined in terms of evolution specific operations on architectural elements, that is, adding, removing, replacing components and (or) connectors, transforming configurations according to the required changes. In our view of software architectures, connectors are more likely to change since they are the architectural elements which reflect business rules. This work is focused on the evolution of connectors in architectures describing detailed design. Coordination contracts are introduced by Fiadeiro et al. as a realization of connectors at this detailed architecture level, which enables a three-layer architecture to separate concerns of components, connectors and configuration during evolution. Furthermore, to constrain the evolution in a predictable direction, we have established a matching scheme for justifying behavioral relationships between coordination contracts by specification matching based on pre- and postconditions of contracts and methods. A number of specification matches, with various degrees of similarity between the evolved and evolving contracts, have been developed for system behaviors after evolution operations. Case studies are exhibited give a better understanding of these matches.

ACKNOWLEDGEMENTS

It has been a great pleasure working with the faculty, staff, and students at McMaster University, during my life journey.

I start showing my sincere appreciation to my master's advisor, Dr. Tom Maibaum, for his invaluable guidance, advice and financial support, for his confidence in me to carry out this work, and for his considerable assistance in writing of this thesis. I am thankful for his insights on the "big picture" of this research area, as well as for many technical details in this specific topic. He is always willing to help me through when I have any difficulty in work and life.

I wish to thank my fellow students at the Software Engineering Research Group, especially Zhe (Jessie) Li, Rongshu (Bill) Yi, Xiang Ling and Yazhi Wang, for their kind support, and also Salvador Garcia, Pablo Castro and Jorge Santos, for the enjoyable time we have spent together.

Thanks to Jessica Stewart, for her every smile on her face and "paper" .

I am deeply indebted to Dr. Wolfram Kahl and the reviewers of my thesis, Dr. Emil Sekerinski and Dr. Spencer Smith, whose comments and suggestions have contributed to the quality of this work.

I would also like to thank Jonathan (Yumeng) Li for sharing me with faith in God.

I appreciate a number of people not listed explicitly above, whom I am so fortunate to receive help from.

To God Be the Glory! This work is an offering to Him.

Dedication

To my parents and my entire family,

for their unconditional support,

everlasting patience, endless encouragement,

and unfailing love.

Contents

List of Tables

List of Figures

Preface and Thesis Outline

This thesis is the final product of my Master of Science studies in Computer Science at the Department of Computing and Software, McMaster University, Canada. It serves as documentation of my research work during these studies, which have been carried out from Fall 2005 until Summer 2007. The work has been funded by my supervisor Dr. Tom Maibaum, NSERC, the Department, and the University.

The objective of this thesis is to propose an approach to software evolution based on software architectures, in particular for the detailed design level of development. We define architecture-based software evolution as adding, removing, modifying components and (or) connectors, and transforming configurations of components and connectors. However, most techniques for modeling Software Evolution do not model connectors, or they model connectors, but have no mechanisms to evolve them effectively. On the other hand, the approaches to modeling connectors in Architecture Description Languages rarely support the evolution of connectors, and some of them do not even model connectors as first-class entities. As a consequence, we need a way to model connectors as first-class entities and evolve connectors effectively. Coordination contracts are a realization of connectors at the detailed design level. With a three-layer approach, where the coordination layer models business rules by means of contracts, we are able to separate the concerns of components and connectors and focus on the evolution of coordination contracts. Generally speaking, we would like to see incremental and predictable evolution. Therefore, in principle, adding and modifying coordination contracts are predictable evolutionary operations. To evolve software systems predicably, firstly, we consider two evolving systems to be related by evolutionary operations by characterizing their change histories. When adding and modifying contracts, the signatures of two contracts should match, which means the types of each rule's input and output parameters, as well as the exceptions that may be raised, must match. Then, we compare their behaviors in terms of specification matching based on pre- and postconditions. In order to do this, we have to develop a pre- and postcondition characterization of contracts. We then propose a matching scheme with 5 cases, which relate the pre- and postcon-

ditions of two contracts with different conditions. The evolution of contracts is defined to be predictable up to the limits imposed by these specification matches. The research in this thesis has been influenced by work in several areas: software architecture, software evolution, architectural connectors, formal specifications, UML, coordination contracts.

The thesis consists of seven main chapters.

The first chapter contains a general introduction to the research background and an evaluation of the current state of the art. The discussion starts with fundamental knowledge of software maintenance, software evolution and software architecture, as a basis for our research motivation for evolution based on software architecture. We define architecture-based software evolution from a reconfiguration perspective, as adding, removing, replacing components and/or connectors, according to the required changes.

In Chapter 2, we survey the literature on techniques for modeling software evolution. Three relevant techniques are selected for the purpose of comparison and evaluated from two aspects, the representation of changes in architectures and the mechanism to evolve connectors. We find that two common problems with these techniques are that they either lack the architectural granularity of connectors or have not established an effective mechanism for evolution of connectors.

Chapter 3 investigates techniques for modeling architectural connectors and gains us an understanding of connectors. We demonstrate the significance of connectors being represented as first-class architectural citizens. We also explore several taxonomies of connector types. Notations and approaches for modeling connectors are discussed. A new light-weight way is suggested to represent connectors and support evolving architectural connectors using coordination contracts, which define a modeling and implementation primitive that allows transparent interception of method calls.

Chapter 4 provides a necessary background in coordination contracts. We compare coordination contracts with some popular techniques that may support modeling of architecture based software evolution. A three-layer architecture applied on coordination contracts is proposed to facilitate separation of concerns. We introduce graphical and textual notations of contracts, as well as a tool for developing contracts. Several applications of coordination contracts are also presented.

Chapter 5 proposes an approach to evolution of the coordination dimension. We characterize change histories that in a way enables control of system evolution in a predictable direction. Moreover, we define pre- and postconditions of method calls, coordination rules and coordination contracts, and make use of specification matching to justify the behavioral relationships between

coordination contracts by means of pre- and postconditions. Additionally, we provide a framework to assess different cases of specification matches, and demonstrate proof sketches and properties of a variety of matches.

Chapter 6 demonstrates the ideas on some case studies of a banking example to instantiate the corresponding concepts and approaches in the previous Chapter. In particular, we explored Exact Pre/Post Match and Plug-in Match with this example.

Chapter 7 contains the conclusions with emphases on this thesis's original contributions and proposes some future research goals for extending this work.

Chapter 1

Introduction

1.1 Motivations for Software Evolution

1.1.1 Origins of the Research on Software Evolution

The research on Software Evolution originated in the 1970's by Lehman [87] when studying over twenty releases of the IBM OS/360 operating system. Based on these experiences, Lehman and Belady proposed a group of laws for software evolution [88]. Many of these laws are still relevant even today [90]. From these laws, we are aware of the fact that software systems are subject to *continuous change*, *continuing growth* and *increasing complexity.*

Almost in the same period as Lehman's work, Parnas emphasized the significant impact of the fundamental principles of Software Evolution as early as the 1970s and 1980s. The concept of "information hiding" [118] decomposes a design into modules, which yields modularization as a basic design decision. The underlying intention is to encapsulate design decisions within individual modules so that changing design decisions will affect only some of the modules, rather than the whole software. Parnas also proposed the ideas of "design for change" and "anticipation of changes" as crucial aspects in software engineering [119], which are motivations for research on Software Evolution. In 1994, Parnas dealt with the issue of software evolution in more details [120].

1.1.2 The Ability to Accommodate Changes

It is evident that an increasing percentage of information systems as e-Business and e-Commerce is in need, which are mission-critical and large in size. Building such software systems is a big challenge for software practitioners. The

complexity and the pressure are not only from techniques or collaborations among people, but also from time-to-market, economic resources, severe constraints, as well as demanding from the software producer discipline and efficiency. Software Evolution is unavoidable both before and after deployment [111], so that software systems need to be evolvable in order to build large, complex, multi-lingual, multi-platform, long-running systems economically.

Thus, an intractable problem has come up: how can we construct software systems gracefully adapted to changing requirements over time? [53] gives us a general guideline — "the ability to change is now more important than the ability to create e-commerce systems in the first place. Changes become a first-class design goal and require business and technology architecture whose components can be added, modified, replaced, and reconfigured".

Technically speaking, most of these systems are modeled by component-centric software development, where business rules are reflected as volatile relations among components. When components become more complicated, the complexity of components' interactions will grow exponentially. Therefore, the complexity in software development is now focused on evolution rather than construction. We review a software system as an entity under development as well as evolution.

1.2 Software Evolution versus Software Maintenance

The waterfall model categorizes the development cycle of a software project into several phases. Though the waterfall model assumes the development to be a "linear" process, which is not true generally, we follow the model to introduce the concept of *Software Maintenance* and *Software Evolution.* Software Maintenance in Figure 1.1 is the very last stage, as well as the longest phase in the software life cycle.

The IEEE Standard 1219 [4] defines *Software Maintenance* as "the modification of a software product *after delivery* to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." Sommerville concluded that Software Maintenance has rich functionalities, such as maintenance to repair software faults, maintenance to adapt software systems to a different operating environment, maintenance to add to or modify the system's functionality [141].

However, the fact that Software Maintenance is the most cost-intensive in the entire software development life cycle surprised many software practition-

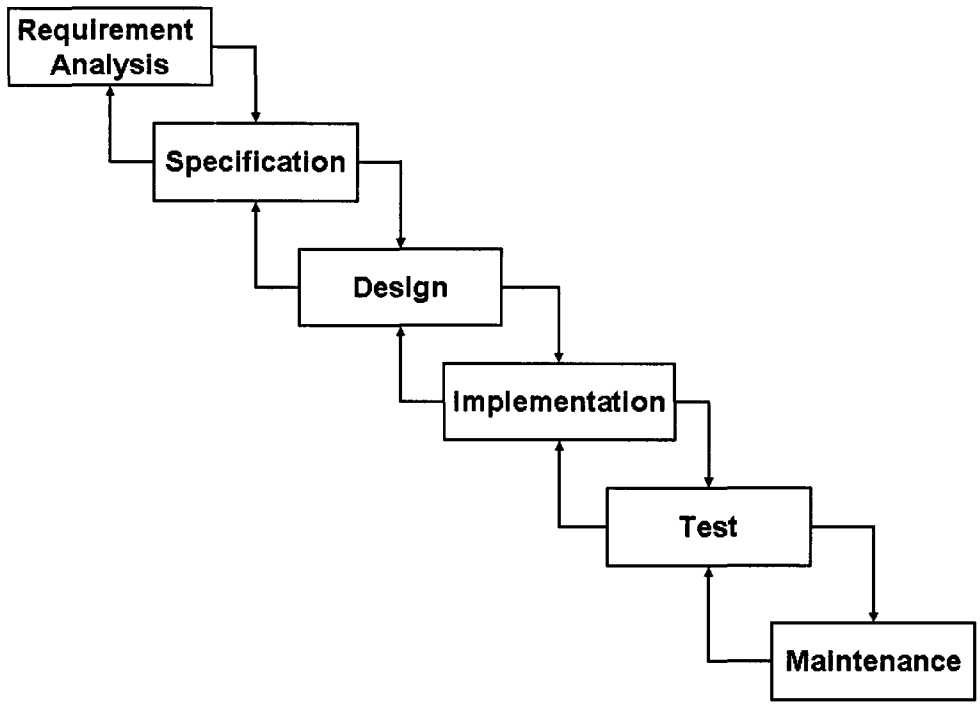


Figure 1.1: Software Development Lifecycle Waterfall Model

ers. According to Sommerville, approximately 60% of the cost of software system construction occurs during maintenance period [141]. A relatively new assessment in [81] even declares that the cost for maintaining software and managing its evolution now amounts to more than 90% of the total cost. These numbers have properly illustrated the magnitude of such a serious situation.

The concept of Software Evolution and Software Maintenance are highly related. In 2000, Lehman and Ramil [39] defined Software Evolution as "*all* programming activity that is intended to generate a new software version from an earlier operational version". However, a list of various definitions in [39] shows that there is not a consensus yet on how to define Software Evolution. Considering this, firstly we will induce a general definition of Software Evolution by referring to that of Software Maintenance.

From the analyses above, compared to Software Maintenance, Software Evolution should support a more general concept for being involved in all phases of development. Ghezzi [66] argues that evolution should be planned throughout software development activities. Experiences already testified to the fact that software systems with an evolvable architecture cost less to maintain and Software Evolution is a key to software productivity [111].

Toward reducing the cost of software evolution, making it more effective

and positioning our research objectives, we will explore Software Evolution from various perspectives.

Chikofsky et al. classifies three different activities in constructing software: *requirements*, *design* and *implementation* [37]. From these criteria, they built concept of Forward Engineering, Reverse Engineering and Re-engineering. *Forward Engineering* is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system. *Reverse Engineering* is the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction. A typical reverse engineering framework is presented by Sartipi [131, page 15]. *Re-engineering* is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

In consequence, research on Software Evolution in the early stages of reverse engineering and re-engineering was mainly operating on legacy systems. The process of evolution within this scope usually involves analyzing, understanding the program that has to be changed, implementing required changes [141], versioning resulting systems, architectural recovery, etc. Techniques have focused on reducing the complexity through automatic support for program comprehension, which includes visualization and reverse engineering techniques like artifact extraction, as well as focused on restructuring or refactoring[1], which includes problem detection and problem correction. Thus, practitioners have to reason from experiences or source code even in the absence of an explicit architecture. However, by patching the problematic areas, they may introduce more problems and by applying immense modifications, design decisions will drift [65].

It has been shown that the *after-delivery* Software Evolution is extremely expensive because it requires understanding, analyzing and evaluating the application or system thoroughly. If the original system is hard to comprehend or even itself has poor software quality, software evolution in such a context will encounter great impediments. To avoid the high cost, we want to emphasize the importance of Software Evolution at the initial phases in software development, curbing the deteriorating situation.

Furthermore, the timing of evolution unfolds two new notions for us: *static* evolution which happens at the design or specification time, and *dynamic* evolution which happens at the execution time (run-time or dynamic). Changes are provided at many levels of granularity, so that the corresponding evolution

[1]Refactoring is defined as structural transformations on source code that do not affect the external behavior of the code.

pervades all development activities: *coarse-grain* as in specification, framework, design, architecture and *fine-grain* as in data, schemas, source-code, modules, test cases, etc.

Moreover, research on Software Evolution is dedicated in two aspects according to Mens and Wermelinger [104]: the *what and why* studies evolution as a noun in the sense of observing phenomena, nature and underlying motivations of Software Evolution; and the *how* involves the methods, tools and practices for evolving a system, in particular for a constantly changing model, a specification of the system and the system implementation.

In summary, though recognizing the importance of Software Evolution, most of the support for evolution research has been focused on techniques or tools for dealing with structural complexity, finer-grain software artifacts and the "what and why" aspect of software evolution. For the purpose of our research, we are using forward engineering approach of construction to discuss evolution, performing static evolution as early as design time on the architectural level concerning coarser-grain artifacts and considering the "how" of software evolution.

1.3 Architecture-Based Software Evolution

1.3.1 Motivations

To address the recently-emerged problems in engineering large, complex software systems, three popular techniques come up: Component-Based Software Development (CBSD), middleware platforms and software architecture [102][2]. Component-based software development highlight components and their interrelationship, but ignore the importance of connectors, which we will mention later. Software architecture has come to the fore and become an indispensable area in software engineering. Researchers and practitioners have started to consider software development processes from an architecture point of view.

Since the beginning, software architecture based approaches have taken over the previous absolute authority of modules. Architectural design is the initial design process of identifying sub-systems and establishing a framework for sub-system control and communication [141]. As the output of the design process, software architectures describe the structure of a system or a program and its global properties. Once determined, any change in architectures will impact a substantial set of functional and non-functional properties. By raising

[2]Generally speaking, architecture-based development is top-down decomposition and component-based development is bottom-up composition.

the level of abstraction, software architecture is a way to control software development, evolution costs and challenges and to improve software quality.

Surprisingly, most definitions of software architecture do not explicitly mention evolution. Though in recognition of the fact that software evolution will extract huge costs after initial development, software processes and design techniques still concentrate on software construction, so that current support for architecture-based software evolution has been insufficient. It is important for evolution to be based on the architectural level to make the decision effective throughout the whole life cycle of a system. As a saying goes, prevention is better than cure.

Jazayeri even argued that the primary goal of a software architecture is to guide the evolution of the system [78]. Such ideas of integrating software architecture with evolution motivates our research.

To make our motivation more explicit, we position our notion of software evolution by dimensions under timing and granularity in Figure 1.2[3]. During *design time*, the architecture is still under development; *pre-execution time* is when the architecture has already been specified and implemented but not yet running; *runtime* means the architecture can be modified dynamically while running.

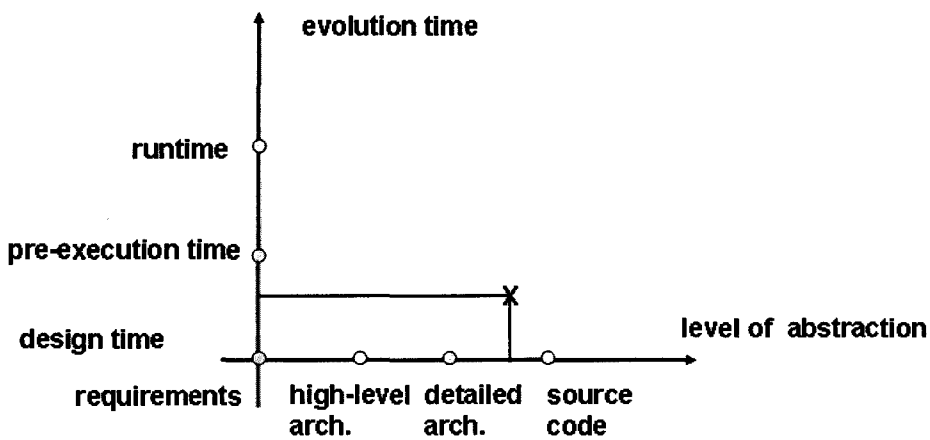


Figure 1.2: Software Evolution Dimensions under Timing and Granularity

1.3.2 Software Architecture as a Buzzword

The word "architecture" is not original to, nor specific to software engineering. Since the day it was introduced, controversy on how to define it has not

[3]In the "evolution time" direction, we adopt concepts by Mens et al. [103].

stopped. The website of Software Engineering Institute (SEI) [3], exhibits over 90 definitions to interpret various perspectives of an architecture.

We will adopt the definition by Bass et al. [27]: the software architecture of a *program* or computing *system* is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

In spite of the existing dissension on how to define software architecture, there is an almost-clear consensus on elements of software architecture, which are components, connectors and configurations [64]. *Components* represent the primary computational elements and data stores of a system. *Connectors* represent the interactions among components. *System* represents hierarchical organizations of components and connectors. An instance architecture is a topology of a particular set of components and connectors.

What benefits do we desire for software architecture? Software architectures not only function as a bridge between requirements and implementation [62], but also represent several roles, such as enhancing understanding of system, supporting reuse at different levels, directing construction, facilitating evolution, analysis and management [61].

1.3.3 Documenting Software Architecture

Generally speaking, architecture documentation may serve as a means of educating associated people, ease communication among different stakeholders, support the basis for system analysis [34]. Notations in architecture documentations could be either graphical or textual, or both. In any case, they all should be capable of describing architectures in multi-faceted and systematic ways.

The "4+1" View Model

As a straightforward graphical representation applied universally, box-and-line approaches [135] represent components as nodes and connectors as edges so that an architecture of a system is a directed graph. While enjoying simplicity, one has to compensate for the ambiguous meaning of every unit, perspectives from different stakeholders and analysis or reasoning about architectures.

The "4+1" view [84] was created to reflect concerns from involved stakeholders. As its name indicates, the "4+1" view is illustrated by five organized views: the logical view (end-user functionality) models objects of the design using an Object-Oriented method; the development view (programmers, software management) describes the *static* organization of the software; as a com-

plement to the development view, the process view (integrators, performance, scalability) captures the concurrency and synchronization aspects of the design; the physical view (system engineers, topology, communications) maps the software to the hardware and explicates distributed properties. The fifth view is the scenario view, representing dynamic aspects shared in the other views. To a large degree, UML (Unified Modeling Language) is influenced by this seminal idea.

ADL

Prior to Architecture Description Languages (ADL), Module Interconnection Languages (MIL) and Interface Description Languages (IDL) tried to serve a similar purpose for the source code. The two languages specify relationships between modules at the source code level but lack the capability of architectural level abstraction.

The way ADLs named suggests that ADL is used to describe the architecture of a software system. With the indispensable architecture elements discussed in Section 1.3.2 (page 10), ADL should adequately capture components, connectors, and the system configuration.

Over the past 10 years, a lot of ADLs come into play. Medvidovic et al. set up a classification and comparison framework for several ADLs [100]. In 2007, they advocated a creative idea that the second generation ADLs should consider "three lampposts" [96] — not only from insights of pure technology as the first generation also from two other aspects — domain, and business.

1.3.4 Software Evolution at the Architectural Level

Most earlier research works on software architectures are dealing with specifying, describing, analyzing, implementing, evaluating issues. On the other hand, though research works on software evolution have been proceeding for almost thirty years since Parnas and Lehman, only recent phenomena and the related underlying significance have started to be realized [89]. While the two companies working alone separately for a long time, current advances in both sides lead to develop architecture-based software evolution coincidentally.

Research on architecture-based software evolution is not only essential to evolve software architecture also to assess an architecture design, facilitate architecture-based development, enhance software quality and increase confidence on products by managing changes more successfully. Software architecture is evolvable, so are its elements — components, connectors, and system configurations.

In favor of our research, evolution of architectures is managed on architectural elements through a particular set of strategies or rules. Then we define software evolution by means of architecture re-configuration as evolving operations on architectural elements — that is, adding, removing, replacing component and/or connectors, according to the required changes.

1.4 Summary

In this Chapter, we have introduced fundamental knowledge of software maintenance, software evolution and software architecture as a basis for our research motivation for evolution based on software architecture. Software systems are subject to continuous change, continuing growth and increasing complexity so that evolution is unavoidable. We define architecture-based software evolution from a reconfiguration perspective, as adding, removing, replacing components and/or connectors, according to the required changes.

Chapter 2

Techniques for Modeling Software Evolution

Many approaches to modeling software evolution are predominantly based on a relatively low level of abstraction, being programming language specific, or operating in an ad-hoc way. After extensively surveying the literature, we have found that current research on evolution at the architectural level essentially applying some logical formalism, type theory and graph transformation techniques. In this chapter, we will present some related work and compare them using two main criteria: the representation of changes in architectures and the mechanism to evolve connectors.

2.1 Logical Framework

Lucena and Alencar [10, 94] proposed a theorem-proving based logical framework for software architecture analysis and evolution by structural and functional descriptions that allow validation of architectural changes and to assess the generated impacts. The support formalism is many-sorted deontic modal action logic which describes transitions that may occur in software evolution (defined as a sequence of change processes). They describe the evolution by means of architectural configuration where transitions are represented as actions with deontic constraints given by the statement of "permission" and "obligation" allowing deduction about the validity of transitions.

They proposed a formal software architectural description which consists of features like versions, modules and subsystem families. Furthermore, they defined a tuple $SS = \langle SG, SR, SV, SI, SC \rangle$ to represent an architecture. SG is an acyclic structure graph which describes the hierarchical relations between

module families and subsystem families. SR contains the resource-related information. Likewise, SV holds version-related information, SI the interface-related information, SC the configuration-related information. Each element in tuple SS is also defined in terms of tuples as shown in Figure 2.1. Figure 2.2[1] is an expanded overview of SS.

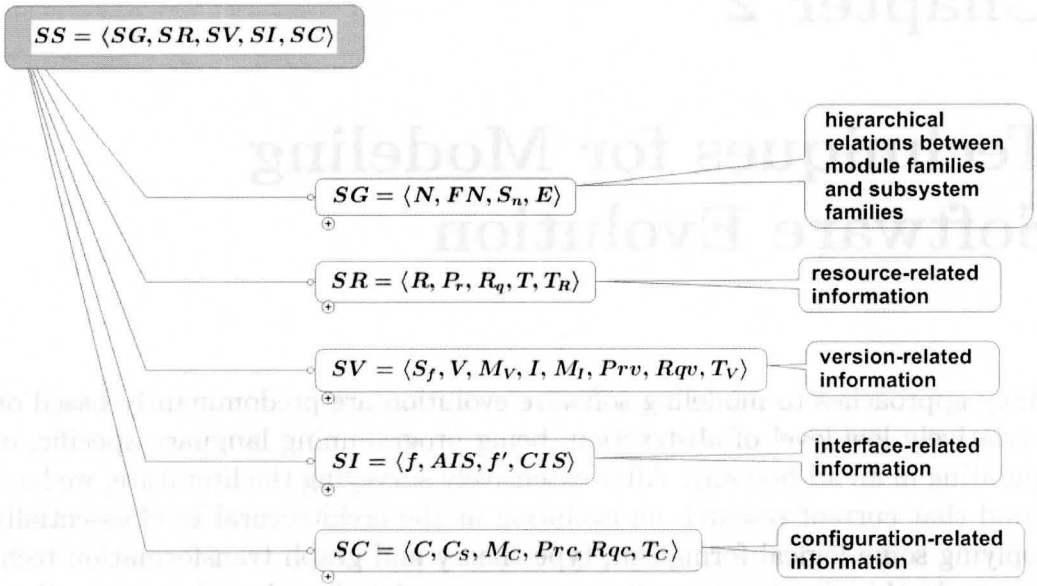


Figure 2.1: Software Architecture Denoted by Tuples

The logical framework for modeling software evolution is based on architecture configuration. Each particular architectural description is instantiated by SS_i ($i \in \mathbb{N}$), which consists of a sequence of system configuration states. The initial description $SS_0 = \langle SG_0, SR_0, SV_0, SI_0, SC_0 \rangle$. From the start, an architecture evolves to SS_i through executing a chain of transition rules r_j ($j \in \mathbb{N}$) until arriving at an expected architecture, i.e., $SS_0 \xrightarrow{r_0} SS_1 \rightarrow ... \rightarrow SS_{i-1} \xrightarrow{r_j} SS_i$. Like the possible changes on each element in the tuples shown in Figure 2.2, the rules can be accordingly imposed on structures, resources, versions, configurations, functionalities, or static and dynamic properties of the architectural configuration transformation.

To conclude, Lucena and Alencar's logical framework identified a conceptual model evolving through architecture configurations based on transitions, and worked "as a programming-in-the-large transformation process applied to architectural descriptions of software systems" [10]. Theorem provers are used as tools to reason about software evolution in this logical framework. From the

[1]Both Figure 2.1 and Figure 2.2 are adapted from [10].

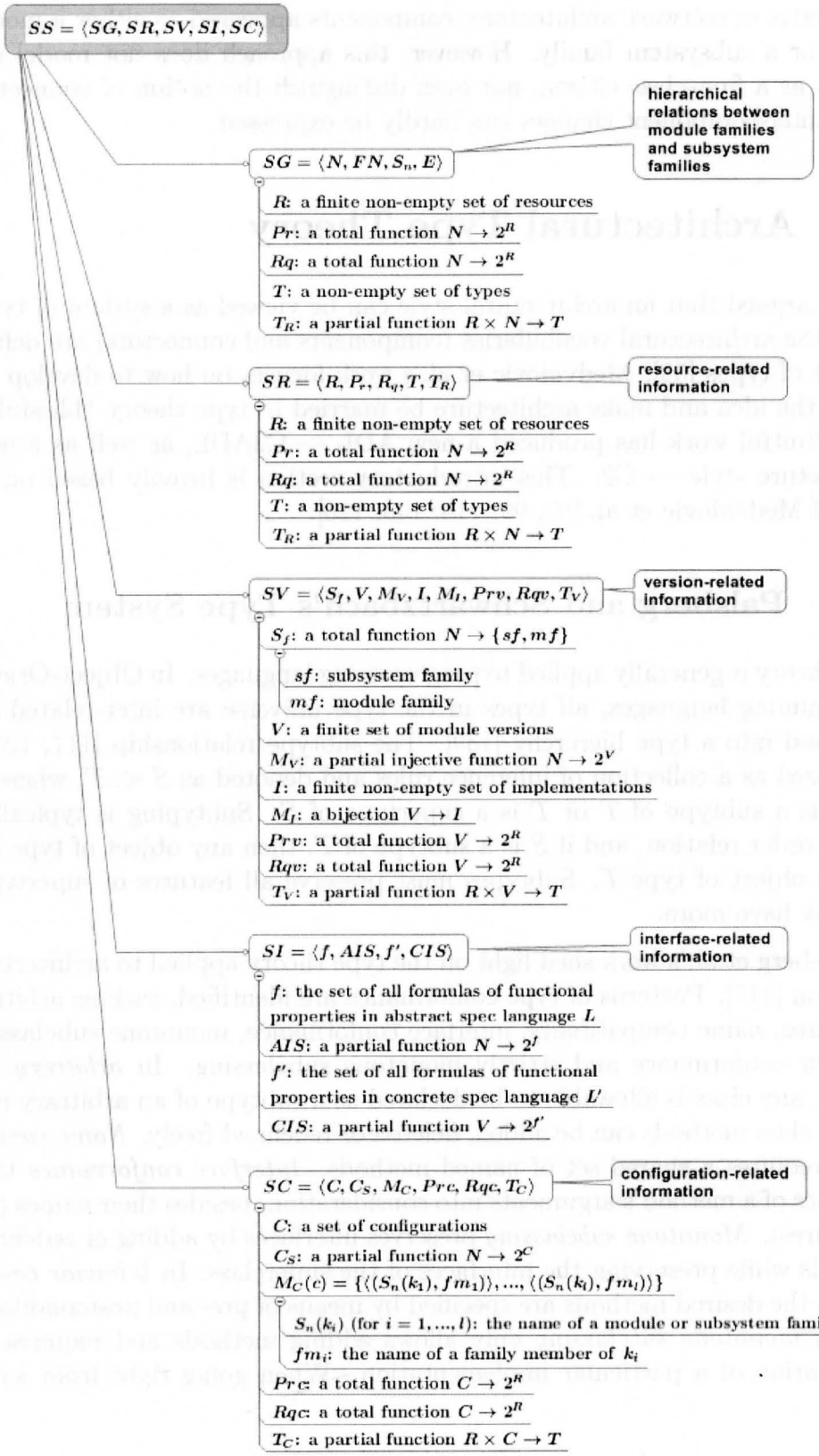


Figure 2.2: Expanded Software Architecture Description

perspective of software architecture, components appeared as either a module family or a subsystem family. However, this approach does not model connectors as a first-class citizen, nor even distinguish the notion of connectors. Thus, inter-component changes can hardly be expressed.

2.2 Architectural Type Theory

Garlan argued that an architectural style can be viewed as a system of types, where the architectural vocabularies (components and connectors) are defined as a set of types [59]. Medvidovic et al.'s work focuses on how to develop and realize the idea and make architecture be married to type theory "blissfully". Their fruitful work has produced a new ADL — C2ADL, as well as a novel architecture style — C2. This introductory section is heavily based on the work of Medvidovic et al. [95, 99, 110, 143, 116].

2.2.1 Palsberg and Schwartzbach's Type System

Type theory is generally applied to programming languages. In Object-Oriented programming languages, all types in the type universe are inter-related and composed into a type hierarchy [139]. The subtype relationship [117, 126] is formalized as a collection of inference rules and denoted as $S <: T$, where we call S is a subtype of T or T is a supertype of S. Subtyping is typically a partial-order relation, and if S is a subtype of T, then any object of type S is also an object of type T. Subtypes must preserve all features of supertypes, but may have more.

Palsberg et al.'s work shed light on the type theory applied to architectural evolution [117]. Patterns of type conformance are identified, such as arbitrary subclasses, name compatibility, interface conformance, monotone subclassing, behavior conformance and strictly monotone subclassing. In *arbitrary subclasses*, any class is allowable to be declared as a subtype of an arbitrary class so that class methods can be added, deleted or redefined freely. *Name compatibility* requires a shared set of named methods. *Interface conformance* takes the types of a method's arguments into consideration, besides their names (i.e., signatures). *Monotone subclassing* preserves interfaces by adding or redefining methods while preserving the interfaces of the superclass. In *behavior conformance*, the desired methods are specified by means of pre- and postconditions. *Strictly monotone subclassing* only allows adding methods and requires the preservation of a particular implementation. When going right from left in

Figure 2.3[2], the typing mechanism becomes more expressive and rigorous by considering more object features. Table 2.1 builds a correspondence between patterns of type conformance and the subtyping relations.

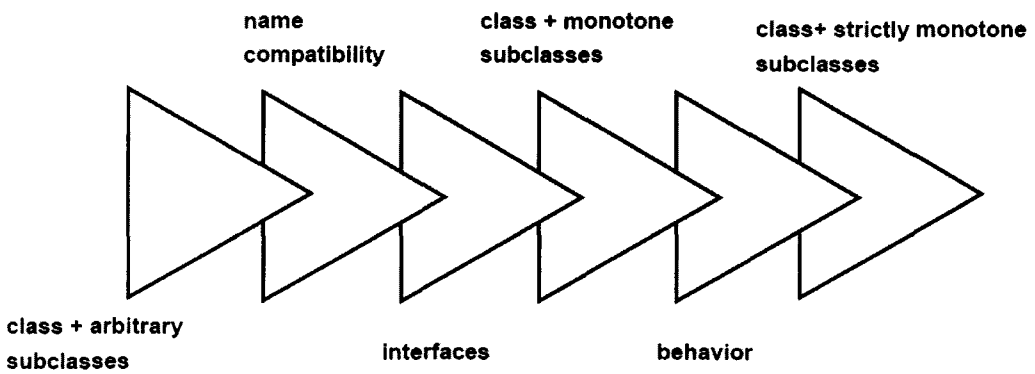


Figure 2.3: Subclassing Mechanism

Table 2.1: Subtype Relationships Established on Figure 2.3

Patterns of Type Conformance	**Subtyping Representation**
class + subclasses	subclassing
name compatibility	more methods
interface	conformance
behavior	weaker preconditions, stronger postconditions
strictly monotone subclassing	weaker preconditions, stronger postconditions, other specs to preserve implementation

2.2.2 Evolving Architectural Components

As shown by Medvidovic et al. [99], the notion of subtyping adopted by ADLs is richer than that typically provided by programming languages; that is, it involves constraints on both syntactic (e.g., naming and interface) and semantic (e.g., behavior) aspects of a component or a connector. The subtype relations that are currently captured in [99] allow a subtype to preserve its supertype's interface, behavior, or both.

[2]Figure 2.3 is adapted from [117].

C2 Architecture Style

Originally intended to model *Graphical User Interface* (GUI) intensive applications, the C2 architecture style is generally applied to build architectures of large-scale, highly-distributed, heterogeneous, evolvable, and dynamic systems, and is independent of implementation language.

In a C2-style architecture, each component or connector has two defined interfaces — "top" and "bottom" (see also the filled black circles in Figure 2.5), maintaining messages sent and received, respectively. One such component interface may be attached to at most one connector. However, a connector may be attached to multiple components or connectors, transferring messages between them. A C2 architecture, where components are linked together by connectors into a hierarchical network, follows a principle of *limited visibility* (a.k.a., *implicit invocation* or *substrate independence*), i.e., a component is only aware of services provided from "above" but has no knowledge of services provided "beneath". Thus C2 messages (event-based) are of two kinds — requests (sent up) and notifications (sent down). Passing these two kinds of messages via connectors is the only way for components to communicate (direct communication between components is disallowed). C2ADL is an ADL for defining architectures built according to the C2 style.

C2 Component

Indicated in the above sections, a C2 component is a unit of computation, maintaining states or a data store, performing operations and exchanging messages (synchronous and asynchronous) with other components via two interfaces, "top" and "bottom". A well-formed definition of C2 component is shown in Figure 2.4[3]. C2ADL treats each component specification in an architecture as a type and supports its evolution via subtyping. C2 specifies component's semantics in FOL (First Order Logic).

Formally, each component is identified as a type and represented as a tuple, $Component =< nam, int^*, beh, imp >$, which consists of a name, a set of interfaces, a behavior and an implementation as in Figure 2.4.

- Each interface element has a direction indicator and a set of parameters, $int =< dir, int\_nam, param^* >$, where
  - *dir*: direction indicator (*pro*vided, *req*uired);
  - *int_nam*: interface name;

[3]Figure 2.4 is adapted from [99].

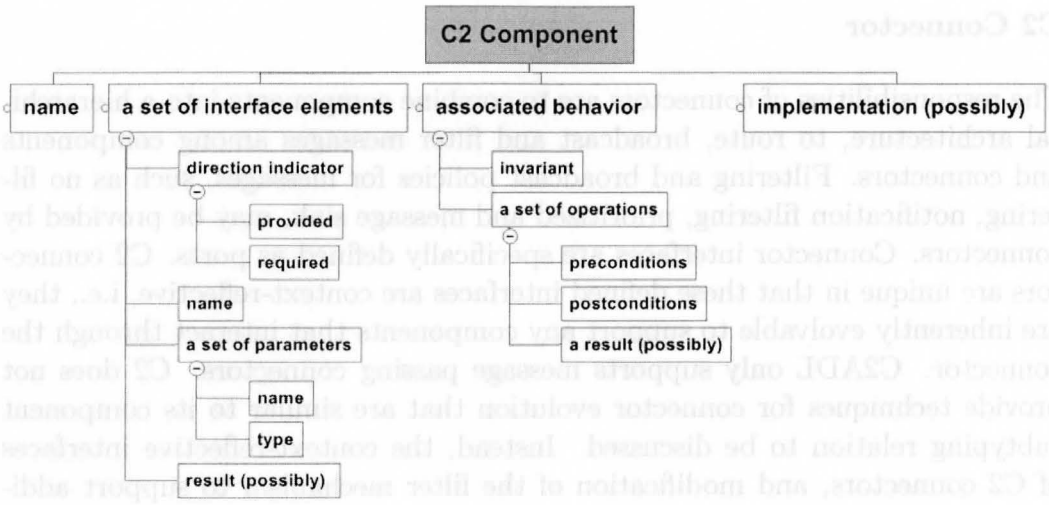


Figure 2.4: C2 Component Elements

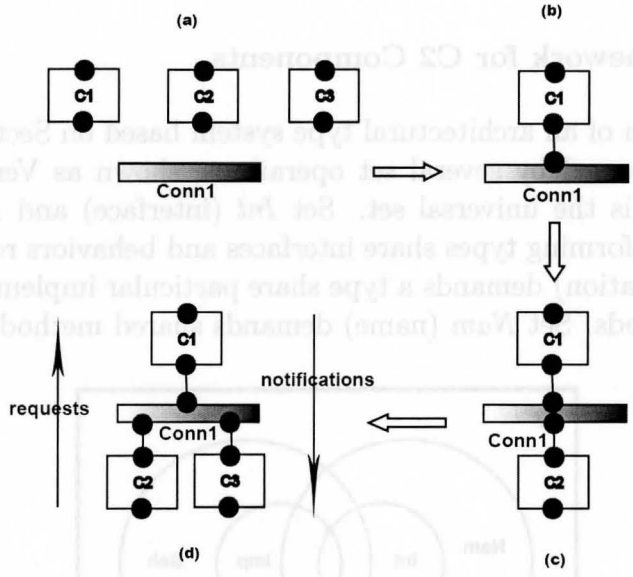


Figure 2.5: A Demo for C2 Component-Connector Architecture

  - $param^*$: a set of parameters, each parameter has a name and a type: $param =< param\_nam, param\_type >$.

- The behavior has an invariant and a set of operations, each operation has pre- and postconditions and a result (if there is any):
  - $beh =< inv, oper^* >$
  - $oper =< pre, post, result >$

C2 Connector

The responsibilities of connectors are to combine components into a hierarchical architecture, to route, broadcast and filter messages among components and connectors. Filtering and broadcast policies for messages, such as no filtering, notification filtering, prioritized and message sink, may be provided by connectors. Connector interfaces are specifically defined as ports. C2 connectors are unique in that these defined interfaces are context-reflective, i.e., they are inherently evolvable to support any components that interact through the connector. C2ADL only supports message passing connectors. C2 does not provide techniques for connector evolution that are similar to its component subtyping relation to be discussed. Instead, the context-reflective interfaces of C2 connectors, and modification of the filter mechanism to support addition or removal of components are two techniques to realize evolution of C2 connectors.

Evolving Framework for C2 Components

The construction of an architectural type system based on Section 2.2.1 (page 18) can be expressed by several set operations, shown as Venn diagrams in Figure 2.6[4]. *U* is the universal set. Set *Int* (interface) and *Beh* (behavior) demand two conforming types share interfaces and behaviors respectively. Set *Imp* (implementation) demands a type share particular implementations of all supertype methods. Set *Nam* (name) demands shared method names.

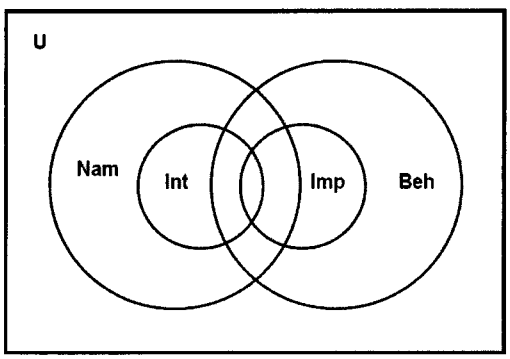
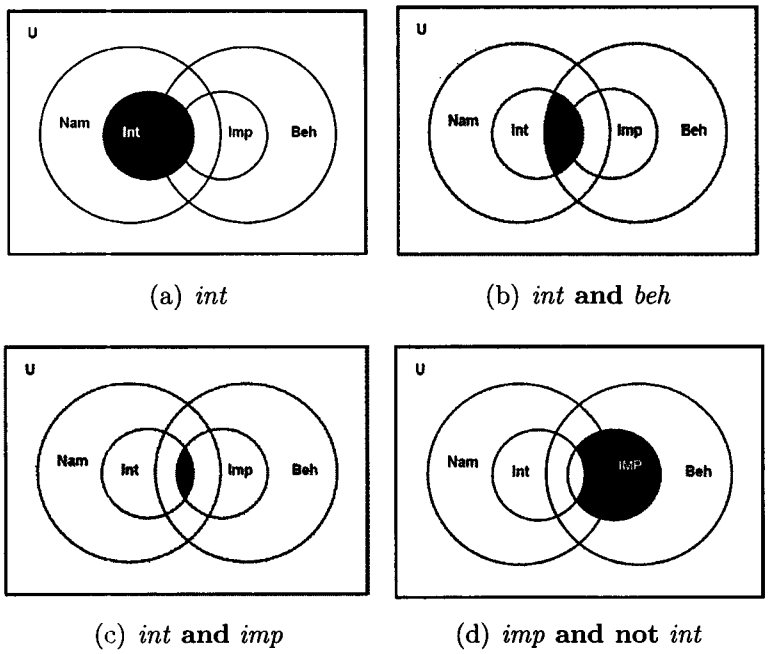


Figure 2.6: A Framework for Understanding OO Subtyping Relationships

- *interface conformance* (`int` as in Figure 2.7(a)) applies interface subtyping to provide a new implementation for a component of the original

[4]Figure 2.6 and Figure 2.7 are adapted from [99].



(a) *int* (b) *int* **and** *beh*

(c) *int* **and** *imp* (d) *imp* **and not** *int*

Figure 2.7: Examples of Component Subtyping Relationships

architecture, which is useful for interchanging components without affecting dependent components.

- *behavioral conformance* (the intersection of **int and beh** as in Figure 2.7(b)) requires that both interface and behavior of a type be preserved in demonstrating correctness during component substitution.

- *strictly monotone subclassing* (**int and imp** as in Figure 2.7(c)) extends the behavior of an existing component while preserving correctness relative to the rest of the architecture, so as to evolve a component with additional functionalities.

- *implementation conformance with different interfaces* (**imp and not int** as in Figure 2.7(d)) is useful specifically in describing domain translators[5] in C2, which allow a component to be fitted into an alternate domain of discourse.

- *multiple conformance mechanisms* allow to create a new type by subtyping from several types, potentially using different subtyping mechanisms.

In most Object-Oriented Programming Languages (OOPL), the subtyping mechanisms above would be realized by different programming languages

[5]Domain translators provide functionality similar to that of the adapter design pattern.

since every single OOPL supports at most one such mechanism. It is worth mentioning again that architectural types are not of the same meaning as the general notion of types in programming languages (integers, strings, arrays, records, etc.). When scaled up to the architectural level, they all may need to be supported by more than one language.

Medvidovic proposed a framework for evolving software architecture by using type theory with regard to architectural elements like components and configurations [95].

- a *component* evolves by means of a heterogeneous subtyping theory for software architectures;
- a *connector* evolves by context-reflective interfaces to support any components that interact through the connector and by heterogeneous information filtering mechanisms;
- a *configuration* evolves by employing heterogeneous, flexible connectors and minimal component interdependencies using implicit invocation, asynchronous communication or substrate independence.

In such a context, evolution of components can be represented by a subtyping relationship between two components, i.e., $C_j \leq C_i$, as the disjunction of sets *nam*, *int*, *beh* and *imp*, which are shown in Figure 2.6 (page 22). Thus, the subtyping relation can be defined intuitively as:

$$(\forall C_i, C_j : \text{Component})(\ C_j \leq C_i \Leftrightarrow$$
$$C_j \leq_{\text{name}} C_i \vee C_j \leq_{\text{int}} C_i \vee C_j \leq_{\text{beh}} C_i \vee C_j \leq_{\text{imp}} C_i),$$

i.e., by the disjunction of name subtyping, interface subtyping, behavior subtyping, implementation subtyping.

ArchStudio 3 [116] is an architecture-driven software development environment that supports the C2 architectural style. Mae [128] is an external change management tool assistant for ArchStudio in providing revisions of components and connectors.

To sum up, this approach is notable in its support for components and configurations evolution. The subtyping mechanisms for evolving components are claimed to be independent from domain, style, and ADLs [98], where each component specification is treated as a type and evolved via subtyping rules. However, the evolution of connectors is largely dependent on components by means of context-reflective interfaces and information filtering mechanism, which makes addition, removal, replacement, and reconnection of connectors difficult and not flexible.

2.3 Transformation Techniques

According to Heckel et al. [72], a general concept of transformation refers to "the manual, interactive, or automatic manipulation of artifacts according to pre-defined rules, either as a conceptual abstraction of human software engineering activities, or as the implementation of mappings on and between modeling and programming languages". Favre et al. [47] summarized a few transformation formalisms applied to software evolution: program transformation (over Java, C, or C++, etc.); model transformation (over UML and other visual languages); graph transformation; term rewriting, category theory, algebra, and logic. Transformation techniques such as model transformation, graph transformation and category theory are appropriate for software evolution at the architectural abstraction level. Especially, it is very straightforward to describe software architecture as a directed graph, where components are represented by nodes and connectors are the edges connecting nodes. In what follows, we will introduce two approaches in this category.

2.3.1 UML-based Algebraic Graph Rewriting

We have discussed in Section 1.3.2 (page 10) that UML reflects different views of software architectures. Recognized as a de facto standard in industry, UML has been used for modeling, analyzing and designing Object-Oriented software development. To describe architectural level evolution, Ciraci et al. [33] models designs by using UML (especially class and interaction diagrams), and is based on category theory, where software architecture is a typed graph and evolution processes can be viewed as morphisms between components (e.g., classes); see Figure 2.8[6]. Actually, they present a model as a class diagram and focuses on addition and removal as representative evolving operations for components. In Figure 2.8, the four nodes in the abstract software evolution model are:

- *Component*: the components which are going to evolve
- *New Component*: the components after evolving
- *System*: the system with original components
- *New System*: the system with evolved components

The evolution requests can be viewed as morphisms on the architectural components. The evolution model in Figure 2.8 can be depicted as a pushout, where the input consists of morphisms *Embedding: Component* $\rightarrow$ *SYSTEM*

[6]The following example is taken from [33].

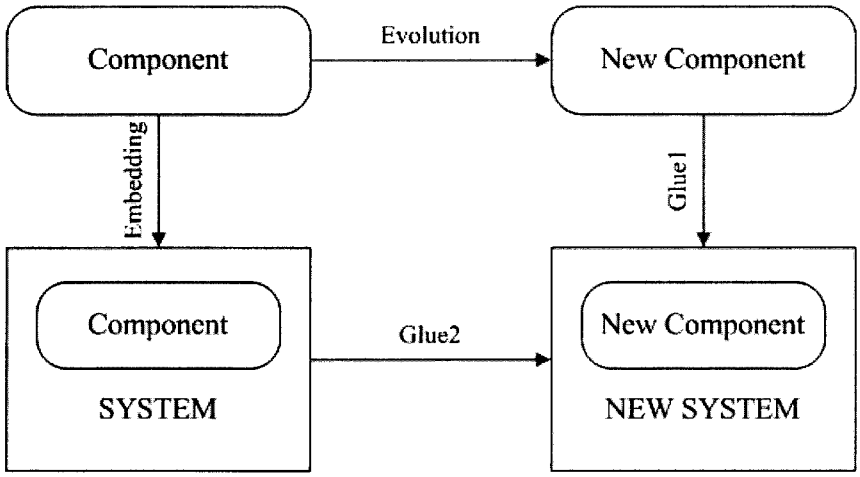


Figure 2.8: An Evolution Model

and *Evolution: Component* → *New Component*, and the output consists of *Glue2: SYSTEM* → *NEW SYSTEM* and *Glue1: New Component* → *NEW SYSTEM*, so that the diagram commutes. *Evolution* defines which rewriting should be done. *Embedding* identifies the occurrence of the part of *Component* that should be rewritten.

Ciraci et al. [33] allow three levels of adding and removing components: parameter and return value level, method and attribute level, class level. For simplicity and relevance, we will concentrate on class level operations.

Suppose that, as in Figure 2.9, the *Component* node is composed of two classes, F and D. Via *Embedding* morphism the *System* node contains relationship *aggregation* between classes F and D. The *New Component* node has one more class E with the *generalization* relation to class F and *aggregation* relation to class D. The *Glue* morphism contains the new class and its relationship with existing classes, as well as the relationships between existing classes. Thus, the *New System* node contains the new relations while preserving the relation between classes F and D in the *System* node.

In Figure 2.10, to demonstrate removal operations on classes, we will remove class E from the *System*, which will cause the edges connecting class E to its superclass F and to class D to be removed as well. The marked items ($\star$) stay in the graph temporarily but have a "discarded" status. Once removing the marked items will not cause any dangling arcs, the items can be removed from the system completely. So in this case, we are able to remove E safely.

Graph transformation theory provides the power to reason between versions. GXL (Graph Transformation Language) [130] is a programmable graph rewriting language that aims to address the limitations of both graph rewriting

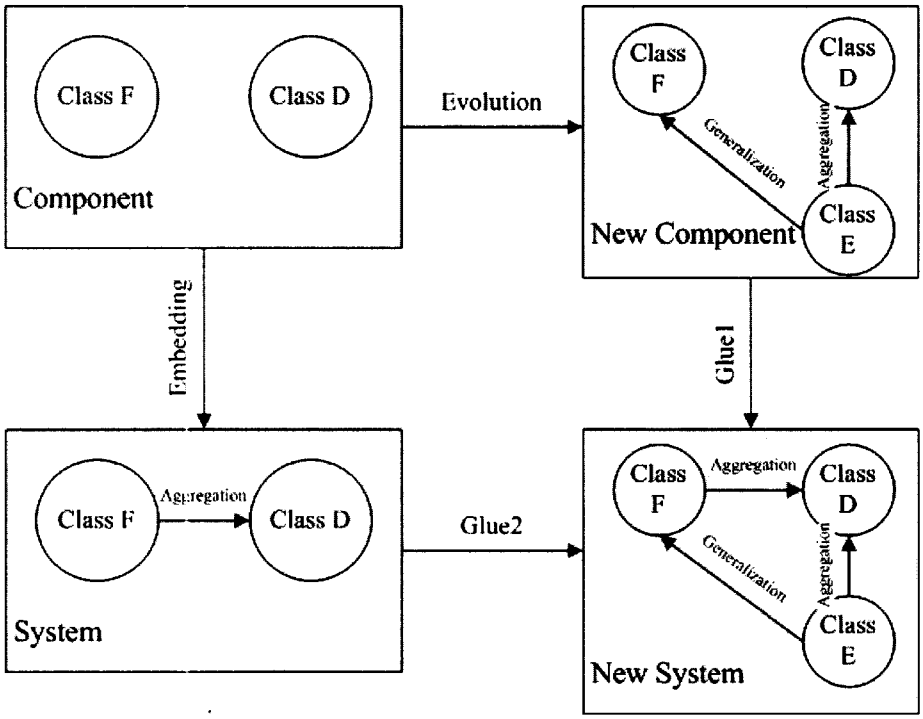


Figure 2.9: Adding a Class *E*

and tree rewriting by a synthesis of the two that uses the strengths of each one to address the weaknesses of the other. To represent graph rewriting as used by Ciraci et al. [33], GXL is an advisable choice. Tools which can "speak" the GXL dialect will support software evolution by means of algebraic graph rewriting techniques.

This method employs UML as a modeling language, where adding and removing classes have proven to be feasible. However, component's addition and removal are only a partial version of software evolution described in Section 1.3 (page 9). Connectors and related evolving operations are absent from the model.

2.3.2 Fiadeiro & Wermelinger's Approach

By applying algebraic graph rewriting, as described in Section 2.3.1 (page 25), Fiadeiro and Wermelinger [52, 146, 147] have proposed an approach to modeling software evolution at the architecture level. It is motivated by the research status quo that arbitrary reconfigurations are not possible, the languages used for representing computations are very simple and at a low level of abstraction, and that the combination of reconfiguration and computations leads to

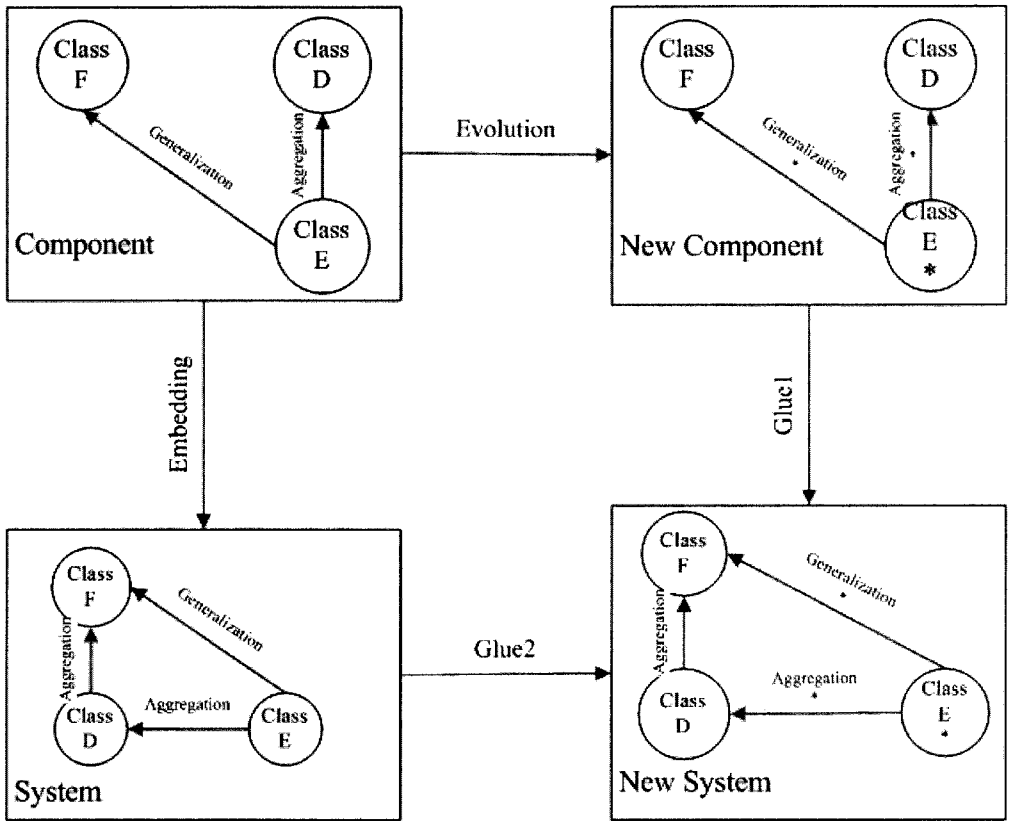


Figure 2.10: Removing a Class E

additional formal constructs. Nowadays, the application of category theory to software engineering is a very active research area. This method uses a uniform algebraic framework based on category theory and a program design language with explicit states, representing a software architecture via an ADL — CommUnity.

CommUnity [114, 148] is a UNITY-like parallel programming design language to describe computations, which represents architectures by diagrams in Category Theory, thus specifying reconfigurations by graph transformation rules. An instance architecture in CommUnity is composed of nodes and interactions between the nodes, where a component is a CommUnity design, a node is a component instance, interactions could be connections between input and output channels of different nodes or synchronization of actions of different nodes. A design is a unit of computation which has input/output/private attributes (called "channels" in CommUnity) and shared/private actions.

The CommUnity workbench [36] is a graphical integrated development environment for CommUnity programs which also supports the configuration and reconfiguration of architectures of complex systems as well as the

management of a library of components and connectors.

Software evolution in COMMUNITY is reconfiguration-based through conditional graph rewriting rules on the state of involved components. These rules are defined by using the double-pushout approach to graph transformation and category theory. Thus, primitive operations defined in Section 1.3.4 (page 12) for evolving connectors at architectural level are feasible. Comparing changes in different versions of architectures is via underlying formalism such as graph transformation theory and category theory. The coordination contracts we will discuss in Chapter 4 have the same semantics as COMMUNITY.

2.4 Other Related Approaches

There are innumerable other related techniques and methods that may or may not have direct impact on modeling software evolution at an architectural level. For example, Ducasse et al. [44] model software evolution by treating history as a first-class entity, represents evolution by means of a matrix and represents release history by a meta-model, focusing on evolution of properties. The lack of consideration of connectors and their evolution is a major impediment to the management of evolving software architectures with current methodologies.

2.5 Summary

We have surveyed the literature on techniques for modeling software evolution. Three relevant techniques are selected for the purpose of comparison in this Chapter, that is, Lucena and Alencar's logical framework, Medvidovic et al.'s architectural type theory and transformation techniques including a UML-based Algebraic Graph Rewriting and Fiadeiro et al.'s approach. We evaluate these approaches with two criteria: the representation of changes in architectures and the mechanism to evolve connectors. As a result, we find that two common problems with some of these techniques are that they either lack the granularity of connectors or have not established an effective mechanism for evolution of connectors.

Chapter 3

Architectural Connectors

3.1 Connector as a First-Class Architectural Citizen

Categorized as architectural elements, "Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and auxiliary mechanisms required" [137]. In the literature, connectors may also be named as *connections*, *bindings*, *component connectors* [95]. The concept of connectors also appears in middleware specifications, for example, J2EE connectors [1]. However, a connector in such a sense is out of the scope of this thesis since connectors are regularly hidden in the pre-defined middleware infrastructures, not easily extensible or evolvable, so that specification and description techniques are much less clear.

The main difference between components and connectors can be derived from their definitions, which we have presented in Section 1.3.2 (page 10). Components are computational elements independent of the context, to provide functionalities. In contrast, as the interactions among components, connectors are completely dependent on the context to connect components so that connectors may not "correspond to compilation units in an implemented system" [100].

For historical reasons, many theories, methodologies and tools still predominantly focus on components in design decisions, which makes connectors less obvious as compared to components. This fact leads to "fat components", where the code for connectors is interwoven with that of components at implementation time, and places the description of connectors implicit. Hence, connectors lose their identity, more or less. Many problems in modeling architectures are thus caused by the expressive shortcoming of "inadequacies of the

mechanisms for defining component interconnection" [134].

Addressed in Section 1.1.2 (page 5), a current trend is that business rules and their interactions constitute most of the complexity in software development. The ability to deal with the situation in the Software Engineering community, however, has not increased accordingly [107]. Connectors are exactly the corresponding elements which reflect the trend in the setting of software architecture.

Therefore, the idea of treating architectural connectors as first-class entities is proposed by Shaw [134]. Shaw models connectors as first-class entities to give the benefits for localizing interaction related code in connectors, increasing the reusability of components, enhancing the performance and maintenance of a software architecture, and supporting dynamic changes and evolution in system connectivity.

In light of connectors being first-class citizens, not only components but also connectors should be evolved at the architectural level. In the following sections, based on a survey of architectural connectors, we will introduce taxonomies of connector types, discuss notations and approaches for modeling connectors and explore a light-weight way to support architectural connector evolution.

3.2 Connectors Taxonomies

Connectors are complex and rich enough to deserve a taxonomy to show relations among similar kinds of connectors [134]. A connector type "expresses the designer's intention about the general class of connection to be provided by the connector; it restricts the numbers, types, and specifications of properties and roles" [136]. The fact that there are fewer connector types than components makes it easier to work with them. Admittedly, some taxonomies tend to be ambiguous in some aspects. But, in general, they are beneficial to understanding architectures, to optimizing the underlying mechanism, to facilitating implementation of family architectures and software development.

3.2.1 Architectural Styles

An architectural style typically "defines a vocabulary of components and connectors types, a set of constraints on how they can be combined" [137]. From the definition, we assume the knowledge that a style includes a set of specific connector types.

Some other novel terms have come up very recently such as "frameworks",

"architectural patterns[1]", "idioms", etc. In this thesis, we do not emphasize these distinctions, so that all of them are recognizing a pre-defined architecture at some abstraction level, representing a set of architectural instances. Any architectural style may have several variations. A software product can be constructed in or associated with more than one architectural style.

Table 3.1: Architectural Styles and Vocabularies

Architectural Style	**Component**	**Connector**
pipes and filters	filters	pipes
object-oriented organization	objects	messages, method invocations
event-based (implicit invocation)	events	procedures for certain event
layered system	layer	protocols of layer interaction
repositories	data-related structure or store	data access

Though not proposed for evolution, architectural styles are very appealing to identify connectors. A few styles are so named as to manifest connectors, e.g., pipe-and-filter and event-based style, etc., which facilitate further understanding of connector vocabulary, in some sense. Completely depending on the connector types appearing in these styles, however, would be considerably confusing, for identification and characterization of styles are not comprehensive. The concepts of such types in Table 3.1 are intentionally ambiguous on specific values of components and connectors presented. In addition, the mapping between architectural styles and their implementations is generally poorly understood.

3.2.2 Bures's Types of Component Interaction

Bures et al. [31] present a generic connector framework, reflecting middleware, to capture communication styles, where a communication style represents a "basic contract" among components. Though excluding connectors in the context of middleware infrastructures, we find Bures et al. [31] contribute a similar categorization in our context. Shown in Figure 3.1[2], communication styles generally fall into one of the four interaction types — procedure call, messaging, streaming and blackboard.

[1]Note that design patterns are not architectural.

[2]Figure 3.1 is adapted from [31].

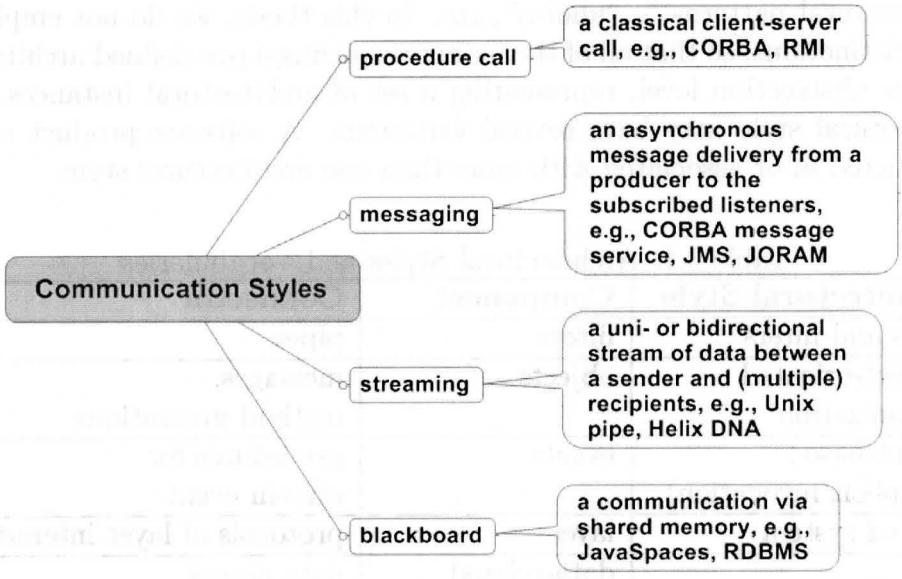


Figure 3.1: Bures's Communication Styles

The striking refinement is that Bures's connector types identify an initial framework to classify connectors. However, limited to the studied entities, connectors in middlewares are mostly variations of message-passing and RPC (Remote Procedure Call), which does not cover a large portion of connectors in architectural styles.

3.2.3 Mehta et al.'s Taxonomy of Connectors

To illustrate types of connectors with less ambiguity, Mehta et al.'s four-layer classification framework [101, 102] uses service categories, connector types, dimensions, sub-dimensions and values for dimensions or sub-dimensions. The service category comprises four groups of primitive services (interactions) — communication, coordination, conversion, and facilitation. *Communication connectors* support transfer of data among components. *Coordination connectors* support transfer of control among components. *Conversion connectors* convert the interaction required by one component to formats provided by another, so as to enable heterogeneous component interactions. *Facilitation connectors* mediate and streamline component interaction, providing mechanisms for facilitating and optimizing interactions among heterogeneous components. Instance connector types could provide one of these or composite connector types. The category generates 8 connector types: procedure call, data access, linkage, stream, event, arbitrator, adaptor, and distributor, see also Figure 3.2 (with an extra initial "T_" to denote *types* of connectors in the

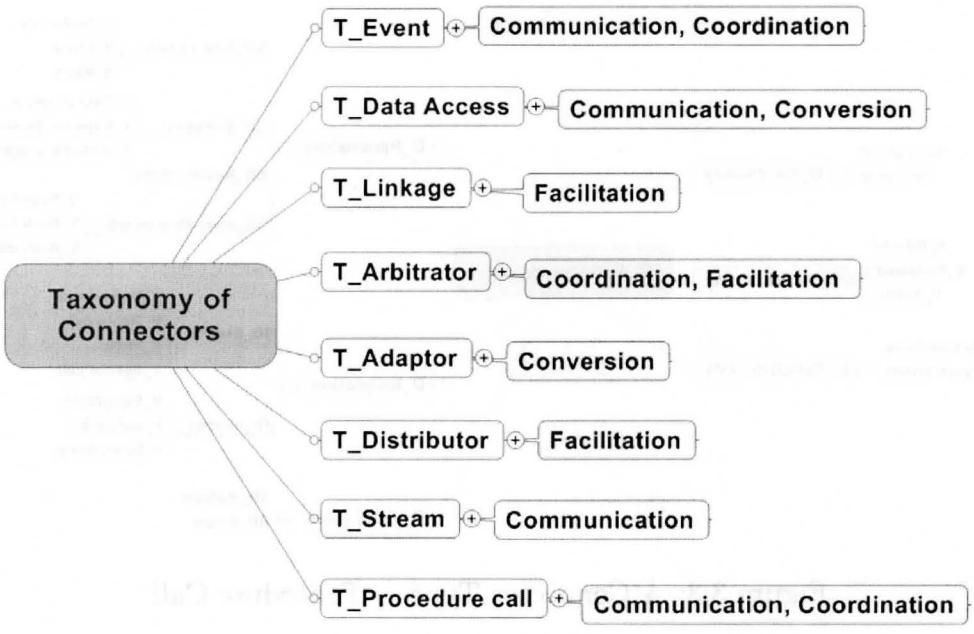


Figure 3.2: A Taxonomy of Connectors

four-layer classification).

- *procedure call* provides coordination and communication services. Such connectors model the control flow among components by invocation and transfer data among the interacting components via parameters; see Figure 3.3[3]. Typical examples are functions, procedures, object-oriented methods, callback invocations, operating system calls, etc.
- *event* provides coordination and communication services. Such connectors also model the control flow among components, as in a procedure call, except that the flow is related with events. Or the event messages carry information for communicating; see Figure 3.4. Typical examples are GUI events, interrupts and page faults caused by hardware.
- *stream* provides communication service. Streams perform data transfers between autonomous processes; see Figure 3.5[4]. Typical examples are Unix pipes, TCP/UDP communications.
- *data access* provides coordination and conversion services. Such connectors allow components to "access data maintained by a data store

[3]Initial "D_" denotes *dimensions*, "SD_" denotes *sub-dimensions*, and "V_" denotes *values for the dimensions or sub-dimensions* in the four-layer classification.

[4]Figure 3.2, Figure 3.3, Figure 3.4 and Figure 3.5 are adapted from [102].

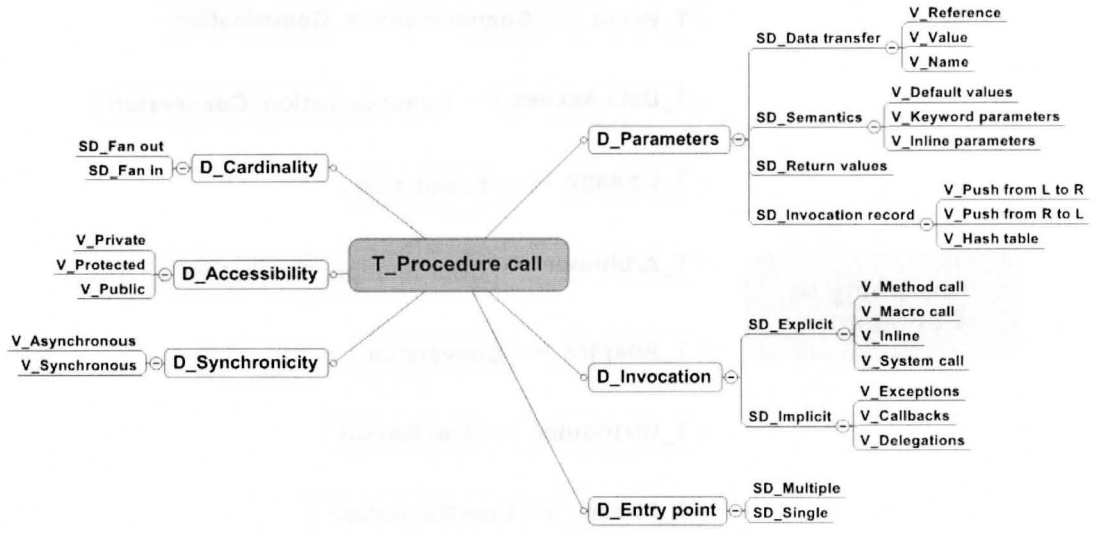


Figure 3.3: A Connector Type — Procedure Call

component" [101] and perform the conversion of data formats. Typical examples are SQL, File I/O, etc.

- *linkage* provides facilitation services. Such connectors combine the system components. Typical examples are linkage connectors in the C2 architectural style and Java dynamic class loader.
- *arbitrator* provides facilitation and coordination services. Arbitrators mediate system operations, resolve any conflicts (facilitation), and redirect the flow of control (coordination), when components cannot presume other existing components' needs and states. Typical examples are multi-threaded systems that require shared memory access use concurrency control.
- *adaptor* provides conversion services. Adaptors "provide facilities to support interaction between components that have not been designed to inter-operate". A typical example is virtual memory translation.
- *distributor* provides facilitation services. Distributors "perform the identification of interaction paths and subsequent routing of communication and coordination information among components along these paths". A typical example is DNS (Domain Name System).

In their later work, Mehta et al. [101] have tried to further develop the framework and characterize connector compatibility by means of a matrix, but it is not guaranteed to be orthogonal for each dimension. Regardless of this

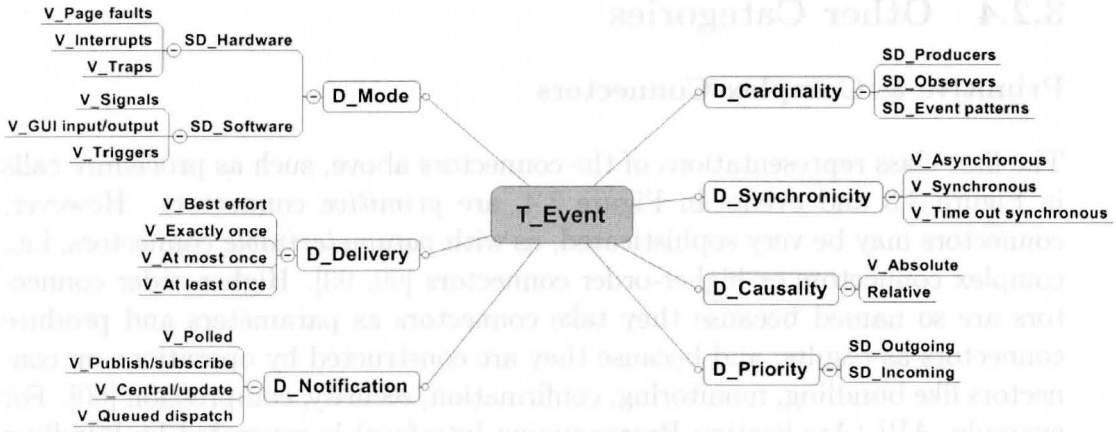


Figure 3.4: A Connector Type — Event

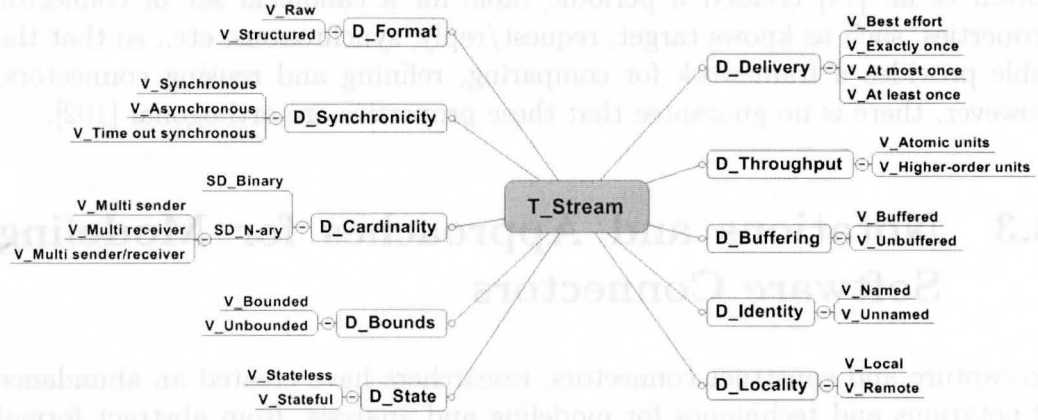


Figure 3.5: A Connector Type — Stream

weakness, the taxonomy is useful for building product family architectures [45] in terms of connectors, and also useful for directing connector evolution.

3.2.4 Other Categories

Primitive & Complex Connectors

The first-class representations of the connectors above, such as procedure calls in Figure 3.3 and events in Figure 3.4, are *primitive* connectors. However, connectors may be very sophisticated, as with *parameterizable* connectors, i.e., complex connectors or higher-order connectors [60, 93]. Higher-order connectors are so named because they take connectors as parameters and produce connectors as results, and because they are constructed by operations on connectors like bundling, monitoring, confirmation, security, compression [60]. For example, API (Application Programming Interface) is generated by bundling procedure calls to a single entity. By adding security features, a connector is enriched with encryption or authentication facilities. The data that a connector is transmitting may be compressed by compression operation. Confirmation operations can be applied on RPC connectors to acquire an acknowledgement or a verification. Monitoring operations are able to upgrade a connector to transmit communication services that a monitoring component may require.

Periodic Table of Connectors

Hirsch et al. [74] created a periodic table for a canonical set of connector properties, such as knows target, request/reply, synchronous, etc., so that the table provides a framework for comparing, refining and reusing connectors. However, there is no guarantee that these properties are orthogonal [102].

3.3 Notations and Approaches for Modeling Software Connectors

To capture and construct connectors, researchers have created an abundance of notations and techniques for modeling and analysis, from abstract formal models to practical languages, which endow connectors with rich semantics.

Allen and Garlan [11] were concerned with three properties for an expressive notation for connectors. Firstly, it should allow the specification of common types of architectural interaction; see Table 3.1. Secondly, it should be

able to describe complex dynamic interactions among components. Thirdly, it should allow for "fine-grained distinctions between variations of a connector".

3.3.1 ADLs

Wright ADL

Wright is a general purpose ADL created by Carnegie Mellon University, supporting specification and analyzing interactions between components (specifically, deadlock analysis), with *process algebra* as the formal basis. The goal of Wright is to define architectural connectors as "explicit semantic entities". Wright is implementation language independent, since for an architectural description in Wright, the ways to perform implementation are not specified.

A Wright architecture typically consists of three parts: component and connector types; component and connector instances; configuration of component and connector instances [11]. Wright components cannot be directly connected and are enforced to communicate through a connector. Unlike C2, Wright excludes the possibility that two connectors are directly attached to one another. A component type is "described as a set of *ports* (component interfaces) and a *component-spec* that specifies the component's abstract behavior". Connector types specification characterize the protocols of interaction between components provided by CSP-like notation (Communicating Sequential Processes). A connector type is composed of a set of *roles* (connector interfaces) to describe the expected local behavior of each of the interacting parties and a *glue* specification to describe how the activities of the roles are coordinated. Connector types in Wright are defined by users so that Wright allows arbitrary connector types. Component and connector instances are used to specify actual entities in configuration. These instances are combined into a configuration by the way how component ports are attached to (or instantiate) connector roles.

UniCon

UniCon (*Uni*versal *Con*nector) is an ADL developed at Carnegie Mellon University, which is intended to provide a rich selection of abstractions for the connectors that mediate interactions among components [136]. Supported by a library of built-in types of connectors, UniCon is known as a connector-oriented ADL.

Each UniCon component has an interface that defines computational rules and constraints. UniCon has defined a group of component types, such as

filter, SharedData etc. Connectors are specified by protocols defining how components may interact. The built-in connector types are Pipe, FileIO, ProcedureCall, DataAccess, PLBundler, RemoteProcCall and RTScheduler. According to type categorizations in Section 3.2 (page 32), we conclude that all connectors in UniCon are primitive. However, the pre-defined types of components and connectors are all enumerated so that there is no room for change, consequently support no evolution according to the definition in Section 1.3.4 (page 12).

ArchJava

Software architectures are described using specialized ADLs as we have learned in Section 1.3.3 and the above subsections, while implementations are described using programming languages. There is no intermediate language to guide the transition between these two phases, causing problems in the analysis, implementation, understanding, and even the evolution of software systems. Figure 3.6 shows the abstraction gap between specification and implementation.

ArchJava [9] is proposed with a motivation to bridge the gap, where an architecture is composed of a hierarchy of components communicating through explicitly described connections, and every component is an instance of a component class. Components in ArchJava are special kinds of objects described with an extended Java language, their connections allow components to communicate. Ports are the endpoints of connections [8] to represent a two-way interface with *provided* and *required* methods. Connections bind each required method to a provided method with the same name and signature. The goal of ArchJava is also to enforce *communication integrity* of components, i.e., components can only communicate with other components through interactions declared in the architecture.

ArchJava specifies component interactions — user-defined connector types with a clear relation to their implementation, and unifies architectural structure and implementation into one language, so that dynamic co-evolution of architecture and implementation is feasible.

Summary

So far, we have analyzed some ADLs in this section to show different perspectives to model connectors. First generation ADLs (e.g., Wright, UniCon) were constructed for specific purposes and few take evolution of architecture into considerations. Even fewer ADLs support evolution of connectors than do evolution of components. ADLs that do not model connectors as first-class

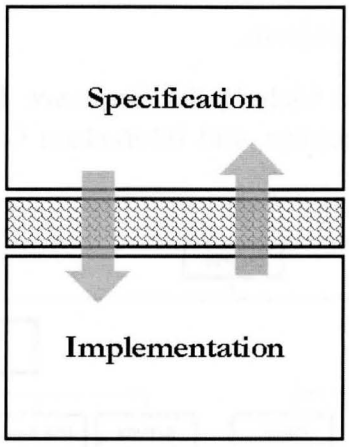


Figure 3.6: The Gap between Specification and Implementation of Architectures

entities (Darwin, MetaH, and Rapide) therefore provide no facilities for their evolution. ArchJava uses a single mechanism for specifying the semantics of both components and connectors. Coordination contracts, which we will see later in this Chapter, follow the idea of components as objects and user-defined connector types in ArchJava to separate concerns and facilitate evolution of connectors.

3.3.2 UML

Many specification languages have been introduced to specify and model the architectures of system. To explore the feasibility and evaluate the suitability of UML for modeling software architectures, research assignments such as [34, 63, 77, 97] have been accomplished as milestones. Motivation of these research comes from two needs. On the one hand, multiple design perspectives help software architects to build complex architectures easily, such as the Kruchten "4+1" view in Section 1.3.3 (page 11). On the other hand, ADLs have not been broadly applied in the industry, where "standard" notations and languages are generally required.

The UML 2.0 standard includes a collection of graphical notations, comprising 13 types of diagrams [29]; see Figure 3.7.

- *Structure Diagrams* include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.
- *Behavior Diagrams* include the Use Case Diagram, Activity Diagram,

and State Machine Diagram.

- *Interaction Diagrams* include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

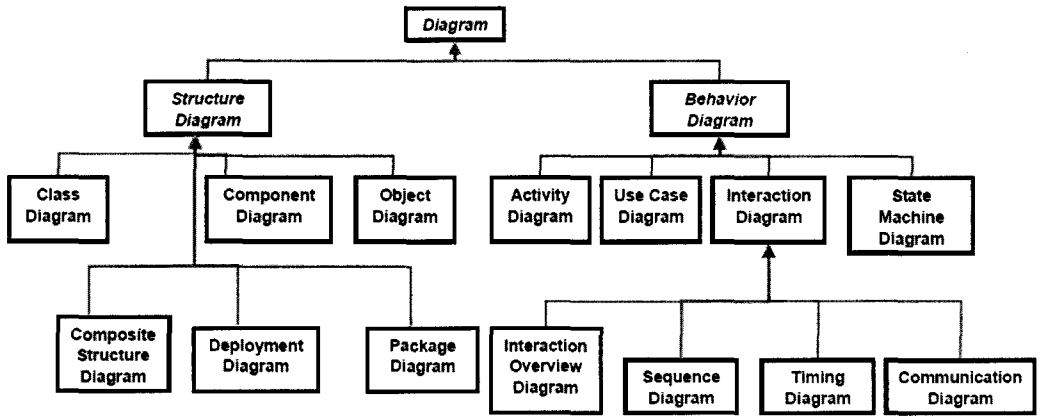


Figure 3.7: UML Diagrams

Documenting Connectors

"UML *as* an ADL" claims UML is able to provide strong supports for modeling software architectures. The key argument is that each of Kruchten's "4+1" views can be mapped into UML diagrams. For instance, the logical view is formalized by Class diagrams, the process view is mapped to Activity diagrams, the implementation and deployment views are modeled by Component and Deployment diagrams, respectively. The scenarios view is represented by Sequence and Collaboration diagrams, etc. Consequently, given the suitability of representing architectural views, UML once was promoted as a "universal" notation one-size-fits-all.

Despite weaknesses, when it comes to rigorous semantic concerns, UML 2.0 standard targets improving software architecture modeling and enhances the expressiveness for connectors. Informally defined, in UML 2.0, a connector represents "a communication link between two or more instances". The relationship between UML classes and objects can simulate that of component types and instances so that connectors can be documented as UML associations, UML association classes or UML classes [77].

A UML association is "a *structural* relationship among classes that describes a set of links, a link being a connection among objects that are instances of the classes" [29]. For example, in Figure 3.8, the link "≪pipe≫"

between two *Filter* objects simulates a pipe connector. An association class is "a modeling element that has both association and class properties, which can be seen as an association that also has class properties or as a class that also has association properties" [29]. For example, in Figure 3.9[5], an instance of association class *Pipe* expresses a pipe connector. Association classes have richer semantics, as compared to associations, e.g., providing attributes, behavioral descriptions and even substructures, hence is more expressive.

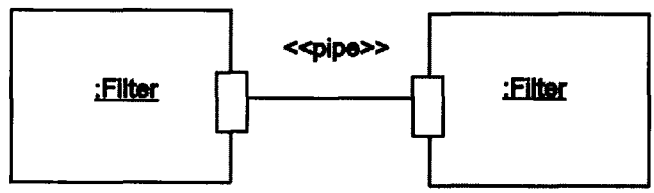


Figure 3.8: UML Connector as an Association

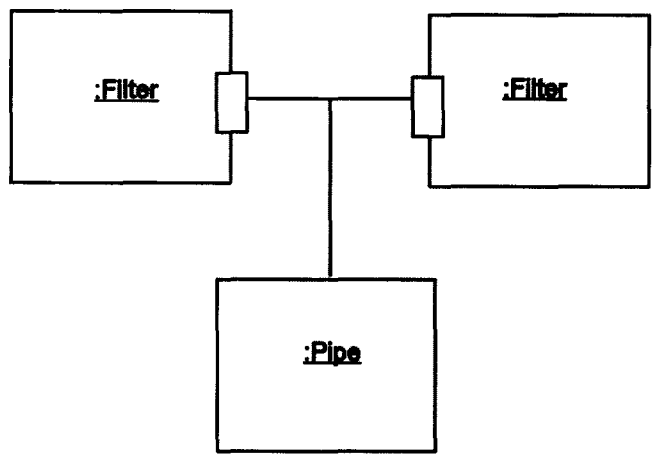


Figure 3.9: UML Connector as an Association Class

Documenting Behavior Views

Software architecture derives behaviors of a system from the behaviors of architectural elements, i.e., components and the way they interact through connectors. We focus on *functional* behaviors in this context.

Documenting behaviors of a software architecture is essential since it exhibits the functionality of each component and interactions between components. There are many ways to specify behavior of the elements in an architectural model, ranging from plain English to sophisticated formal methods.

[5]Figure 3.8 and Figure 3.9 are adapted from [77].

Garlan [62] summarizes some formal techniques applicable, such as pre- and postconditions, process algebras, statecharts, POSets (Partially Ordered Set), rewrite rules, and the like. Behavior diagrams in UML can also be used to document behaviors of software architectures. It is worth noticing that in UML interaction diagrams are derived from the behavior diagrams (Figure 3.7).

3.3.3 Formal Notations

Formal notations are used to put architectural connectors on a more "solid" footing, such as FOL, process algebra (CSP), π-calculus and category theory. Many ADLs depend on an underlying semantic model. For example, Wright ADL, introduced in Section 3.3.1 (page 39), models connector glue and event trace specifications with CSP. Category Theory is a branch of mathematics that provides "universal constructions to describe properties of mathematical structures like sets, groups, graphs etc" [48, 125]. As discussed in Section 2.3.2 (page 27), CommUnity [52] is an ADL with category theory as its formalism basis.

Some formalisms describe connectors without support of architectural languages, for example, Barbosa et al. [25] specify connectors by using co-algebras. Formal notations and approaches have their place in modeling connectors. However, the high cost of using formal methods prohibit their application in industries to a certain degree, except for safety or security critical systems.

3.3.4 Coordination Contract

Figure 1.2 (page 10) has illustrated that architectures are not always described at a higher level of abstraction. Such a lower level architecture is required because the high level software architecture falls short in support for software engineers at the important level of program abstraction. Additionally, in Section 3.3.1 (page 40) we learned that, as a representative of existing ADLs, ArchJava provides support for evolution by treating an architectural description as a conventional program and relying on a special implementation language, which is limited.

To bridge the gap between the architectural level and implementation level of connectors and to direct evolution in a wider spectrum by separating concerns, we are in need of a substantially independent language for connectors from the programming languages of the components. It turns out that Andrade and Fiadeiro have introduced the notion of coordination contract as an alternative to connectors when developing financial systems for Grupo Espírito Santo in Portugal [13]. Modeled as a first-class abstraction, coordination contracts

are able to ensure that global properties will emerge, represent connectors throughout the entire application life cycle, and seriously enhance component reusability. This section is a summary of [14, 17, 19, 68, 86] and in Chapter 4 we will introduce coordination contracts in a systematic way.

A coordination contract is a modeling and implementation primitive that allows "transparent interception" of method calls and, as such, interferes with the execution of the service in the client [19]. Thus, coordination contracts take over the function of connectors in software architectures. The evolutionary operations defined in Section 1.3.4 (page 12), such as adding, removing and replacing connectors, will be defined via coordination contracts instead without breaking into the functionality of components.

The construction of coordination contracts consists of a collection of constraints and rules describing coordination effects (as the *glue* of connectors) that are superposed on the involved component partners, which then enables evolution of connectors as a means of localizing change.

What is Coordination?

Coordination models and languages are used to provide a specification-level description of detailed architectures, to enforce separation of components and connectors [76], where the two dimensions are able to change at a different rate with less impact on each other. Generally, coordination contracts are used to implement connector types in Section 3.2.3 (page 34), such as procedure call, data access and event. However, it is worth mentioning that the notion of *coordination* in "coordination contract" is different from that of Mehta et al.'s classification of connector types, which refers to transfer of control among components specifically. We claim that coordination contract provides a foundation for a general notion of connector or even higher-order connector.

Coordination in "coordination contract" is based on *superposition* (or superimposition) in parallel program design [138]. A superposition denotes a structure preserving transformation on designs through the extension of their state space and control activity while preserving their properties [52]. The notion of coordination describes process interaction by abstracting away the details of computation and focusing on the interactions [21], as well as bridging the gap between the high-level architecture and detailed architecture so that interacting and evolving processes are managed.

What is a Contract?

Like many terms, the concept of contracts is overloaded in the literature and supports different design intentions. In legal terms, a contract involves agreement, consideration, certainty, order or intention, etc. As opposed to it, contract serves as a functional specification in software engineering society[6] and often stands for *Design by Contract* ™(DbC), pioneered by Meyer [105], which is widely acknowledged as a powerful technique for constructing reliable software. DbC is used to build the relationship between a class and its clients, using a formal agreement, and to express each party's rights and obligations [106]. The three key ingredients of DbC are pre-conditions, post-conditions and class invariants. However, its semantics supports only pre- and postconditions of methods and invariants of individual classes.

Coordination contracts are different from contracts in DbC because they are superposed on classes in stead of being operated on a single class. However, we can still consider coordination contracts such as an "extension" of DbC [49]. One of the principal purposes of coordination contracts is to facilitate evolution of connectors in program architectures.

3.4 Summary

In this Chapter, we demonstrate the significance of connectors being represented as first-class architectural citizens. We also explore several taxonomies of connector types, and study Mehta et al.'s taxonomy in detail. Notations and techniques for modeling connectors have been discussed, such as Wright ADL, UniCon, ArchJava. The feasibility of UML *as* an ADL to modeling software architectures with UML is presented. In the end, a new light-weight way is suggested to support evolving architectural connectors. As a realization of connectors, coordination contract is a modeling and implementation primitive that allows transparent interception of method calls and, as such, interferes with the execution of the service in the client. The evolutionary operations defined in Section 1.3.4 (page 12), such as adding, removing and replacing connectors, are defined via coordination contracts.

[6]Different meanings of contracts also appeared in [22, 46, 73].

Chapter 4

Coordination Contract

4.1 Introduction to Coordination Contract

In Section 3.3.4 (page 44) we have presented a general overview of the coordination contracts. The separation of architectural connectors from components allows for the explicit representation of object interactions in the form of contracts, communication objects or connectors. Coordinations and coordinated entities are independent so that they can evolve separately. Coordination contracts define a declarative modeling language, other than a programming language. In the following subsections, we will inspect the ability of several popular techniques to support modeling of architecture based software evolution, such as Object-Oriented Design (OOD), design patterns, AOP (Aspect-Oriented Programming) and the association class in UML.

Object-Oriented Design

Before exploring the mechanisms of coordination contracts, we test similar ideas in OO. As we talked about in Section 1.1.2 (page 5), the development complexity is focused now on software evolution rather than construction. This phenomenon has a ripple effect on the methodologies previously adopted for software construction. Those methods should be re-evaluated for the purpose of evolution, for example, the most popular one, Object-Oriented Design.

OOD provides only two ways to "use" a class, to *inherit* from it or to become a *client* of it [105]. We define new subclasses by reusing the behaviors of an existing class or we establish client/supplier relations between objects through feature calls. With the two characteristics, OOD broadly encapsulates data, controls the construction complexity and enhances the reusability

of software artifacts. For example, a minimal version of BlueJ[1], an Integrated Development Environment (IDE) for Java applications, only "uses" and "inherits" associations are supported.

However, objects are abstracted as "white-boxes", in the sense that any subtle changes require knowledge of implementation details and will be performed on their internal structure, which is not desirable for evolution. On the other hand, the *clientship* practice makes components highly coupled, which contradicts principles of flexible interactions discussed in Section 1.1.2 (page 5) and thus incurs the solicitation to separate components and coordinations. However, most OOPLs have not the corresponding language construction for connectors, and then interactions are directly implemented in the code for components.

Mechanisms, such as *inheritance* and *clientship*, do not provide connection as a first-class entity, like contracts, which leads to inheritance and composition being even more intrusive. As a consequence, OOD does not facilitate evolution in relation to our concerns.

Design Patterns

The similarities of Design Patterns [58] to architectural styles drive one to consider the possibilities of modifying patterns to support the architecture-level evolution. To clarify this, we distinguish *architecture* from *design* first. Software architecture is concerned with architectural elements, their interactions and related constraints on these elements and interactions [124]. Design is concerned with "the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements" [124]. Once instantiated, component interactions implemented by design patterns require evolution to be intrusive because they were not initially conceived to be evolvable.

Such patterns do not provide modeling of connection as a first class entity like contracts do. The behaviors of components are scattered in compound classes. Therefore, design patterns have more to do with capturing design construction rather than dealing with the evolution of software. We will learn later in this Chapter how a micro-architecture based on design patterns is generated for transforming contracts into a chosen implementation platform.

[1] `http://www.bluej.org/`

Aspect-Oriented Programming

In addition to the separation of concerns (SoC), AOP [79, 80] aims to model cross-cutting concerns specifically when composing software artifacts. AOP is also nominated as a post-object-oriented programming paradigm. However, the notion of contract is not cross-cutting, as is explained by Balzer et al. [24]. Besides this, AOP is operating at a very low abstraction level, which cannot satisfy our requirements at architectural level. AOP is principally dealing with modification or evolution at implementation time. We will not benefit from *aspectization* during design time if component interactions are implemented by coordination contracts.

To conclude, the techniques such as OOD, Design Patterns or AOP have no mechanism to prioritize connectors as first-class entities, so that the behaviors of interaction between components merge in classes.

The Association Class in UML

From the discussion in Section 3.3.2 (page 41), an association class can be treated as an association that is also a class and has both association and class properties [115]. Though a coordination rule should be processed as an atomic transaction which cannot be satisfied by association classes, the facilities that association classes model class interactions as first-class citizens and the sufficient interior organization are appreciable. In later sections we will show that a coordination contract is a variation of association classes whose semantics rely on principles used in software architectures and coordination languages and reflect changes in the business rules. Coordination contracts externalize the interactions between participant components (objects) and support evolution of systems with respect to changes of business requirements in a compositional way. To achieve this goal, contracts require a rich internal structure with private attributes and operations since the components should not be able to access these features.

4.2 Contract Abstraction Levels

We have claimed in Section 3.3.4 (page 44) that coordination contracts are operating on a different level of abstraction in contrast to DbC. Beugnard et al. [28] presents a four-level contracts model as in Figure 4.1[2], where each level corresponds to a class of contracts.

[2]Figure 4.1 is adapted from [28].

- *Basic contract* (or syntactic contract) is required simply to make the system work. Interface specifications are as contracts between a client of an interface and a provider of an implementation of the interface [142, page 43]. Typical examples are Interface Definition Languages (IDLs) and typed OOPLs.

- *Behavioral contract* improves the level of confidence in a sequential context; typical examples are DbC applied in the Eiffel language, pre- and postcondition and OCL.

- *Synchronization contract* specifies the global behavior of objects in terms of synchronization between method calls, so that it improves confidence in distributed or concurrency contexts.

- *Quality-of-service contract* (non-functional) specifies all behavioral properties, including even non-functional properties like availability, throughput, latency and capacity.

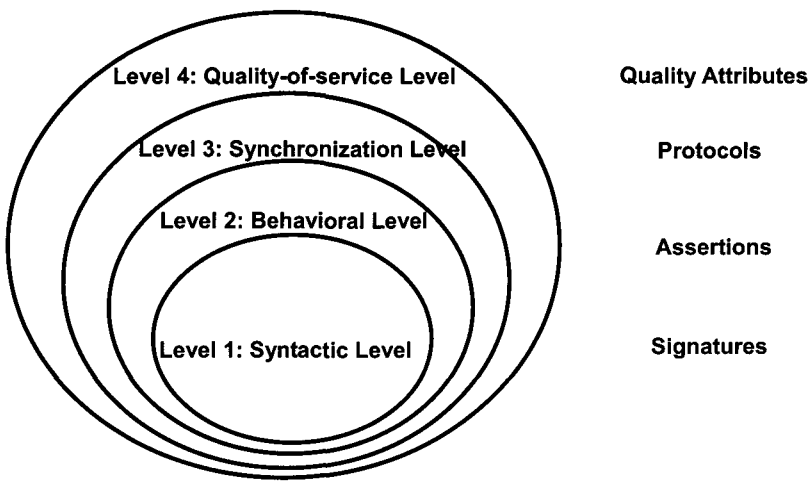


Figure 4.1: Level of Contracts

By virtue of this model, the proposed coordination contracts belong to level 3 — synchronization contracts. It is true that we will present an approach to justifying predictable software evolution using pre- and postconditions in the next Chapter. Concerning this, one may argue that such contracts mainly are of level 2 as behavioral contracts. We undertake that coordination contracts are synchronization contracts essentially for three reasons. First of all, coordination contracts are used as a top-level abstraction to specify the global behaviors of objects, synchronize objects and superpose behaviors upon components which reside in a distributed or concurrent environment, to provide

more flexibility in terms of connector evolution. Moreover, we take the advantage of pre- and postcondition as a facility to justify relationships between contracts for predictable software evolution, where coordination contracts have their specific syntax and semantics independent of these pre- and postconditions. However, the level 2 contracts, in the role of behavioral contracts, are structured in the form of pre- and postconditions. It is also important to notice that coordination contracts are working in both design modeling and implementation phases.

Collet et al. [35] consider that a contract must provide: a specification formalism; a rule of conformance, to allow substitution; a runtime monitoring technique, if the contract cannot be enforced before runtime. In the rest of this Chapter we will present specification languages for coordination contracts and a runtime environment but will leave the rules of conformance to the next Chapter.

4.3 The Three-Layer Architecture

In the architecture model adopted when using coordination contracts, to separate coordination from computation, connectors are modeled explicitly as first-class entities and evolved by reasoning about coordination contracts between components so as to reflect changes of business rules. By architecture, we mean the structure of applications at the program level, which is also called detailed architectures in Figure 1.2 (page 10). The three-layer architecture [19] applied on coordination contracts is organized into computation layer, coordination layer and configuration layer, see Figure 4.2[3].

- *Computation layer* models business entities and encapsulates functionalities of services performed locally in components. Components should be as simple as possible, providing only the core functionalities not vulnerable to be changed.
- *Coordination layer* models business rules and consists of contracts coordinating interactions between components so that the global properties are able to emerge. The purpose of contracts is exactly to provide mechanisms for this layer to be modeled and implemented in a compositional way [14].
- *Configuration layer* models business context and manages the current configuration of contracts and components (also called "coordination context") where the reconfiguration operations and rules for evolution

[3]Figure 4.2 is adapted from [19].

are executed. We constrain contracts as connectors to not have a direct relation with other contracts.

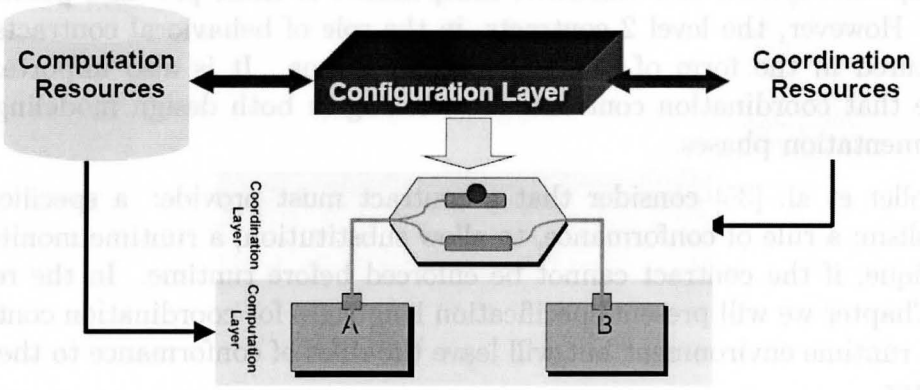


Figure 4.2: A Coordination-based Three-Layer Architecture

4.4 Notations

We claim that coordination contracts are declarative since they regulate what must hold instead of what should be done to enforce a contract, so that we are not worried about enforcement or penalty for contract breach. The semantics for contracts is based on CommUnity, as described in Section 2.3.2 (page 27) and 3.3.3 (page 44). Fiadeiro and Andrade [49] used the notion of superposition for parallel program design to define the semantics when multiple contracts manage the same components. For brevity, we will show its semantics informally while presenting the syntax.

4.4.1 Graphical Notation

UML is used to model the three-layer architecture in Section 4.3 (page 51) with the aim to describe, represent and analyze systems. Though UML 2.0 is still not fully formal, we will take advantage of the features it provides and illustrates a coordination contract as a "scroll" by extending a notation for UML, see Figure 4.3. With this extended notation, we are able to enrich UML to model architectures with coordination contracts. Sometimes, a coordination contract is called an *association contract* [85].

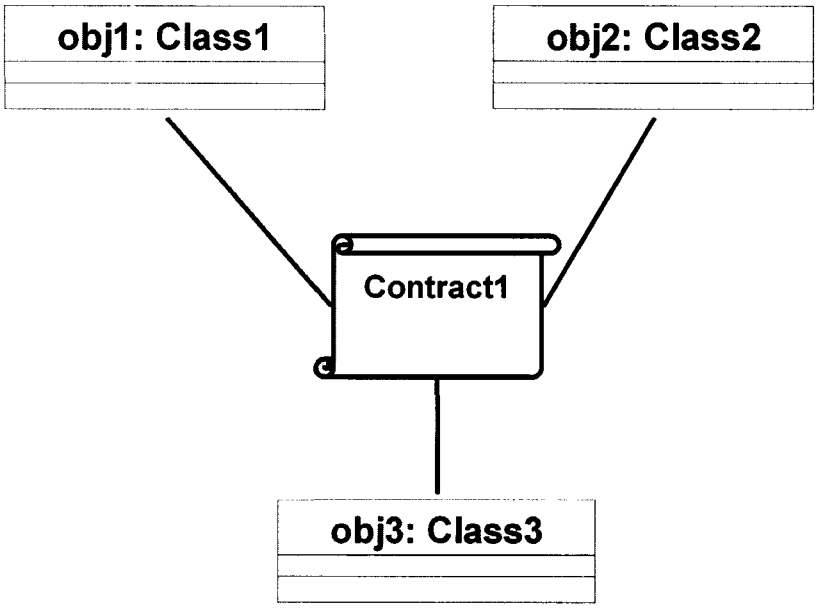


Figure 4.3: An Example Architecture with a Coordination Contract

4.4.2 Textual Notation

In textual notations for contracts, we identify what is needed to be coordinated (i.e., entities, processes and resources) and how these objects are coordinated (i.e., rules). The language of coordination contracts is originally from the OBLOG (OBject LOGic) specification language [112]. The language for contracts is ideally independent from the language for objects. In the literature of coordination contracts, two levels of representation are described: an implementation neutral language, which is abstract, free of technical details and specifying only business rules, and an implementation specific language, which is a refinement of the former to include technical details in Java, supported by CDE[4] (Coordination Development Environment), a software environment for coordination contracts. Ideally, the relationships between these two levels and the fundamental semantics are shown in Figure 4.4 [5]. If there was a definite formalization of the operational semantics we could have proved that the translation between the abstract and CDE specification is correct by proving that the diagram in Figure 4.4 commutes.

[4]CDE will be introduced in Section 4.6, page 63.

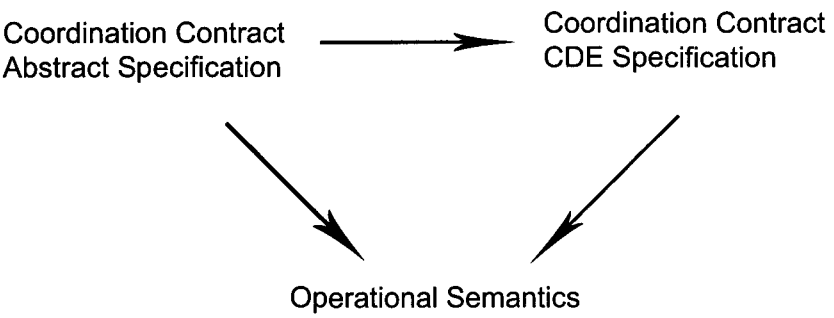


Figure 4.4: The Abstract and CDE Specification of Coordination Contract

The Abstract Language

The abstract version of specification is very straightforward, independent of specific choices of design languages and behavioral models, where a coordination rule is shown below:

Table 4.1: The Abstract Language

```
contract <name>
   participants <list of component instances>
   constants <local constants>
   attributes <local variables>
   operations <local methods>
   invariant <properties required>
   coordination <coordination rules>
end contract
```

The `name` of a contract is its *unique* identity in the coordination layer. `participants` are instantiated participant components from the computation layer with coordination interfaces. These instantiated participant components exist as UML classes or Java objects, which make assumptions about their behaviors interacting with other instances. The state of an object is given by the set of values of the object's attributes and interfaces changing the attribute values through *setter*s and *getter*s. Contracts can operate on one or more objects. A unary contract is allowable on a single object. As we noted when talking about representing coordination contracts by means of association classes in UML, a contract may have private attributes, constants, operations and invariants. The `attributes` clause declares variables for an instance contract and `constants` can be thought of as special kinds of attributes whose values do not change. An `invariant` states the properties required to be true when executing the contract. Invariants and constants are important for reasoning

about a contract. The difference between them is that a predicate p is an invariant if p is true at all times in the execution, and a predicate p is a constant if it either always remains true or always remains false.

Under the `coordination` clause, coordination rules are defined to perform actions on participant components. An action is a primitive operation whose execution is like an atomic, uninterrupted transaction guarded by a predicate. Either all the actions that the rule describes will happen, or none of them will. Thus, an action is a "step" relation between two sets of values of state variables. When an action can be executed, we say it is *enabled* [107]. The language regulates that coordination rules cannot be nested.

Coordination Rules Specification

Table 4.2: Coordination Rules Specification

coordination rulename:
`when` <triggers> by keyword "AND" to extend trigger conditions
`with` <condition> guards
`do` <set of actions>

A rule is activated by the trigger given in **when**, and superposes the service/reaction/method in the rule body. The triggers can be a conjunction (by logical operation "AND") of conditions on the state of participants, requests for a certain service, or messages received by one or more participants, etc. All these conditions should be satisfied at the same time, or the contract would be marked *inactive* and the same trigger in other contracts (if there is any) or the method call in the original component will be executed. There is the *only* place to decide if a rule will be triggered and whether a method invocation will be intercepted.

Rules are guarded atomic actions, where guard conditions are Boolean conditions composed of predicates. If the guard conditions are omitted, their effects will be evaluated to be **true**. The guard in the **with** clause uses local variables or state conditions of components, for example, $(x >= y) \wedge (obj == o)$. When the occurrence of given triggers is detected, the firing of an action is possible if its guard holds true. For example, the *allowWithdraw* rule in Figure 4.6 in the *Ownership* contract may only fire when $a.balance() >= n$ is **true**[5]. If any condition under the **with** clause is not satisfied, no actions in the rule will be executed, and instead, we will see it later that an exception will be thrown as a result.

[5]We will learn in Chapter 5 that trigger is not the only decisive condition.

A more specific notation for contracts supported by CDE [6] is as follows; it is dependent on the Java language [2].

The Language Supported by CDE

In practice, we desire to carry through the spirit of "separation of concerns" so that we model components and connectors by different abstraction levels of languages, with the former in the Java programming language and the latter in the contract language supported by CDE. The contract language details will vary depending upon the implementation techniques and standards that are selected. For the language supported by the latest version of CDE, v1.1.1, coordination contracts are compiled into Java [67, 68]. However, we have to admit that contracts will lose generality when interpreted into a specific language. Fortunately, the other side of a coin is that we thus manage to integrate a rich architectural description into a mainstream programming language. Such a representation can be used by technical people for implementing contracts. The current version of the contract language CDE supports is a combination of Java code and fragments from an abstract specification of contracts.

`participants` are existing Java classes as components and implemented by Java source code documents, defined in the form of "participantName: Class;" or "Class participantName;". `attributes` are declared as in Java syntax. These attributes are private to the contract, therefore, no modifiers such as `public`, `private` are required. For example, we declare "`double` balance;" instead of "`private double` balance;". CDE will automatically generate two public methods for each attribute: a *setter* to set its value and a *getter* to get the current value. For instance, the setter for "`double` balance;" is "`void` setBalance(`double` _balance){balance = _balance;}", and the getter is "`double` getBalance(){`return` balance;}". As to the underscore notation before the variable name, as in "_balance", we will explain its role later in Section 4.5.1 (page 60). `operations` are local to the contract and conform to Java language conventions as well.

In the coordination rule section, the CDE specification has refined the abstract version and provided two kinds of coordination rules: the *TriggerRule* for intercepting method calls on participants, with the form of "`when` $\star-$ >> participant$i$.operation(arguments)", and the *StateConditionRule* for rules reacting to state conditions, with the form of "`when` ? (condition in Java) on participant1, participant2...". State condition rules are declared initially by a question mark (?) with the condition statements specified in Java. Only statements that do not change the state of the participants may be used to define conditions in state rules; they are sometimes called *query* operations.

In principle, all public methods provided by the participants can be coor-

Table 4.3: The Language Supported by CDE

```
contract contractName
  participants //contract participants
    participant1: Class; //or declared as "Class participantName"
    participant2: Class; //e.g., obj: Object;
    ...
  attributes //private attributes of the contract
    JavaType name 1; //e.g., int limit;
    JavaType name 2;
    ...
  operations //private operations of the contract
    //operation body in Java
  coordination //rules
    TriggerRuleName:
      when *- >> participanti.operation(arguments)
            && (trigger conditions in Java)
      with (JavaGuardConditions)
      failure (Java guard failure actions throw
               an exception or return a value)
      before {actions in Java to be executed before
               participanti.operation(arguments) };
      do {actions in Java to be executed replacing
           participanti.operation(arguments) };
      after {actions in Java to be executed after
               participanti.operation(arguments) };
    StateConditionRuleName:
      when ? (condition in Java) on participant1, participant2...
      do {operations in Java};
end contract
```

dinated, except for the constructors and the Java operations from Java API (e.g., `toString()`). When the trigger in "`when` *- >> participant1.operation (arguments) && (trigger conditions in Java)" is a call like *- >>`obj.x(a)` (*- >> means any call to that operation), it denotes the call of method x on object *obj* with the sequence of arguments a. The contract is listening to the actual interaction x, and will intercept the call and superpose the forms of functional behaviors it prescribes.

If any condition under the `with` clause is `false`, none of the actions in the rule is executed. Accordingly, exceptions will be thrown and failures will be reported to the object that called the operation. The notion of exception is represented here as the `failure` statement dealing with errors not able to be

processed by contracts. The `failure` clause should be ended with either a `throw Exception` or a `return` statement.

In Figure 4.5[6], `before` actions are performed before `obj.x(a)` is executed. `after` actions are carried out after the execution on `obj.x(a)`. The `do` action is carried out to replace the original method body of `obj.x(a)` (if the trigger occurs and the guard is `true`), or the original operation will execute if there is no `do` block (namely, `do` is omitted as a shorthand). The semantics of contracts regulate that only one `do` block is executed at the same time to prevent the conflict of two alternative valid actions for the same trigger. If *rule1* in *Contract1* and *rule2* in *Contract2* are triggered for the same call `obj.x(a)`, all guards will be evaluated first. If any guard is `false`, the corresponding `failure` part is executed and actions in that rule abort. If all guards are `true`, then all `before` actions will be executed, interleaving them without conflict. The `do` action, however, will be assigned to the first coming contract rule by the time stamp when created. When the `do` actions finish, all `after` actions will be executed.

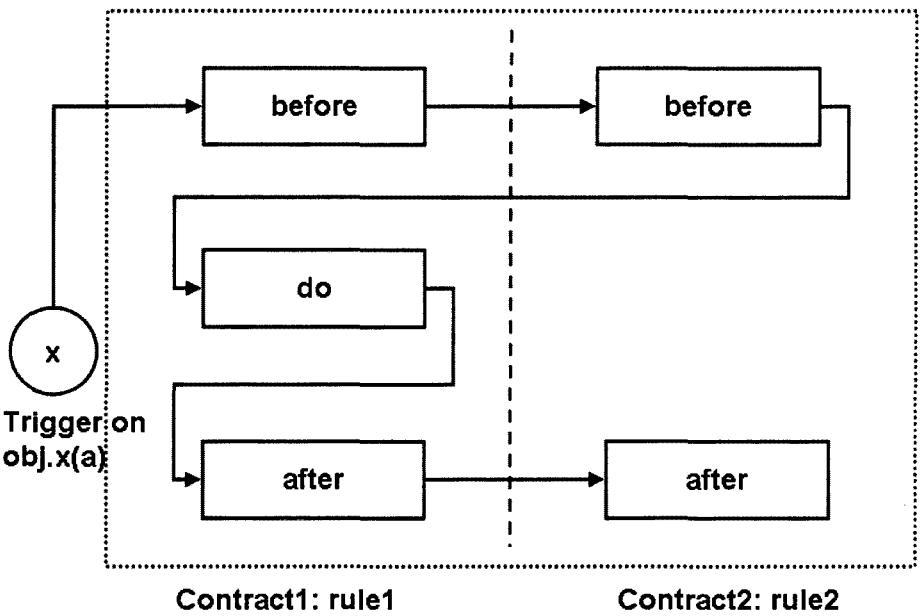


Figure 4.5: Multiple Coordination Contracts

Lano and Fiadeiro [85] present a more general version of specification with `extends` to build a composite contract. The semantics of the contract inheritance mechanism and nested contracts were not yet clear at the time of developing this thesis. Such an extended contract is informally called a "subclass contract", where rules and features will be inherited and overridden. In

[6]Figure 4.5 is adapted from [19].

Figure 4.6[7] the new contract *VIPownership*, which "inherits" from *Ownership*, has a weaker `with` guard, i.e., $a.balance >= n \Rightarrow a.balance + limit >= n$ (suppose $limit >= 0$), and they both are triggered by a call to $c.withdraw(n, a)$.

In a coordination contract, we are able to identify the similar three kinds of expressions that DbC delivers: preconditions, postconditions and invariants. In the next Chapter, we will make efforts to explain the meaning of `extends` and design a method to preserve the functionality of the original operation in the `do` action by means of a pre- and postcondition specification in order to ensure predictable evolution.

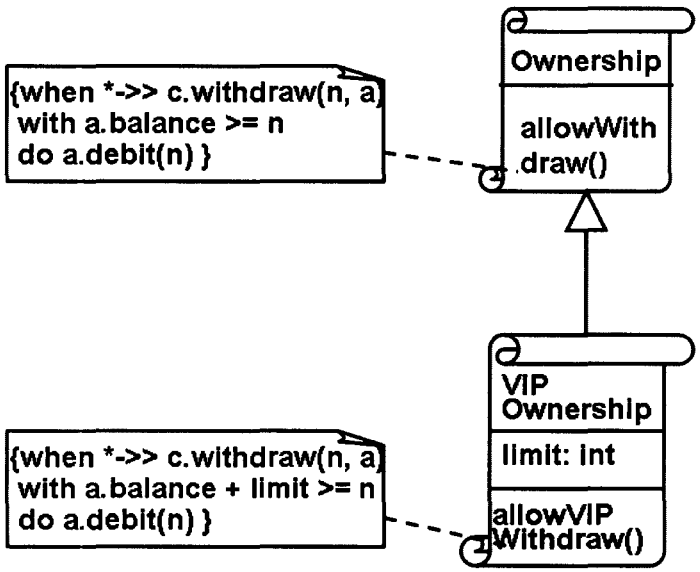


Figure 4.6: A Proposed Contract Inheritance Mechanism

4.5 Patterns for Coordination Contract

Fiadeiro and Lopes [51] proposed a categorical pattern for coordination and provided a clear separation between components and contracts in a formal way. To implement the categorical pattern where the contract is language-independent, [12, 20, 68] came up with a micro-architecture (design pattern) that can be used to implement contracts in standards for commercial component models like CORBA (Common Object Request Broker Architecture), JavaBean, DCOM (Distributed Component Object Model) and .NET. The generic micro-architectures managed, with well-known design patterns [58] (such as Proxy and Chain of Responsibility), to transform contracts into working environments that the solution required. However, the mathematical mapping

[7]Figure 4.6 is adapted from [85].

between categorical pattern and micro-architecture is not available due to the lack of support formalisms in those business component standards and chosen platforms.

The micro-architecture[8] consists of two parts: mechanisms for components in the computation layer, and mechanisms for the contracts in the coordination layer. Figure 4.7[9] shows one feasible solution.

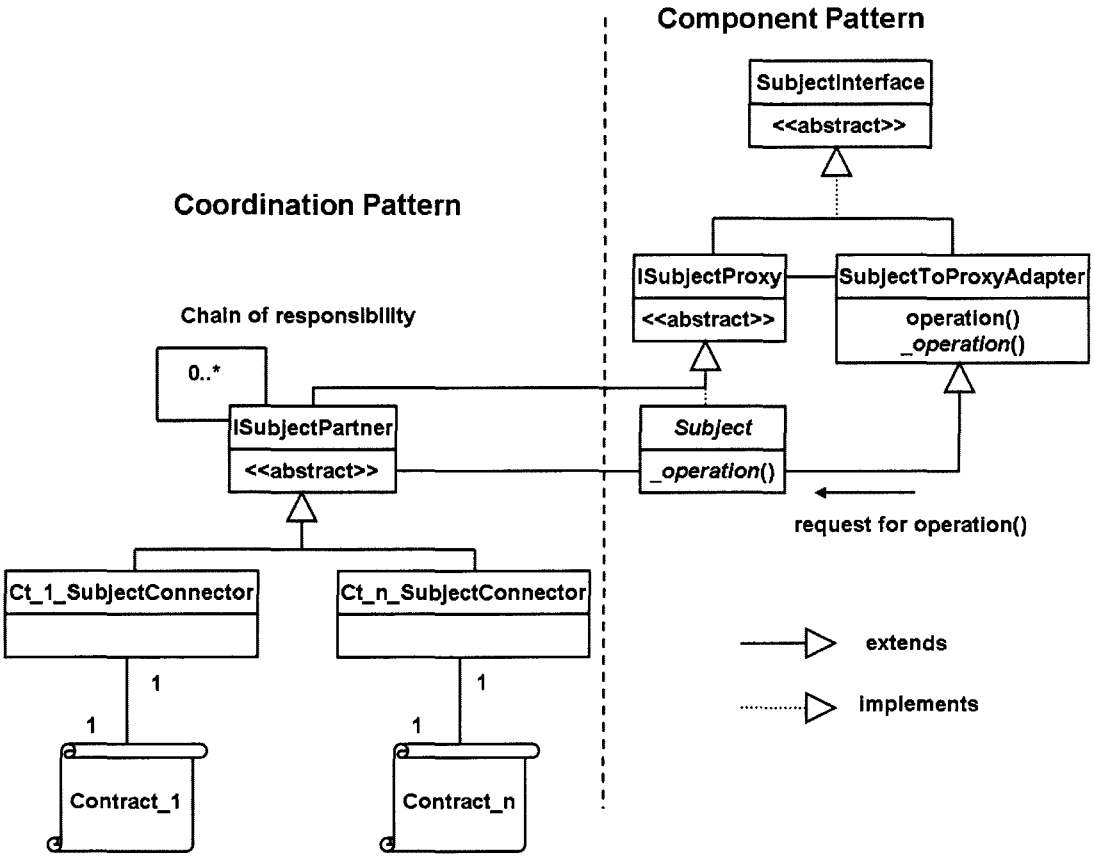


Figure 4.7: A Design Pattern for Coordination Contracts

4.5.1 The Component Part

The classes in the component part are organized by using the Proxy pattern, which provides a surrogate or placeholder for another object to control access to it.

[8]There may be alternatives for implementing coordination contracts other than this one.
[9]Figure 4.7 is adapted from [68].

- *SubjectInterface* is an abstract class (type) where the public operations under coordination are predescribed. According to the "implements" relationship of classes in Figure 4.7, the interface consists of the common interface of services in *SubjectToProxyAdapter* and *ISubjectProxy*. *SubjectToProxyAdapter* and *ISubjectProxy* must implement, or realize, the behavior specified by *SubjectInterface*.

- *Subject* is a concrete class with implementation subjected to coordination. The class extends *SubjectToProxyAdapter*. Before executing the object, *Subject* intercepts the service request on the object, and endows the contract with rights to judge the validity of the request and rules to perform, so as to realize the superposition of the contract.

- *SubjectToProxyAdapter* is a concrete class which enables the proxy pattern for the original methods in *Subject*, and employs proxy at run time, for delegating requests to *ISubjectPartner*, which links the *Subject* to its relevant contracts. We will show later that if no contract (proxy) is defined, it forwards requests directly to *_operation*() in *Subject*.

- *ISubjectProxy* is an abstract class that defines the common interfaces of *Subject* and *ISubjectPartner*. It represents an object with the capability of implementing the *Subject* interface. The interface is inherited from *SubjectInterface* to guarantee that all these classes offer the same interface as *Subject*, with which real subject clients have to interact.

4.5.2 The Coordination Part

- *ISubjectPartner* contains the connection between *Subject* as the real object and the contracts. *ISubjectPartner* delegates the received requests to *Ct_i_SubjectConnector* using the Chain-of-Responsibility pattern.

- *Ct_i_SubjectConnector* represents the particular coordination for those contracts where *Subject* is a participant. For each contract superposed, there is exactly one *Ct_i_SubjectConnector*.

- *Contract-i* is a coordination instance that will be superimposed on *Subject* as the real object.

If there are no contracts coordinating a real object, the micro-architecture can be simplified. For the component pattern, there is a call from *SubjectToProxyAdapter* to *Subject*, which is shown in Figure 4.9 as a call from *AccountToProxyAdapter* to *Account*.

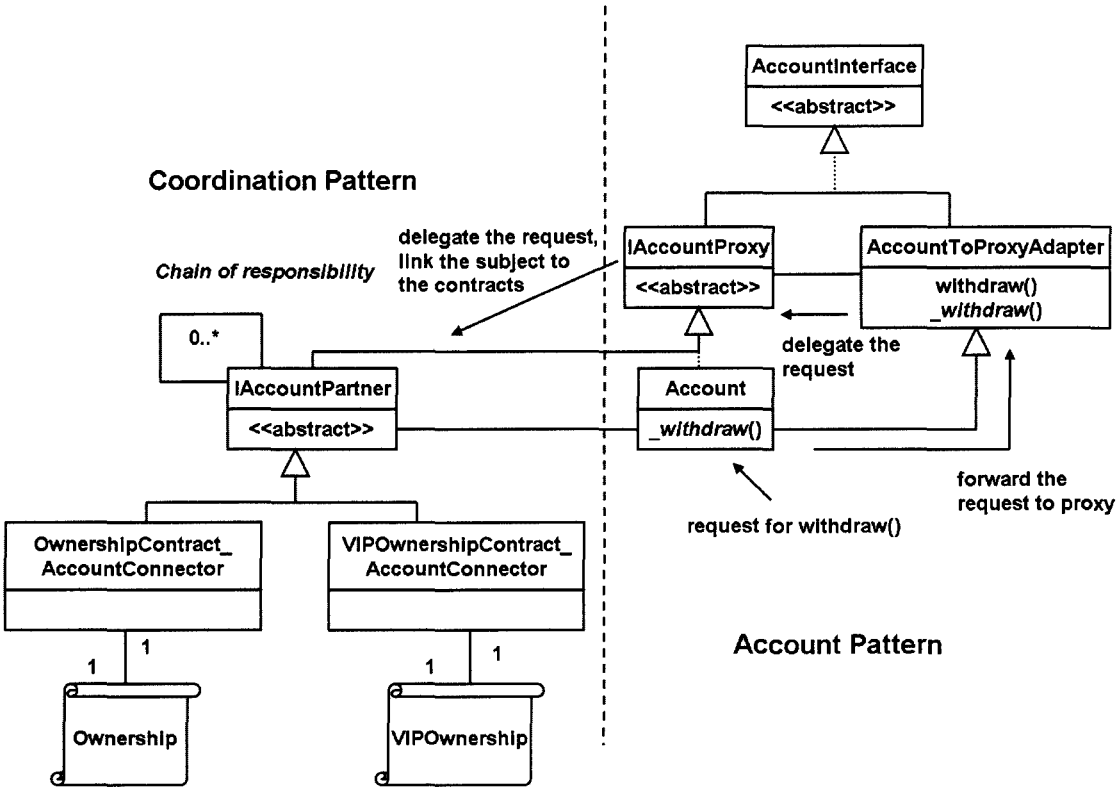


Figure 4.8: A Design Pattern for Account and Contracts in Figure 4.6

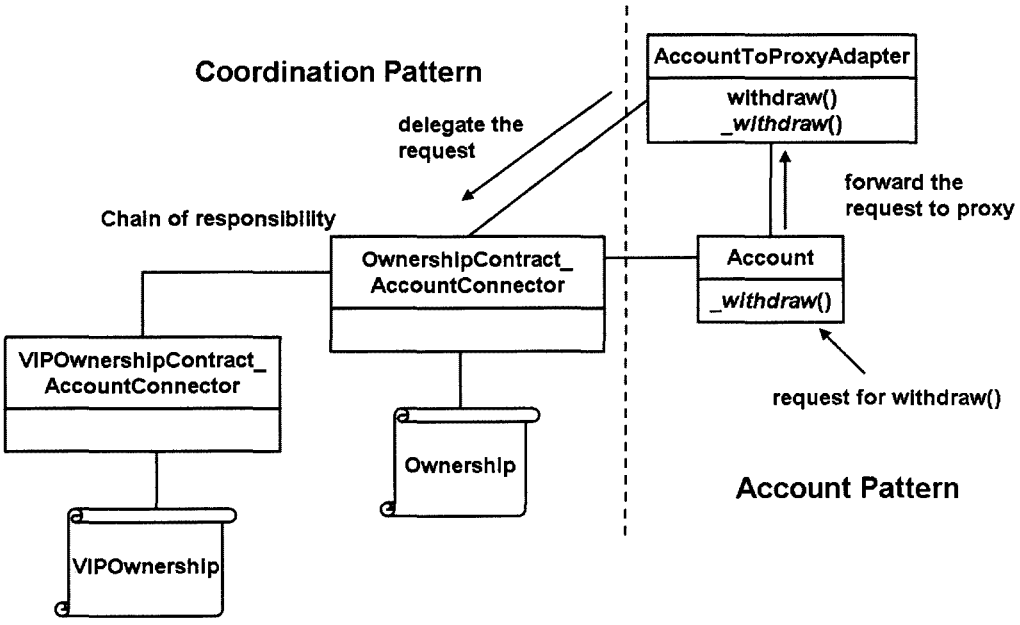


Figure 4.9: Design Pattern for Account without Contracts

The micro-architecture gives a blueprint for OOPLs to achieve coordination contracts. In what follows, we will present a development tool built using this micro-architecture.

4.6 Coordination Development Tool

Coordination contracts come along with a supporting toolset — CDE [69], which has been developed by ATX Software [6] with JDK 1.2 (as of CDE v1.1.1) under Windows operating systems. The tool enables us to specify, analyze, evolve systems and provides us the following functionalities:

- *Registration*: before being declared as contract participants, components under coordination should already exist as Java classes.
- *Editing*: defines a contract for components using the CDE specific language illustrated in Section 4.4.2 (page 53).
- *Deployment*: with the micro-architecture introduced in Section 4.5 (page 59), CDE automatically generates a Java implementation for a coordination contract from contract specifications.
- *Animation*: assists in testing/prototyping of contracts for both the development environment and runtime management purposes. The animator can dynamically perform operations such as create, destroy and execute objects and contract instances, monitor objects' states and operations in a sequence diagram, hence simulating applications. A *sequence diagram* [29, page 95] is an interaction diagram that emphasizes the time ordering of messages between objects involved in the interaction. A sequence diagram shows a set of roles and the messages sent and received by the instances playing the roles, which illustrates the behavior of the system.

The project source files are grouped in `src` where Java classes under coordination are located in the `components` subdirectory. The Java files generated by CDE to implement the micro-architecture are in the `generation` directory. The compiled class bytecode files are in `classes`.

With CDE, Java classes can be automatically generated to implement contracts and to adapt component classes in order to work with contracts. At run time, the system can be reconfigured by activating and deactivating contracts which enables run-time evolution. The CDE compiler will not check semantics of contracts, but checks if a contract instance conforms to the CDE language with respect to Java syntax.

In Section 4.4.1 (page 52) we have shown how UML extends the notion of coordination contract as a special kind of association class with enriched its semantics. Thus, any development of such an application would be based on the MDA (Model-Driven Architecture) approach, where the PIM (Platform-Independent Model) contains business rules declared in terms of contract specifications, PIM to PSM (Platform-Specific Model) transformation is accomplished by using Java language with the support for CDE.

4.7 Applications of Coordination Contracts

Coordination contracts have enlightened many related research areas. Silva et al.'s approach extends the concept of coordination contracts with a fault-tolerant scheme, which integrates C2 architectural style (Section 2.2.2, page 19) and the coordination layer (Section 4.3, page 51) where connectors are represented as contracts [38]. Bruel [30] puts forward a service-oriented implementation of component associations where the layered design regarding coordination contracts is purposed. Gahide et al. [57] promote their early work on component reuse by merging AOP with CBSD and the coordination contract approach in an architecture model, where coordination is modeled as a *transverse functional aspect.* Moreover, coordination contracts have demonstrated their surprisingly expressive power in software applications for many industry sectors, such as telecommunications [82], financial services [16, 83], transports [109], energy supplies [6], web-services technology [15], service-oriented development [18] etc.

4.8 Summary

First of all, we have compared coordination contracts with some techniques that may support modeling of architecture based software evolution, such as Object-Oriented Design, design patterns, AOP and the association class in UML. We characterize coordination contracts as synchronization contracts according to a category of contract abstraction levels. A three-layer architecture applied on coordination contracts is proposed with the computation layer, coordination layer and configuration layer. We introduce graphical and textual notations of contracts, where a scroll notation is created to extend UML, an abstract language and a CDE specific language to implement contracts. To develop contracts based applications, a development tool built using a micro-architecture is used to specify, analyze, evolve systems and provides us the functionalities like registration, editing, deployment and animator. In the end, several applications of coordination contracts are presented.

Chapter 5

Our Approach to Architecture-Based Evolution

5.1 The Multi-Dimensional Evolution Approach

Section 4.3 describes support to facilitate software evolution using a three-layer architecture, which includes a computation layer, a coordination layer and a configuration layer. In addition, the approach supports the divide-and-conquer law in Software Engineering by dealing with concerns separately. As a result, we view an architecture using a multiple dimensional perspective[1] in order to reduce the complexity of evolution, where "dimension" is formalized into these three supporting layers thanks to coordination contracts.

Although ideally, the principles of software evolution should be independent from implementation details at the architectural level, software development strategies and paradigms of programming languages[2] [56] have a substantial impact on the target software system under evolution. In this thesis, we are particularly interested in the evolution of Object-Oriented software systems.

For the purpose of focusing on connectors in this thesis, components and connectors may exist at different levels of abstraction. Stating this more clearly, components in the computational dimension are OO classes (in UML or Java), and connectors, reflecting business rules in the coordination dimension, are defined at a relatively higher level of abstraction, in contrast to components. Subsequently, we will take advantage of the mechanism of multiple dimensions, and in such a context we are able to discipline our approach to software evolution at the architectural level.

[1] Mikkonen [107] proposed a two-dimension architecture with components and their connections similar in spirit to our treatment.

[2] Paradigm refers to a category of entities that share a common characteristic.

5.1.1 The Component Dimension

The thesis's *main* concern is the fundamental support for the evolution of connectors in terms of coordination contracts. Evolution of components is not the issue of most concern, but as an indispensable part of a system, we will cover it briefly in this section. In the architectures that we are working on, an architectural component is similar to an OO class (probably a class in UML or Java). The permissible evolution operations on components are adding, modifying, and removing classes or their instances.

The components (specifications or instances) in the system that need to be evolved may change in some way as part of contracts, which is to say, the evolved system would result in introducing new components (adding components) to the old system that make the new contracts work. For example, a monitor component (class) may be included in a contract to assist in logging activities of classes being coordinated [5]. However, we will not give support in this thesis for how these components might evolve. The assumption is made so that this thesis is able to exclusively focus on how evolution of connectors will be managed.

The approach that we have discussed in Section 2.2 (page 18) seems very constructive for directing the evolution of components using type theory, when studying the evolution of detailed architectures[3]. In addition, we are working on the architecture of programs with the intention to bridge the gap between specification and implementation of architectural connectors. The state-of-art of coordination contracts supports Java implementations, so that subtyping in Java with pre- and postcondition specifications for Java methods is sufficient.

5.1.2 The Coordination Dimension

Considering the requirement for adapting changes in Section 1.1.2 (page 5) and various modeling techniques in Chapter 2, we decided to use coordination contracts in place of architectural connectors in Chapter 3, and the features of coordination contracts have been presented in Chapter 4. Coordination contracts conduct affairs and interactions between a chosen set of partners, defining the interactions superposed transparently on partners' behaviors [26]. In Section 1.3 (page 9), we have reasoned that our approach to evolution is still at specification time prior to executing code, since the architecture is still under development, at least for the contract/connector dimension in the coordination layer, even if the rest of the system may be partially implemented.

[3]We have clarified the evolution time and the level of abstraction that we are studying in Section 1.3, page 9.

Just as we are able to evolve components via certain operations, the evolution of connectors depends on well-defined operations like adding, removing or substituting connectors.

We conclude that our approach to evolution is based on reconfiguration of architectures, which is achieved through the addition, deletion, substitution of components and/or coordination contracts.

5.2 Predictable Evolution

5.2.1 Permissible Changes

It is difficult to predict which behavior will emerge when evolving a system in an arbitrary way; probably, "unexpected" or "undesired" behaviors would mess up the system after composition or deletion. We thus desire a simple and safe approach to evolving architectures. Contracts as connectors are instant plug-n-play modules while preserving the system behaviors in a preferred degree during evolution (see Section 5.5, page 78). For this reason, special care is needed to rule out unexpected results by regulating the kinds of possible extensions of the original system.

We have been examining contracts isolated in an individual system in Chapter 4 and the first few sections in this chapter so far. When considering whether an evolved system (SYS' in Figure 5.1) has predictable properties after performing permissible evolution operations on contracts, we have to inspect the behaviors of the original system (SYS in Figure 5.1) and even the change histories.

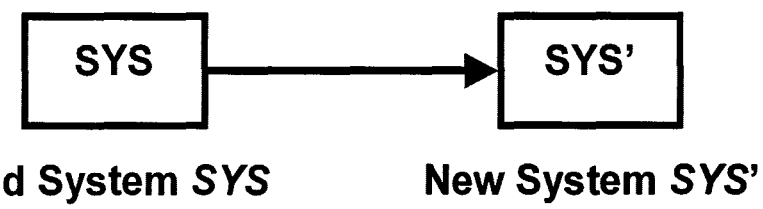


Figure 5.1: An Old System SYS and an Evolved System SYS'

For a coordination rule defined in a contract[4], the atomic execution process of which can be represented as a transition system, where nodes are states, arrows are transitions. Inside the frame in Figure 5.2, there is a transition system composed of 4 states and 3 transitions as a sequence of actions that occurs

[4]For readers who are not familiar with the specification language of contracts, referring to Section 4.4.2 (page 53) is highly encouraged.

in executing the contract. By "atomic", we mean if any state or transition cannot be reached as prescribed, execution of the entire transition system will be aborted. As far as the contracts are concerned, transition systems are relevant *only* when we compute the relation between inputs and outputs. Apparently, we intend to describe *static behaviors* [129] of contracts (or called input-output behaviors in the context), which means behavioral properties of a system at specific "snapshots", especially before and after the system's execution.

Admittedly, *static behaviors* are not expressive enough to represent how the internal states are reached. Accordingly, *dynamic behaviors* [129] of contracts complement *static behaviors* with a more detailed view of how the computation proceeds in the internal states via transitions during execution. Desirable state-based specification techniques that may be used to model *dynamic behaviors* are FSM (Finite State Machine), temporal logic [7], etc.

However, at this stage, we are not worried about the intermediate states and the associated transitions yet. In Section 4.4 (page 52), we claim that the language for coordination contracts is *declarative* because declarative specifications generally do not give details of intermediate steps, whereas *operational* specifications describe a series of steps that a functionality performs.

Figure 5.2 shows the observable behavior as a direct transformation from the input states (i.e., the states in which the rule is invoked) to the output states (i.e., the states resulting from the execution of the rule), viz., the dashed line T_1' from S_1 to S_4. We are interested in the input and the overall effect of the contract that are being preserved in evolution. In Section 5.4 (page 74) we will demonstrate that pre- and postconditions can be used to characterize the behavior of contracts.

Consequently, our approach is to compare the observable input-output functional behavior of an old contract to that of a new contract, and authorize changes only if the behaviors are "predictable". To achieve this, we will base our work on two fundamental techniques, which are *pre- and postconditions* formally capturing observable behaviors and comparing those conditions via *logical proof.*

Figure 5.3 illustrates the arrow appearing in Figure 5.1 by showing the possible evolution operations performed on contracts. For example, a contract C_1 in the old system SYS has been removed in the new system SYS'. C_2 is added in SYS'. Though its name is kept, the functionality of C_3 has been changed. C_4 stays unchanged both syntactically and semantically.

Generally speaking, we anticipate an incremental and predictive evolution so that we do not allow changes in a subtractive way, for example, removing a part of or the entire contract, such as C_1 in Figure 5.3, as this is less predictable and error-prone. However, we will include one possibility, namely that

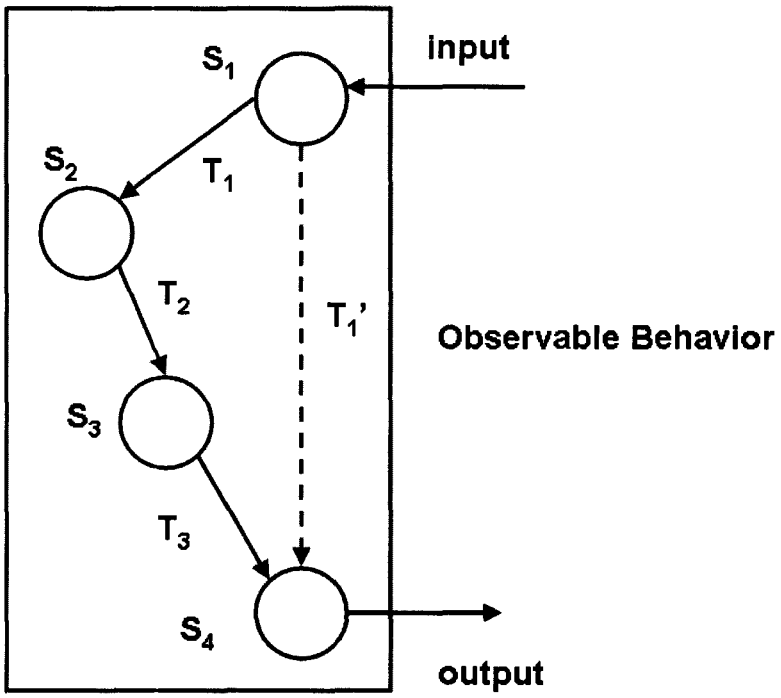


Figure 5.2: Executing a Coordination Contract as a Transition System

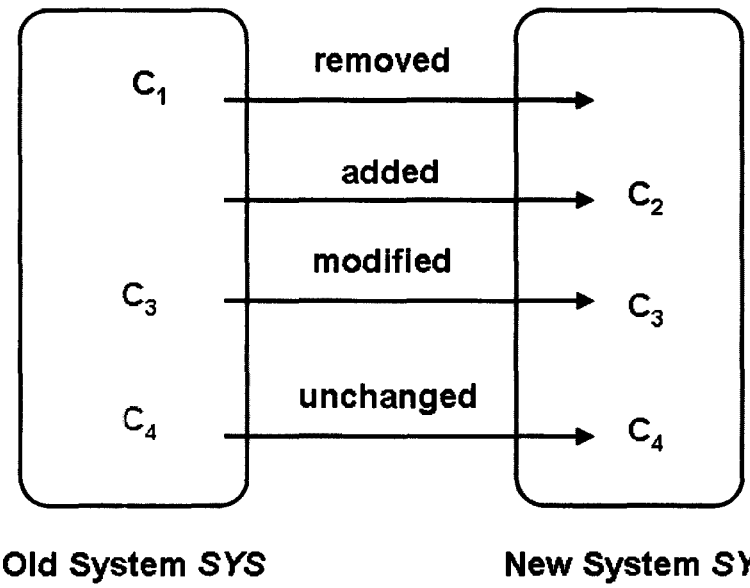


Figure 5.3: Possible Changes in an Evolved System against the Old System

one or more of the participants required by the contract is absent from the system for some reason. Once having detected the absence of participant(s),

removing contract(s) involving them seems reasonable, though it might not be predictable for the whole system.

5.2.2 Change Histories

Are we able to compare two arbitrary systems in terms of evolution? Certainly, we do not desire to compare any two systems. It is also a rare case that we evolve a system in the absence of prior knowledge of it. So first of all, we will characterize change histories in a way that enables control if system evolution in a predictable direction. In addition, how different are these versions allowed to be in terms of contracts? In the rest of this section, we will answer the first question and leave the second one to the following sections.

Basically, we will start to compare two consecutive evolving systems. For example, we have 3 systems including the original system, SYS, and two evolving systems, SYS_1 and SYS_2. If SYS has no contracts yet, we assume contracts to pass on exactly the original method call, while preserving the behaviors of the original method calls without introducing any new features. How are these systems related to each other?

Case 1: SYS, SYS_1 and SYS_2 in Figure 5.4 are systems evolving "sequentially" from each other. If the evolution processes E_1 and E_2 are predictable from SYS and SYS_1, respectively, we are able to combine these two cumulative changes into Figure 5.5 and make an evolution E_1' from SYS to SYS_2 directly.

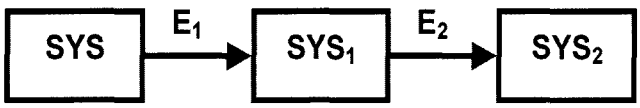


Figure 5.4: System Evolution – Case 1

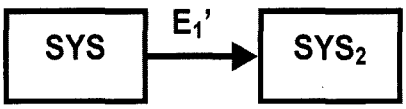


Figure 5.5: System Evolution – Case 1′

Case 2: SYS_1 and SYS_2 in Figure 5.6 are evolving "individually" based on SYS. Suppose the evolution processes E_1 and E_2 are predictable from SYS; we may not be able to fully justify the relationship between SYS_1 and SYS_2, though they are each predictable from SYS, respectively. Probably, there is no direct or obvious relationship between SYS_1 and SYS_2.

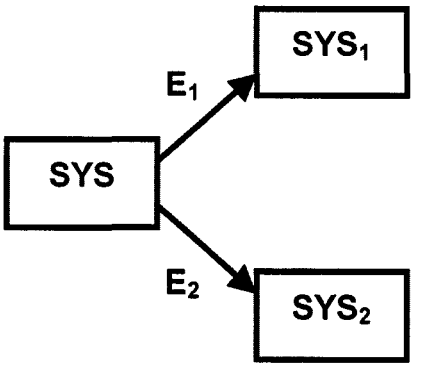


Figure 5.6: System Evolution – Case 2

Case 3: SYS_1 and SYS_2 are systems evolved from SYS individually, SYS_3 can be achieved either from evolution on SYS_1 or SYS_2 in Figure 5.7. If the evolution processes E_1, E_2, E_{13}, E_{23} are predictable from SYS, SYS_1 and SYS_2, respectively, we say SYS_3 is *jointly predictable* by using both paths from E_1, E_{13} and E_2, E_{23}. By imitating the example in Figure 5.5, we combine the changes E_1 and E_{13} into E_3, E_2 and E_{23} into E_3' in Figure 5.8.

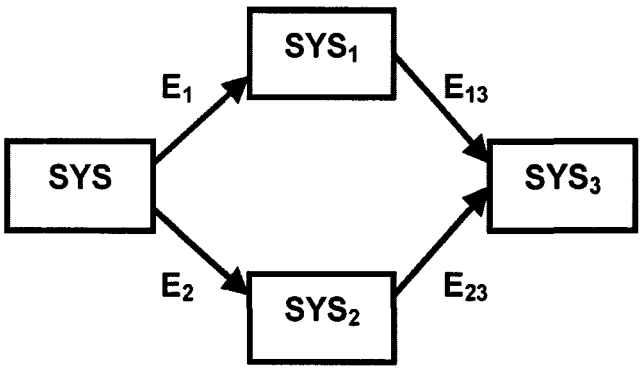


Figure 5.7: System Evolution – Case 3

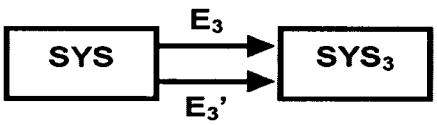


Figure 5.8: System Evolution – Case 3′

Until now, we have shown three different scenarios of change histories that systems may have. The second question, "how different these versions are allowed to be in terms of contracts", is abstractly illustrated as arrows E_1, E_2, etc., in above figures. To illuminate it, some form of specification is required to

capture the specific evolution constraints between contracts in the two different versions, demanding the preservation of specific properties, such as behaviors.

5.3 Inspirations from Related Work

5.3.1 Subtyping

We have introduced the concept of subtype in Section 2.2.1 (page 18) when discussing the approach to modeling component evolution. The subtyping relationship is represented as $S <: T$ ($<:$ is used to mean "is a subtype of").

Object-Oriented programming languages like Java have set up a universe of types; all types are related in some way in a type hierarchy. Inferior types in the hierarchy are ideally compatible with more general superior types in the hierarchy. Types are generally used to give information on the syntax of methods or components. Type checkers are created to guarantee no type errors occur when subtype objects replace supertype objects in a piece of executable program. The role of coordination contracts is similar to types, in the sense that they are able to specify properties and behaviors.

5.3.2 Pre- and Postconditions

Pre- and postconditions, proposed by Floyd [55] and further refined by Hoare [75] and Dijkstra [41], have contributed much to the art of program development. The Hoare triple [75], named after C.A.R. Hoare, is denoted as $\{P\}S\{Q\}$, where P is a precondition, S a program and Q a postcondition, P and Q are assertions. Note that the assertions are Boolean expressions in FOL other than the assertions appearing in [23] when making a contract between two agents working on the same state independently of each other. Precondition P specifies the initial values of variables in the state space of the program before the execution of S, and the postcondition Q specifies the final values and/or their relations with the initial values. Consequently, $\{P\}S\{Q\}$ may be read as "if the assertion P is true before execution of S, S will terminate in a state where the assertion Q is true".[5] Duan [43] has compared these notions of pre- and postconditions and some relevant work.

A formal specification is "the expression in some formal language and at some level of abstraction, of a collection of properties some system should satisfy "[145]. Specification technique based on pre- and postconditions has been

[5]Total correctness is assumed here, by which we mean a particular execution of a contract must terminate, given an event or a condition. But not all contracts should be terminating.

proven to offer a useful modeling paradigm to documenting contracts up to the behavioral level[6]. We will show later that coordination contracts, which belong to synchronization contracts, are unexceptional in this regard, so that pre- and postconditions can be effectively used to represent the functionality of coordination contracts. Unfortunately, the formal semantics of coordination contracts and their relation with program logic cannot be easily inferred without involving the soundness and the internal structure of contracts, and consequently is beyond the scope of this thesis.

5.3.3 Behavioral Subtyping

A behavioral subtyping specification includes both syntactic and semantic aspects in contrast to that of a subtyping relation. Behavioral subtyping is used to guarantee that no surprising or unexpected behavior occurs when subtype objects replace supertype objects.

The *Liskov Substitution Principle* (LSP) [91, 92] is presented as one of the cornerstones for reasoning about behavioral subtyping and substitutability. In LSP, objects of a subtype can only match or if they:

- weaken the preconditions of the supertype, not strengthen them (as contravariance).
- strengthen the postconditions of the supertype, not weaken them (as covariance).
- strengthen the invariants of the supertype, not weaken them.

The LSP is surprisingly similar in spirit to the *Assertion Redeclaration Rule* in Design by Contract [106, page 573], which defines "a routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger". Consequently, we will try to deal with relationships between coordination contracts by means of pre- and postcondition specification.

[6]Different types of contracts are discussed in Section 4.2, page 49.

5.4 Specification Level Representation of Coordination Contracts

5.4.1 Contract Specification Revisited

In Section 4.4.2 (page 53), we have introduced two specification languages for coordination contracts at distinct abstraction levels. The principal differences are explicit constants and invariants in the abstract specifications, and Java specific syntax in CDE specifications. Considering the importance of constants and invariants for specifications, we will include them in order to reason about the relations between contracts. The language we adopt is a mixture of abstract and CDE specifications.

A contract may have two kinds of rules. The trigger rule may be invoked by method calls, while the state condition rule may be invoked by a condition on participant objects. The guard condition imposes additional limits on the trigger. The actions describe the behavior defined by the rule in terms of extra behaviors to be executed before or after action (in the `before` or `after` block), or behaviors to replace the original method call (in the `do` block). If the guard is not true, the `failure` clause will throw an exception or return a value.

Table 5.1: Contract Specification Revisited

```
coordination //rules
  TriggerRuleName:
    when *- >> participant1.operation(arguments)
          && (trigger conditions in Java)
    with (JavaGuardConditions)
    failure (Java guard failure
          actions throw an exception or return a value)
    before { before actions in Java };
    do { replace actions in Java };
    after {after actions in Java};
  StateConditionRuleName:
    when ? (condition in Java) on participant1, participant2...
    do {operations in Java};
```

Pre- and postconditions of coordination contracts are used to specify the observable behaviors (see also Figure 5.2, page 69) in terms of the input trigger and/or guard condition and the effect of output. Our approach is to compare the observed behaviors of an old contract to the observed behaviors of a new contract, and to permit the changes only if the behavior ensures predictability.

Contracts are specified by individual elements composed in the contract

specification language (Section 4.4.2, page 53). Pre- and postcondition of contracts in the old and new system may or may not include changes of each element. For example, though probably not being included in pre- and postconditions, intuitively, syntactic alterations such as renaming contracts or rules are claimable by matching their signatures[7]. The new contract may have the same set or more constants, attributes and operations than the old contract. According to Section 5.1.1 (page 66), participant components may be introduced in a new contract. Every contract may have an arbitrary number of invariants. Following the treatment of LSP in Section 5.3.3 (page 73), invariants private to the new contract maintain or strengthen invariants in the old contract.

From the specification language and the discussion above, an obvious fact is that the pre- and postconditions are not part of the contract working code. To reason about the prediction of software evolution, we will show in the following subsections how pre- and postconditions of methods, pre- and postconditions of coordination rules, pre- and postconditions of coordination contracts are defined.

5.4.2 Pre- and Postconditions of the Method Being Called

The pre- and postconditions of the individual method being called are defined as the pre- and postcondition specification of the corresponding method subjected to coordination in the UML or Java class. We have defined such pre- and postconditions in Section 5.3.2 (page 72).

5.4.3 Preconditions of Coordination Rules

Considering the two kinds of rules supported by coordination contracts, we will define the precondition for each case respectively. The precondition of a coordination rule must be satisfied, otherwise the rule will not be invoked.

- If a trigger is a request for a method call, the precondition of a rule in a contract is the precondition of the original method being called and additional conditions in the `when` clause, and extra conditions in the `with` clause, i.e., $pre_{op} \wedge trigger_{call} \wedge with_{call}$.
- If a trigger is a request for a method call, and what a contract does when executing a rule is pass on the original method call without imposing extra behaviors, then it is evident that the effect of `when` and `with` is

[7]Signature matching is roughly defined in Section 5.5, page 78.

evaluated to be `true`, $pre_{op} \wedge trigger_{call} \wedge with_{call} \equiv pre_{op} \wedge true \wedge true \equiv pre_{op}$. The precondition of the coordination rule is thus the precondition of the original method.

- If a trigger is a state condition on participant components, the precondition of a rule in a contract is exactly the condition in this rule, which is composed of conditions in the `when` clause, i.e., $trigger_{state}$.

5.4.4 Postconditions of Coordination Rules

Following the methodology of the above subsection, we will define the postcondition of rules for each case separately. Any violation of postcondition established by coordination rules will undo the entire execution.

- If a trigger is a request for a method call, the postcondition of the rule in a contract is a *joint effect* of the rule in a contract ($before\text{-}do\text{-}after_{call}$). We define "joint effect" as the effect of `do` (if there is any) and the effect of `before` (if there is any) and the effect of `after` (if there is any). In a word, the effect of `before-do-after` block is an accumulated effect of these sequential activities. If `before`, `do` and `after` are *empty*, the postcondition of the rule is $post_{op}$, shown as follows.

- If a trigger is a request for a method call, and what a contract does when executing a rule is pass on the original method call without imposing extra behaviors, then it is evident that the effect of `before-do-after` is evaluated to be that of the original method, so that the postcondition of the coordination rule is the postcondition of the original method ($post_{op}$).

- If a trigger is a request for a method call, and the guard for a rule is not satisfied, the rule will not be executed regardless of which condition it is triggered on, so that the postcondition of the rule will be the effect of exceptions in the `failure` clause ($failure_{call}$).

- If a trigger is a query on state conditions of participant components, the postcondition of the rule in a contract is the effect of `do` block (do_{state}). Since there is no `with` clause guarding state condition rules, so that no `failure` is defined in the rule body.

5.4.5 Pre- and postcondition of Coordination Contracts

Therefore, the pre- and postcondition of contracts is a combination of the pre- and postcondition of each individual rule.

- If a trigger is a request for a method call,
  - The precondition of a contract may include two parts. For each rule, the fixed part is the corresponding precondition of the original method (pre_{op}). The changing part consists of additional conditions in the `when` clause ($trigger_{call}$), and extra conditions in the `with` clause ($with_{call}$).
  - The postcondition of a contract also has three exclusive possibilities. One is the corresponding postcondition of the original method ($post_{op}$) if a call to which is eventually made. The other is the joint effect in `before-do-after` block ($before\text{-}do\text{-}after_{call}$) if the block is executed in place of the original call. Another is the effect of exceptions declared in `failure` clause ($failure_{call}$) if the guard fails.
- If a trigger is a state condition on participant components,
  - The precondition of a contract is subjected to change, which is the condition in this rule ($trigger_{state}$).
  - The postcondition of a contract is the effect in `do` block (do_{state}).

In a contract, we may define different pre- and postconditions for the same trigger because of different `when` and/or `with` conditions, as well as different `do` blocks.

The pre- and postcondition relation of a contract may be coarsely represented as $((pre_{rule1} \Rightarrow post_{rule1}) \wedge (pre_{rule2} \Rightarrow post_{rule2}) \wedge ... \wedge (pre_{rulen} \Rightarrow post_{rulen}))$, where pre_{rulei} can be either $(pre_{op_i} \wedge trigger_{calli} \wedge with_{calli})$ or $trigger_{statei}$ depending on the type of trigger, and $post_{rulei}$ is one of $post_{op_i}$, $before\text{-}do\text{-}after_{calli}$, $failure_{calli}$ if the trigger is a method call, or do_{statei} if the trigger is a query on states. If a rule has both $post_{op}$ and $failure_{call}$, or $before\text{-}do\text{-}after_{call}$ and $failure_{call}$, the effect of the rule will be considered separately.

If a contract just passes on a particular method call without extra behaviors, the pre- and postcondition of the contract and that of the original invoked method are the same. Or if two contracts are logically equivalent (Section 5.5.2). In these cases, it turns out that the new contract is as good as the old contract.

Multiple rules may be included in a contract, regardless of being invoked by the same trigger (on method call or state) or not. In this case, when replacing a contract with a new one, firstly we will examine if the same rule(s) exists (signatures matched); the new contract may have additional rules. Then for each rule these two contracts have in common, efforts in inspecting the relation between their pre- and postconditions are required.

We have made a commitment in Section 5.2 to solving a more demanding situation, how different these versions are allowed to be in terms of contracts. In the rest of this Chapter, we will discuss this issue from a perspective of specification matching. By checking the logical relations between their pre- and postcondition specifications, we will be able to identify a diversity of relevant connections between contracts.

5.5 Behavioral Relationships between Coordination Contracts

Specification matching has been used to evaluate component relations and to retrieve software components for reuse at an abstract level by means of pre- and postcondition analysis in [149]. We take advantage of specification matching as a method for justifying the behavioral relationships between two coordination contracts. A specification match is a binary Boolean functional relation defined as $match : Spec \times Spec \rightarrow \{true, false\}$. For a match ($match$), two specification of contracts S_1 and S_2, if $match(S_1, S_2) = true$, we say S_1 matches S_2 according to $match$. It is worthy to notice that most matches are *not* symmetric: $match(S_1, S_2)$ does not necessarily imply $match(S_2, S_1)$. Since the name of a contract is its unique identity in the coordination layer, we use the name of a contract to stand for its specification, i.e., C_1 and C_2 in $match(C_1, C_2)$ are used to denote specifications for C_1 and C_2, respectively.

Before talking about specification matching, we assume that the *signatures* of two contracts match, which means the list of types of each rule's input and output parameters and the exceptions that may be raised match. These parameters may consist of parameters used in the original method call, in the rule, or attributes and constants used in the contract. To perform the signature matching, techniques such as currying, type coercion, signature reordering, etc., may be involved. However, to realize the signature matching of coordination contracts is out of the thesis's scope.

In this thesis, we will stick to the convention that the logical operator $\Rightarrow$ means implication, $\Leftrightarrow$ means equivalence and $\Leftarrow$ means reverse implication. An assertion A is said to be stronger than another assertion B, if A logically implies B, i.e. $A \Rightarrow B$. If A is stronger than B, then B is said to be weaker than A.

We treat specifications from a *relational* view, i.e., a specification is considered as a pair of pre- and postcondition, $\langle C_{pre}, C_{post} \rangle$. As illustrated in Figure 5.9, OC_{pre} is the precondition of the old contract (OC) and OC_{post} the postcondition, and likewise for NC_{pre} and NC_{post}, which are the pre-

and postcondition of the new contract (NC). If OC_{pre} holds before the contract performs, then OC_{post} holds after successful execution (total correctness assumed), the same for NC_{pre} and NC_{post}. According to the definition of behavioral subtyping rule (LSP in Section 5.3.3, page 73), a new contract whose specification guarantees $NC_{pre} \Rightarrow NC_{post}$ may replace an old contract whose specification guaranteed $OC_{pre} \Rightarrow OC_{post}$ (dashed arrow in Figure 5.9), if the new contract has a weaker precondition and a stronger postcondition as opposed to that of the old contract, i.e., $OC_{pre} \Rightarrow NC_{pre}$ and $NC_{post} \Rightarrow OC_{post}$ (solid arrows in Figure 5.9). For example, for the same method being called, the precondition of the old contract OC_{pre} is $j = i + 1$ $(i, j \in \mathbb{N})$ and the precondition of the new contract NC_{pre} is $i \neq j$, then $j = i + 1 \Rightarrow i \neq j$. We conclude that the new contract has a weaker precondition, $OC_{pre} \Rightarrow NC_{pre}$. Arrows in Figure 5.9 denote program control flow or logical implication in specifications, the different meanings are shown as the figure legend.

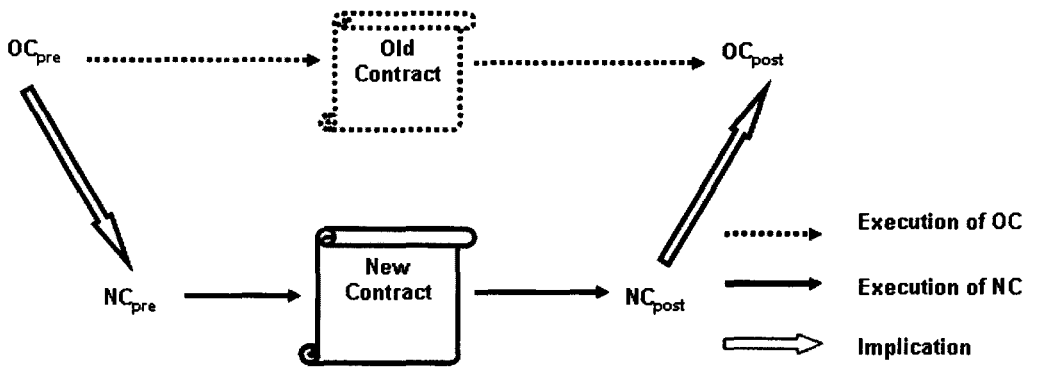


Figure 5.9: Contract Behavioral Relationships

To calculate their relations, we will use expressions in First Order Logic (FOL) or Object Constraint Language (OCL) [113], which is integrated in the UML standard, to annotate pre- and postconditions of contracts. Checking pre- and postconditions and their implication can be done in several ways: by a hand-proof, a theorem prover or even at runtime. However, we have to state that the type checker for Java is not able to handle the proving, since the syntax and the semantics of language for contract is not Java yet and working at the specification time. To automate the calculation, a specification match maker is needed.

5.5.1 Outline of Behavioral Specification Matching

Determining the behavioral relationship between coordination contracts is a principal task for this thesis to compare different contracts during evolution. We establish a semantic foundation for connecting and reasoning about specification matches in Figure 5.10[8], which provides a framework to assess different cases of contract specification match. We will show in the next Chapter by multiple case studies on how a set of these matches will help establish predictable software evolutions practically.

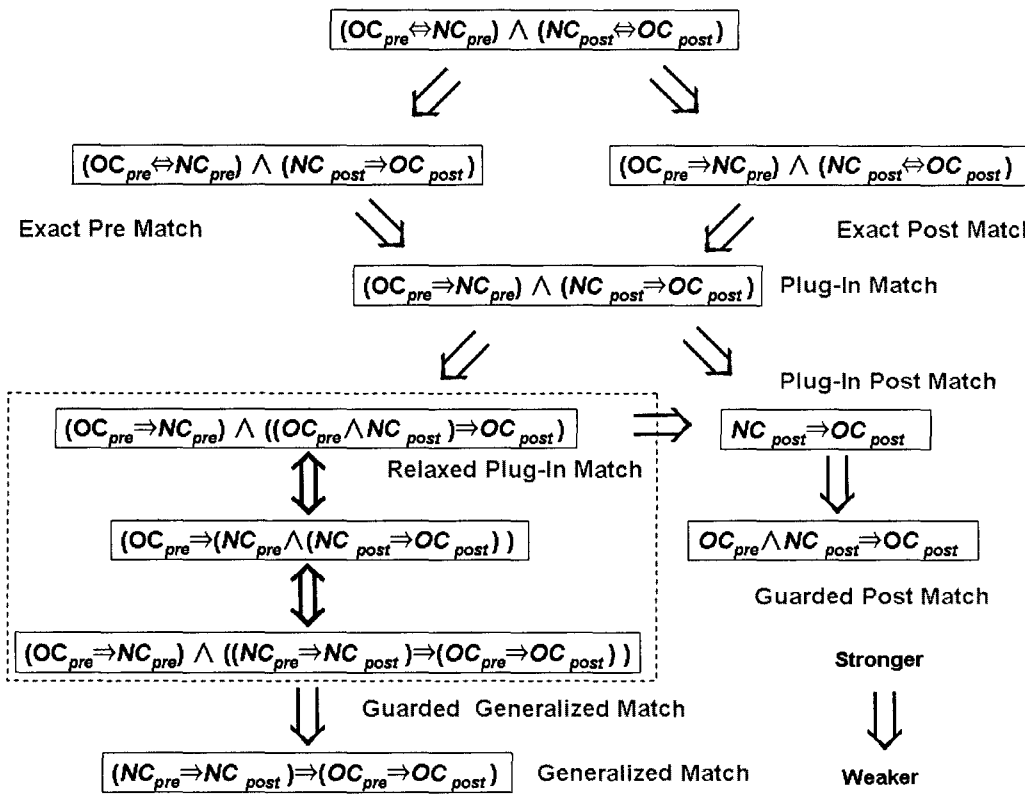


Figure 5.10: Contract Specification Matches

An arrow in Figure 5.10 between two matches indicates that the match at the base of the arrow is stronger than the match at its end, i.e., $match_1$ is stronger than $match_2$ if $match_1(C_1, C_2) \Rightarrow match_2(C_1, C_2)$. On the other hand, the match at its end $match_2(C_1, C_2)$ is more relaxed than the match at the base of the arrow $match_1(C_1, C_2)$. The formal notation is abbreviated by dropping the universal quantifications and respective parameters for the predicates. For example, $OC_{pre} \Rightarrow NC_{pre}$ is equivalent to $\forall p.$ (p satisfies OC_{pre}) $\Rightarrow$ (p satisfies NC_{pre}), given p is an assertion.

[8]Figure 5.10 is adapted by referring to [32, 144, 149].

In what follows, we will testify and adapt the matches for components in [149] to be applied for contracts and establish 5 major specification matches[9]. Generally, all these matching variants are derived from two definitions: one relates the preconditions and postconditions of two contracts separately, whereas the other relates the specification predicates of two contracts together [132]. Sections 5.5.2 through 5.5.6 will demonstrate proof sketches and properties for matches, where the fundamental proof techniques can be found in [127] or [71]. As later demonstrated, the proving tasks are straightforward and based on the properties of predicate logic.

5.5.2 Exact Pre/Post Match

Exact Pre/Post Match is proposed by Zaremski and Wing [149], denoted as $match_{exact}(NC, OC) = (OC_{pre} \Leftrightarrow NC_{pre}) \wedge (NC_{post} \Leftrightarrow OC_{post})$. In Figure 5.10, ExactPre/Post Match appears to be the strongest match. Two contracts exactly match if and only if their preconditions and postconditions are logically equivalent. Hence, the two contracts are interchangeable. However, the equivalence relationship is a strict and rare case in evolution when only signatures of contracts are modified, for example, renaming rules or parameters. Subsequently, we will discuss a variety of cases where the new contract may safely match the old contract in the system by relaxing the Exact Pre/Post Match, such as Plug-in Match, Relaxed Plug-in Match, Guarded Generalized Match and Generalized Match. For example, Plug-in Match relaxes the $\Leftrightarrow$ arrows in Exact Pre/Post Match to $\Rightarrow$ arrows.

A match relation (*match*) is an *equivalence match* if the following conditions are satisfied:

- Reflexive: $match(C, C)$ for all contract specifications C
- Transitive: If $match(C_1, C_2)$ and $match(C_2, C_3)$, then $match(C_1, C_3)$
- Symmetric: If $match(C_1, C_2)$, then $match(C_2, C_1)$

We claim that $match_{exact}$ is reflexive, symmetric and transitive, so that it is an equivalent match.

- Reflexive:

[9]We have been taking efforts to strike a balance between flexibility and predictability. However, sometimes it requires to sacrifice flexibility for predictability.

$$
\begin{array}{ll}
& match_{exact}(C, C) \\
= & \langle \text{Definition of } match_{exact} \rangle \\
& (C_{pre} \Leftrightarrow C_{pre}) \wedge (C_{post} \Leftrightarrow C_{post}) \\
\Longleftrightarrow & \langle \text{Reflexivity of } \Leftrightarrow\text{: } p \Leftrightarrow p \equiv true \rangle \\
& true \wedge true \\
\Longleftrightarrow & \langle \text{Reflexivity of } \wedge : \ p \wedge p \equiv p \rangle \\
& true \quad \blacksquare
\end{array}
$$

- Transitive:

  Suppose we have,

  (1) $match_{exact}(C_1, C_2) = (C_{2pre} \Leftrightarrow C_{1pre}) \wedge (C_{1post} \Leftrightarrow C_{2post})$, and

  (2) $match_{exact}(C_2, C_3) = (C_{3pre} \Leftrightarrow C_{2pre}) \wedge (C_{2post} \Leftrightarrow C_{3post})$, then by the definition of *transitivity*, we will show $match_{exact}(C_1, C_3) = (C_{3pre} \Leftrightarrow C_{1pre}) \wedge (C_{1post} \Leftrightarrow C_{3post})$.

$$
\begin{array}{ll}
\text{Given (1) and (2),} & \\
& ((C_{2pre} \Leftrightarrow C_{1pre}) \wedge (C_{1post} \Leftrightarrow C_{2post})) \\
& \wedge ((C_{3pre} \Leftrightarrow C_{2pre}) \wedge (C_{2post} \Leftrightarrow C_{3post})) \\
\Longleftrightarrow & \langle \text{Symmetry of } \wedge : \ p \wedge q \equiv q \wedge p \rangle \\
& ((C_{2pre} \Leftrightarrow C_{1pre}) \wedge (C_{3pre} \Leftrightarrow C_{2pre})) \\
& \wedge ((C_{1post} \Leftrightarrow C_{2post}) \wedge (C_{2post} \Leftrightarrow C_{3post})) \\
\Longleftrightarrow & \langle \text{Transitivity of } \Leftrightarrow\text{: } (p \Leftrightarrow q) \wedge (q \Leftrightarrow r) \Leftrightarrow (p \Leftrightarrow r) \rangle \\
& (C_{3pre} \Leftrightarrow C_{1pre}) \wedge (C_{1post} \Leftrightarrow C_{3post}) \\
= & \langle \text{Definition of } match_{exact} \rangle \\
& match_{exact}(C_1, C_3) \quad \blacksquare
\end{array}
$$

- Symmetric:

$$
\begin{array}{ll}
& match_{exact}(C_1, C_2) \\
= & \langle \text{Definition of } match_{exact} \rangle \\
& (C_{2pre} \Leftrightarrow C_{1pre}) \wedge (C_{1post} \Leftrightarrow C_{2post}) \\
\Longleftrightarrow & \langle \text{Symmetry of } \Leftrightarrow\text{: } p \Leftrightarrow q \equiv q \Leftrightarrow p \rangle \\
& (C_{1pre} \Leftrightarrow C_{2pre}) \wedge (C_{2post} \Leftrightarrow C_{1post}) \\
= & \langle \text{Definition of } match_{exact} \rangle \\
& match_{exact}(C_2, C_1) \quad \blacksquare
\end{array}
$$

5.5.3 Plug-in Match

Plug-in Match is proposed by Zaremski and Wing [149], which is a relaxed form of Exact Pre/Post Match, and defined as $match_{plug-in}(NC, OC) = (OC_{pre} \Rightarrow NC_{pre}) \wedge (NC_{post} \Rightarrow OC_{post})$. We have illustrated the case in Figure 5.9, where Plug-in Match succeeds when the precondition of the new contract is weaker, and its postcondition is stronger than that of the old contract, respectively. In other words, the new contract allows at least all the conditions that the old contract allows, and provides a guarantee at least as strong as the old contract provides.

Recall the concept of behavioral subtyping in Section 5.4 (page 74), the purpose of which is to enforce the preservation of behavioral properties. Following LSP, we are able to split the plug-in match into two conditions: the precondition rule ($OC_{pre} \Rightarrow NC_{pre}$) means a weakening of the precondition (contravariance) and the postcondition rule ($NC_{post} \Rightarrow OC_{post}$) means a strengthening of the postcondition (covariance).

The relation $match_{exact}(NC, OC) \Rightarrow match_{plug-in}(NC, OC)$ in Figure 5.10 is proved as follows:

Step 1:

$$
\begin{array}{ll}
& match_{exact}(NC, OC) \\
= & \langle \text{Definition of } match_{exact} \rangle \\
& (OC_{pre} \Leftrightarrow NC_{pre}) \wedge (NC_{post} \Leftrightarrow OC_{post}) \\
\Longrightarrow & \langle \text{Definition of } \Leftrightarrow\text{: } p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (p \Leftarrow q), \text{ and} \\
& \text{Implication - Weakening: } p \wedge q \Rightarrow p \rangle \\
& (OC_{pre} \Leftrightarrow NC_{pre}) \wedge (NC_{post} \Rightarrow OC_{post}) \ (3)
\end{array}
$$

Step 2:

$$
\begin{aligned}
& match_{exact}(NC, OC) \\
= \quad & \langle \text{Definition of } match_{exact} \rangle \\
& (OC_{pre} \Leftrightarrow NC_{pre}) \wedge (NC_{post} \Leftrightarrow OC_{post}) \\
\Longrightarrow \quad & \langle \text{Definition of } \Leftrightarrow\text{: } p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (p \Leftarrow q), \text{ and} \\
& \text{Implication - Weakening: } p \wedge q \Rightarrow p \rangle \\
& (OC_{pre} \Rightarrow NC_{pre}) \wedge (NC_{post} \Leftrightarrow OC_{post}) \quad (4)
\end{aligned}
$$

Step 3:

Given (3) and (4),

$$
\begin{aligned}
& ((OC_{pre} \Leftrightarrow NC_{pre}) \wedge (NC_{post} \Rightarrow OC_{post})) \\
& \wedge ((OC_{pre} \Rightarrow NC_{pre}) \wedge (NC_{post} \Leftrightarrow OC_{post})) \\
\Longleftrightarrow \quad & \langle \text{Definition of } \Leftrightarrow\text{: } p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (p \Leftarrow q) \rangle \\
& (((OC_{pre} \Rightarrow NC_{pre}) \wedge (OC_{pre} \Leftarrow NC_{pre})) \wedge (NC_{post} \Rightarrow OC_{post})) \\
& \wedge ((OC_{pre} \Rightarrow NC_{pre}) \wedge ((NC_{post} \Rightarrow OC_{post}) \wedge (NC_{post} \Leftarrow OC_{post}))) \\
\Longleftrightarrow \quad & \langle \text{Associativity and Idempotency of } \wedge : (p \wedge q) \wedge r \equiv p \wedge (q \wedge r), \\
& p \wedge p \equiv p \rangle \\
& (OC_{pre} \Rightarrow NC_{pre}) \wedge (NC_{post} \Rightarrow OC_{post}) \\
& \wedge (OC_{pre} \Leftarrow NC_{pre}) \wedge (NC_{post} \Leftarrow OC_{post}) \\
\Longrightarrow \quad & \langle \text{Implication - Weakening: } p \wedge q \Rightarrow p \rangle \\
& (OC_{pre} \Rightarrow NC_{pre}) \wedge (NC_{post} \Rightarrow OC_{post}) \\
= \quad & \langle \text{Definition of } match_{plug-in} \rangle \\
& match_{plug-in}(NC, OC) \quad \blacksquare
\end{aligned}
$$

A match relation (*match*) is a *partial order match* if following conditions are satisfied:

- Reflexive: $match(C, C)$ for all contract specifications C
- Transitive: If $match(C_1, C_2)$ and $match(C_2, C_3)$, then $match(C_1, C_3)$
- Antisymmetric: Given $match(C_1, C_2)$ and $match(C_2, C_1)$, a corresponding match can be inferred as an equivalence match

We claim that $match_{plug-in}$ is reflexive, transitive and antisymmetric, so that it is a partial order match.

- Reflexive:

$$
\begin{array}{ll}
& match_{plug-in}(C, C) \\
= & \langle \text{Definition of } match_{plug-in} \rangle \\
& (C_{pre} \Rightarrow C_{pre}) \wedge (C_{post} \Rightarrow C_{post}) \\
\Longleftrightarrow & \langle \text{Reflexivity of } \Rightarrow\text{: } p \Rightarrow p \equiv true \rangle \\
& true \wedge true \\
\Longleftrightarrow & \langle \text{Reflexivity of } \wedge : \ p \wedge p \equiv p \rangle \\
& true \quad \blacksquare
\end{array}
$$

- Transitive:

  Suppose we have,

  (5) $match_{plug-in}(C_1, C_2) = (C_{2pre} \Rightarrow C_{1pre}) \wedge (C_{1post} \Rightarrow C_{2post})$, and

  (6) $match_{plug-in}(C_2, C_3) = (C_{3pre} \Rightarrow C_{2pre}) \wedge (C_{2post} \Rightarrow C_{3post})$, then by the definition of *transitivity*, we will show $match_{plug-in}(C_1, C_3) = (C_{3pre} \Rightarrow C_{1pre}) \wedge (C_{1post} \Rightarrow C_{3post})$.

$$
\begin{array}{ll}
\text{Given (5) and (6),} & \\
& ((C_{2pre} \Rightarrow C_{1pre}) \wedge (C_{1post} \Rightarrow C_{2post})) \\
& \wedge((C_{3pre} \Rightarrow C_{2pre}) \wedge (C_{2post} \Rightarrow C_{3post})) \\
\Longleftrightarrow & \langle \text{Symmetry of } \wedge : \ p \wedge q \equiv q \wedge p \rangle \\
& ((C_{2pre} \Rightarrow C_{1pre}) \wedge (C_{3pre} \Rightarrow C_{2pre})) \\
& \wedge((C_{1post} \Rightarrow C_{2post}) \wedge (C_{2post} \Rightarrow C_{3post})) \\
\Longrightarrow & \langle \text{Transitivity: } (p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \rangle \\
& (C_{3pre} \Rightarrow C_{1pre}) \wedge (C_{1post} \Rightarrow C_{3post}) \\
= & \langle \text{Definition of } match_{plug-in} \rangle \\
& match_{plug-in}(C_1, C_3) \quad \blacksquare
\end{array}
$$

- Antisymmetric:

  Suppose we have,

  (5) $match_{plug-in}(C_1, C_2) = (C_{2pre} \Rightarrow C_{1pre}) \wedge (C_{1post} \Rightarrow C_{2post})$, and

  (7) $match_{plug-in}(C_2, C_1) = (C_{1pre} \Rightarrow C_{2pre}) \wedge (C_{2post} \Rightarrow C_{1post})$, then by the definition of *antisymmetry*, we will show $match_{plug-in}(C_1, C_2) \wedge match_{plug-in}(C_2, C_1)$ infers an equivalence relation $match_{exact}(C_1, C_2)$.

Given(5) and (7),

$$
\begin{aligned}
& (C_{2pre} \Rightarrow C_{1pre}) \wedge (C_{1post} \Rightarrow C_{2post}) \\
& \wedge (C_{1pre} \Rightarrow C_{2pre}) \wedge (C_{2post} \Rightarrow C_{1post}) \\
\Longleftrightarrow \quad & \langle \text{Definition of } \Leftrightarrow\text{: } p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (p \Leftarrow q) \rangle \\
& (C_{2pre} \Leftrightarrow C_{1pre}) \wedge (C_{1post} \Leftrightarrow C_{2post}) \\
= \quad & \langle \text{Definition of } match_{exact} \rangle \\
& match_{exact}(C_1, C_2) \quad \blacksquare
\end{aligned}
$$

5.5.4 Relaxed Plug-in Match

Relaxed Plug-in Match [32] is also called the *satisfies* match in [122], or *plug-in compatibility* in [54, 133]. $match_{relexed-plug-in}(NC, OC)$, defined by $(OC_{pre} \Rightarrow NC_{pre}) \wedge ((OC_{pre} \wedge NC_{post}) \Rightarrow OC_{post})$, is based on the Plug-in Match, but puts the precondition of the old contract on the postcondition as a guard to constrain the condition. Intuitively, the postcondition relation $(NC_{post} \Rightarrow OC_{post})$ only holds for inputs that satisfy the old contract's precondition OC_{pre} (domain restriction).

We have dropped quantifiers and parameters of contracts in Section 5.5.1 (page 80) for simplicity. However, to discuss the semantics of Relaxed Plug-in Match more formally, we need to include these constructions.

First of all, we may consider a rule S_i in a contract C under evolution pertains to a problem domain and problem range, similar techniques can be found in [121, 123, 133, 140]. In Section 5.2.1 (page 67), we have discussed to involve the input and the output states only. The specification of rules will then be described in terms of a problem, where D is the input domain and R is the output domain (range). We define I is a relation on D called the input condition which expresses any properties for the desired rule. O is a relation on $D \times R$ called the output condition which expresses the properties that an output should hold after executing the rule. Inputs satisfying I are called *legal* inputs. Any output value z such that $O(x, z)$ holds is called a *feasible* output with respect to an input x. We say a specification is *total* if for every legal input there exists at least one feasible output. Otherwise, a specification is *partial* if for some legal inputs there is not a feasible output. Thus, a (total) specification of a rule is a tuple $\langle D, R, I, O \rangle$, where $\forall x : D, \exists z : R.\ I(x) \Rightarrow O(x, z)$. Considering the relationship between x and z through the rule S_i, we may represent the above form as $\forall x : D, \exists z : R.\ I(x) \Rightarrow O(x, f_{S_i}(x))$, where $f_{S_i} : D \rightarrow R$. Therefore, z can be thought as an output which is generated by a well defined function f_{S_i} over legal inputs (determinism assumed), $z = f_{S_i}(x)$.

A rule S_i *satisfies* the specification if for any legal input, S_i terminates with a feasible output (total correctness assumed). In contrast, a specification is *unsatisfiable* if no feasible output can be found for each legal input.

In Section 5.4.5 (page 76), we say the pre- and postcondition of contracts is a combination of the pre- and postcondition of each individual rule, and represented by a conjunction of $(pre_{rulei} \Rightarrow post_{rulei})$. We suppose two contracts C_1 and C_2 have the same coordination rule S_i and the signatures of S_i in the two contracts match[10]. For a base case, we compare contracts C_1 and C_2 with rule S_i first. A contract C_2 *satisfies* a specification of a contract C_1 with respect to S_i, if for any of C_1's legal inputs to S_i, S_i in C_2 results in one of C_1's corresponding feasible outputs. Formally, C_2 *satisfies* C_1 with respect to S_i if both of the following conditions hold:

1. $\forall x : D_{S_i} \,.\, I_{C_1}(x) \Rightarrow I_{C_2}(x)$. Any legal input to S_i in C_1 will be a legal input to S_i in C_2. The specification of S_i in C_2 assures that a legal input to it results in a feasible output.

2. $\forall x : D_{S_i} \,.\, I_{C_1}(x) \wedge O_{C_2}(x, f_{S_i}(x)) \Rightarrow O_{C_1}(x, f_{S_i}(x))$. All feasible outputs of S_i in C_2 for a legal S_i input in C_1 are valid outputs of S_i in C_1.

We say a contract C_2 *satisfies* a contract C_1 if for every rule S_i $(i \in \mathbb{N})$ in C_1 there is a corresponding rule of C_2 and the above conditions hold for these rules. But there may be additional rules in C_2, which are absent in C_1. However, if some rules in C_1 are dropped in C_2, like the deletion operation in Figure 5.3 (page 69), we may not predict the behaviors of the evolved system. Intuitively, $\forall x : D.\, I_C(x)$ for all rules S_i $(i \in \mathbb{N})$ is the precondition of a contract C if only legal inputs are allowed, and $\forall x : D, \exists z : R.\, O(x, f_{S_i}(x))$ for all rules S_i $(i \in \mathbb{N})$ is its postcondition if feasible outputs are guaranteed. We then rewrite the above requisites to: 1. $(OC_{pre} \Rightarrow NC_{pre})$, and 2. $((OC_{pre} \wedge NC_{post}) \Rightarrow OC_{post})$. This is the way $match_{relexed-plug-in}(NC, OC)$ defined.

The relation $match_{plug-in}(NC, OC) \Rightarrow match_{relexed-plug-in}(NC, OC)$ in Figure 5.10 is proved as follows:

[10]Signature matching of contracts is defined in page 78.

$match_{plug-in}(NC, OC)$

$= \quad \langle$Definition of $match_{plug-in}\rangle$

$(OC_{pre} \Rightarrow NC_{pre}) \wedge (NC_{post} \Rightarrow OC_{post})$

$\Longleftrightarrow \quad \langle$Definition of $\Rightarrow$: $p \Rightarrow q \equiv \neg p \vee q\rangle$

$(OC_{pre} \Rightarrow NC_{pre}) \wedge (\neg NC_{post} \vee OC_{post})$

$\Longrightarrow \quad \langle$Implication – Weakening : $p \Rightarrow p \vee q\rangle$

$(OC_{pre} \Rightarrow NC_{pre}) \wedge (\neg OC_{pre} \vee \neg NC_{post} \vee OC_{post})$

$\Longleftrightarrow \quad \langle$Definition of $\Rightarrow$: $p \Rightarrow q \equiv \neg p \vee q\rangle$

$(OC_{pre} \Rightarrow NC_{pre}) \wedge ((OC_{pre} \wedge NC_{post}) \Rightarrow OC_{post})$

$= \quad \langle$Definition of $match_{relexed-plug-in}\rangle$

$match_{relexed-plug-in}(NC, OC)$ ∎

5.5.5 Guarded Generalized Match

Guarded Generalized Match is proposed by [40], and defined as $match_{guarded-gen}(NC, OC) = (OC_{pre} \Rightarrow NC_{pre}) \wedge ((NC_{pre} \Rightarrow NC_{post}) \Rightarrow (OC_{pre} \Rightarrow OC_{post}))$. In Figure 5.10, the dotted frame indicates that $match_{guarded-gen}$ is logically equivalent to $match_{relaxed-plug-in}$. We will address the proof obligation on their equivalence via a relation $OC_{pre} \Rightarrow (NC_{pre} \wedge (NC_{post} \Rightarrow OC_{post}))$.

Step 1:

$match_{relaxed-plug-in}(NC, OC)$

$= \quad \langle$Definition of $match_{relaxed-plug-in}\rangle$

$(OC_{pre} \Rightarrow NC_{pre}) \wedge ((OC_{pre} \wedge NC_{post}) \Rightarrow OC_{post})$

$\Longleftrightarrow \quad \langle$Implication – Shunting: $p \wedge q \Rightarrow r \equiv p \Rightarrow (q \Rightarrow r)\rangle$

$(OC_{pre} \Rightarrow NC_{pre}) \wedge (OC_{pre} \Rightarrow (NC_{post} \Rightarrow OC_{post}))$

$\Longleftrightarrow \quad \langle(p \Rightarrow q) \wedge (p \Rightarrow r) \equiv p \Rightarrow (q \wedge r)\rangle$

$OC_{pre} \Rightarrow (NC_{pre} \wedge (NC_{post} \Rightarrow OC_{post}))$

Step 2:

$(OC_{pre} \Rightarrow (NC_{pre} \wedge (NC_{post} \Rightarrow OC_{post}))) \Leftrightarrow ((OC_{pre} \Rightarrow NC_{pre}) \wedge ((NC_{pre} \Rightarrow NC_{post}) \Rightarrow (OC_{pre} \Rightarrow OC_{post})))$ (8) can be easily shown, for instance, by a truth table that (8) is a tautology.

So far, we have proved that $match_{guarded-gen}$ and $match_{relaxed-plug-in}$ are

equivalent[11]. ∎

We claim that $match_{guarded-gen}$ is reflexive, transitive and antisymmetric, so that it is a partial order match. Because of its proven equivalence with $match_{relaxed-plug-in}$ and $OC_{pre} \Rightarrow (NC_{pre} \wedge (NC_{post} \Rightarrow OC_{post}))$, we can infer that these two equivalences also have the partial order property.

- Reflexive:

$$
\begin{array}{ll}
& match_{guarded-gen}(C, C) \\
= & \langle \text{Definition of } match_{guarded-gen} \rangle \\
& (C_{pre} \Rightarrow C_{pre}) \wedge ((C_{pre} \Rightarrow C_{post}) \Rightarrow (C_{pre} \Rightarrow C_{post})) \\
\Longleftrightarrow & \langle \text{Reflexivity of } \Rightarrow\text{: } p \Rightarrow p \equiv true \rangle \\
& true \wedge true \\
\Longleftrightarrow & \langle \text{Reflexivity of } \wedge : \ p \wedge p \equiv p \rangle \\
& true \quad \blacksquare
\end{array}
$$

- Transitive:

  Suppose we have,

  (9) $match_{guarded-gen}(C_1, C_2) = (C_{2pre} \Rightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post}))$, and

  (10) $match_{guarded-gen}(C_2, C_3) = (C_{3pre} \Rightarrow C_{2pre}) \wedge ((C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post}))$, then by the definition of *transitivity* in Section 5.5.3, we will show

  $match_{guarded-gen}(C_1, C_3) = (C_{3pre} \Rightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post}))$.

[11] Beware of the difference between equivalent match defined in Section 5.5.2 and the claim that two matches are equivalent.

Given (9) and (10),

$$
\begin{aligned}
& ((C_{2pre} \Rightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post}))) \\
& \wedge((C_{3pre} \Rightarrow C_{2pre}) \wedge ((C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post}))) \\
\Longleftrightarrow\ & \langle \text{Symmetry of } \wedge : \ p \wedge q \equiv q \wedge p \rangle \\
& ((C_{2pre} \Rightarrow C_{1pre}) \wedge (C_{3pre} \Rightarrow C_{2pre})) \wedge \\
& ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post})) \\
& \wedge((C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post})) \\
\Longrightarrow\ & \langle \text{Transitivity of } \Rightarrow : \ (p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \rangle \\
& (C_{3pre} \Rightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post})) \\
=\ & \langle \text{Definition of } match_{guarded-gen} \rangle \\
& match_{guarded-gen}(C_1, C_3) \quad \blacksquare
\end{aligned}
$$

- Antisymmetric:

  Suppose we have,

  (9) $match_{guarded-gen}(C_1, C_2) = (C_{2pre} \Rightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post}))$, and

  (11) $match_{guarded-gen}(C_2, C_1) = (C_{1pre} \Rightarrow C_{2pre}) \wedge ((C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{1pre} \Rightarrow C_{1post}))$, then by the definition of antisymmetry in Section 5.5.3, we will show $match_{guarded-gen}(C_1, C_2) \wedge match_{guarded-gen}(C_2, C_1)$ infers an equivalence relation $(C_{2pre} \Leftrightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Leftrightarrow (C_{2pre} \Rightarrow C_{2post}))$.

  Given (9) and (11),

$$
\begin{aligned}
& (C_{2pre} \Rightarrow C_{1pre}) \wedge (C_{1pre} \Rightarrow C_{2pre}) \\
& \wedge((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post})) \\
& \wedge((C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{1pre} \Rightarrow C_{1post})) \\
\Longleftrightarrow\ & \langle \text{Definition of } \Leftrightarrow : \ p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (p \Leftarrow q) \\
& (C_{2pre} \Leftrightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Leftrightarrow (C_{2pre} \Rightarrow C_{2post}))
\end{aligned}
$$

  It is easy to show that $(C_{2pre} \Leftrightarrow C_{1pre}) \wedge ((C_{1pre} \Rightarrow C_{1post}) \Leftrightarrow (C_{2pre} \Rightarrow C_{2post}))$ is an equivalence match according to the definition in Section 5.5.2. ∎

5.5.6 Generalized Match

The Generalized (Predicate) Match is defined in [149]. In Exact Pre/Post Match, Plug-in Match, Relaxed Plug-in Match and the like, pre- and postconditions of different contracts are treated as parts; now we consider the relationship of the specifications as a whole. By specification, we mean the pair of pre- and postcondition of a contract (see definitions in Section 5.4.5 and 5.5.4). Following [149], Generic Predicate Match is defined as $match_{pred}(NC, OC) = NC_{pred}\ \mathcal{R}\ OC_{pred}$, where relation $\mathcal{R}$ is $\Rightarrow$, $\Leftrightarrow$ or $\Leftarrow$. A contract predicate has two definitions: $C_{pre} \Rightarrow C_{post}$ and a relatively stronger form, $C_{pre} \wedge C_{post}$. In this thesis, we will adopt the former definition and $\mathcal{R}$ is an implication ($\Rightarrow$), so that $match_{gen}(NC, OC)$ is defined as $(NC_{pre} \Rightarrow NC_{post}) \Rightarrow (OC_{pre} \Rightarrow OC_{post})$. We gain insight into the meaning of $match_{gen}$, given that a new contract has a *stronger* specification than the old contract. Following the explanation in Section 5.5.4 (page 86) for Relaxed Plug-in Match, $(C_{pre} \Rightarrow C_{post})$ is constructive because the precondition serves as a *guard* for the postcondition. The combined assertion "precondition implies postcondition" defines what a contract does [108].

Consequently, the overall goal of a contract predicate is to prove that the precondition of the contract (C_{pre}) implies its postcondition (C_{post}) through the contract (C). From predicate logic, the predicate $p \Rightarrow q$ is `false` only when p is `true` but q is `false`. In this case, we mean when the precondition of a contract holds, its postcondition is not satisfied, the predicate for the contract fails[12]. As is normal in mathematical proofs, we may work out such a proof $C_{pre} \Rightarrow C_{post}$ in a forward direction from the precondition towards the postcondition. However, it is empirically easier in program proofs to work backwards from the postcondition towards the precondition. Generally, in program logic, we use weakest preconditions [41, 42] to prove program specifications. That is, we denote a system (machine, mechanism) by S and the desired postcondition by Q, if the weakest precondition is $wp(S, Q)$, and the precondition $P \Rightarrow wp(S, Q)$, then $\{P\}S\{Q\}$ is `true`[13]. $wp(S, Q)$ is called a "predicate transformer". According to [70, page 109], $\{P\}S\{Q\}$ is a statement in the Hoare logic and equivalent to $P \Rightarrow wp(S, Q)$.

However, since we do not have a logic for contracts corresponding to Hoare logic and the weakest precondition calculus, we may be incapable of demonstrating an example for $match_{gen}(NC, OC)$.

The Generalized Match appears as a weaker match than the Guarded Generalized Match, i.e., $match_{guarded-gen}(NC, OC) \Rightarrow match_{gen}(NC, OC)$, as

[12] An exception in contracts is that if only the `with` clause for a rule is broken in the precondition, the effect of `failure` serves as its postcondition.

[13] Total correctness is assumed.

shown in Figure 5.10.

$$
\begin{array}{ll}
& match_{guarded-gen}(NC, OC) \\
= & \langle \text{Definition of } match_{guarded-gen} \rangle \\
& (OC_{pre} \Rightarrow NC_{pre}) \wedge ((NC_{pre} \Rightarrow NC_{post}) \Rightarrow (OC_{pre} \Rightarrow OC_{post})) \\
\Longrightarrow & \langle \text{Implication - Weakening: } p \wedge q \Rightarrow p \rangle \\
& (NC_{pre} \Rightarrow NC_{post}) \Rightarrow (OC_{pre} \Rightarrow OC_{post}) \\
= & \langle \text{Definition of } match_{gen} \rangle \\
& match_{gen}(NC, OC) \quad \blacksquare
\end{array}
$$

We claim that $match_{gen}$ is reflexive, transitive and antisymmetric, so that it is a partial order match.

- Reflexive:

$$
\begin{array}{ll}
& match_{gen}(C, C) \\
= & \langle \text{Definition of } match_{gen} \rangle \\
& (C_{pre} \Rightarrow C_{post}) \Rightarrow (C_{pre} \Rightarrow C_{post}) \\
\Longleftrightarrow & \langle \text{Reflexivity of } \Rightarrow\text{: } p \Rightarrow p \equiv true \rangle \\
& true \quad \blacksquare
\end{array}
$$

- Transitive:

  Suppose we have,

  (12) $match_{gen}(C_1, C_2) = (C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post})$, and

  (13) $match_{gen}(C_2, C_3) = (C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post})$, then by the definition of transitivity in Section 5.5.3, we will show

  $match_{gen}(C_1, C_3) = (C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post})$.

Given (12) and (13),

$$\begin{aligned}
& ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post})) \\
& \wedge((C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post})) \\
\Longrightarrow\ & \langle \text{Transitivity of } \Rightarrow\text{: } (p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \rangle \\
& (C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{3pre} \Rightarrow C_{3post}) \\
=\ & \langle \text{Definition of } match_{gen} \rangle \\
& match_{gen}(C_1, C_3) \quad \blacksquare
\end{aligned}$$

- Antisymmetric:

  Suppose we have,

  (12) $match_{gen}(C_1, C_2) = (C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post})$, and

  (14) $match_{gen}(C_2, C_1) = (C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{1pre} \Rightarrow C_{1post})$, then by the definition of antisymmetry, we will show $match_{gen}(C_1, C_2) \wedge match_{gen}(C_2, C_1)$ equals to $(C_{1pre} \Rightarrow C_{1post}) \Leftrightarrow (C_{2pre} \Rightarrow C_{2post})$, which is an equivalence relation.

  Given (12) and (14),

  $$\begin{aligned}
  & ((C_{1pre} \Rightarrow C_{1post}) \Rightarrow (C_{2pre} \Rightarrow C_{2post})) \\
  & \wedge((C_{2pre} \Rightarrow C_{2post}) \Rightarrow (C_{1pre} \Rightarrow C_{1post})) \\
  \Longrightarrow\ & \langle \text{Definition of } \Leftrightarrow\text{: } p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (p \Leftarrow q), \text{ and} \\
  & \text{Implication - Weakening: } p \wedge q \Rightarrow p \rangle \\
  & (C_{1pre} \Rightarrow C_{1post}) \Leftrightarrow (C_{2pre} \Rightarrow C_{2post})
  \end{aligned}$$

  It is easy to show that $(C_{1pre} \Rightarrow C_{1post}) \Leftrightarrow (C_{2pre} \Rightarrow C_{2post})$ is an equivalence match according to the definition in Section 5.5.2. ∎

5.5.7 Summary

Several cases of specification matches have been discussed for defining a variety of behavioral relationships between coordination contracts. In Figure 5.10 (page 80), we cannot afford to use only plug-in post match relation or guarded post match, etc., with preconditions are dropped. Considering the postcondition of contracts only is insignificant in this context, the precondition is required to hold initially and serves as a guard for postcondition. Without the validity of the precondition, the contract is even not performed, not to mention

its any potential effects. Thus, we simplify Figure 5.10 into Figure 5.11, where Exact Pre/Post Match is the most rigorous match, Generalized Match is the weakest one, and Relaxed Plug-In Match is equivalent to Guarded Generalized Match.

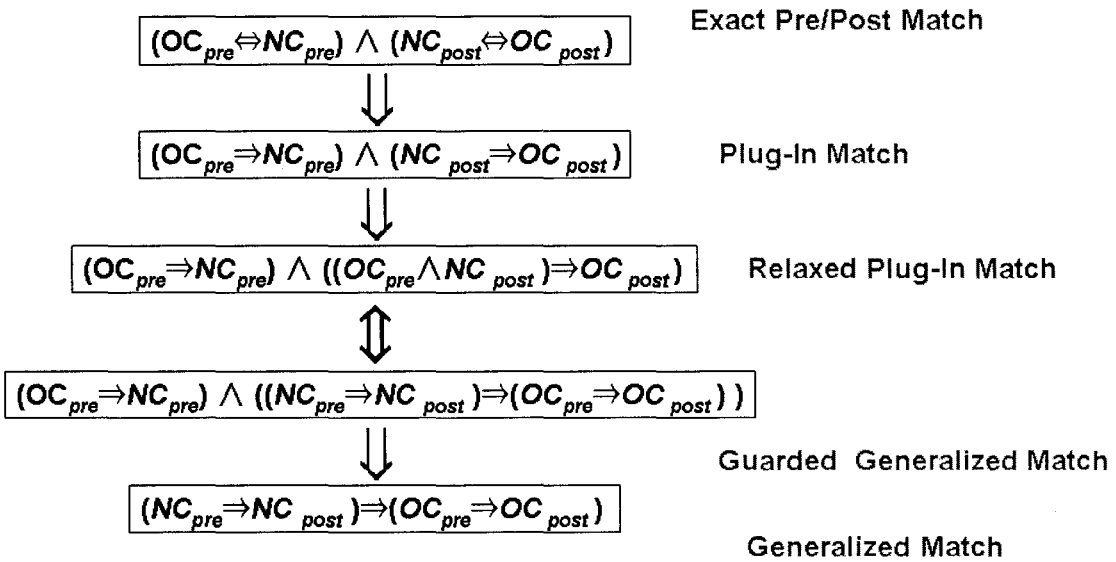


Figure 5.11: Contract Specification Matches (Simplified)

From the specification matches and the matching hierarchy, we are able to tell how different two contracts can be in terms of predictable evolution of coordination contracts. As a result, we conclude that the evolution is predictable up to the limits of these specification matches.

5.6 Summary

In the three-dimensional architecture constructed with coordination contracts, the thesis aims at evolution of the coordination dimension. This Chapter is mainly dealing with two issues. One is to characterize change histories in a way that enables control of system evolution in a predictable direction. The other is the allowable relationships between these versions in terms of contracts. Generally speaking, we would like to see incremental and predictive evolution so that we do not allow changes in a subtractive way. We define pre- and postconditions of method calls, coordination rules and coordination contracts, and use specification matching to justify the behavioral relationships between coordination contracts by means of pre- and postcondition specification. Additionally, we provide a framework to assess different cases of specification matches, and demonstrate proof sketches and properties of a variety

of matches, such as Exact Pre/Post Match, Plug-in Match, Relaxed Plug-in Match, Guarded Generalized Match and Generalized Match.

Chapter 6

Case Studies

The goal of this Chapter is to provide an opportunity to conduct studies on a set of examples to demonstrate our approach in Chapter 5 to evolving software systems in terms of coordination contracts. This approach cannot be fully comprehensible in theory only, but needs simple and adequate case studies in practical settings to illustrate its application. Driven by examples, we are about to discuss how we characterize the system change histories and how behaviors of coordination contracts are related more concretely in real examples. UML extended with a scroll notation for contracts in Section 4.4.1 (page 52) is used to describe program architectures graphically. The language for contracts and definitions of pre- and postconditions in Section 5.4 (page 74) are used to reason about contracts. In the subsequent examples, coordination contracts are mostly used to implement method invocations as the intention with which we develop this section is present representative examples rather than try to be exhaustive.

6.1 Introduction to the Banking Application

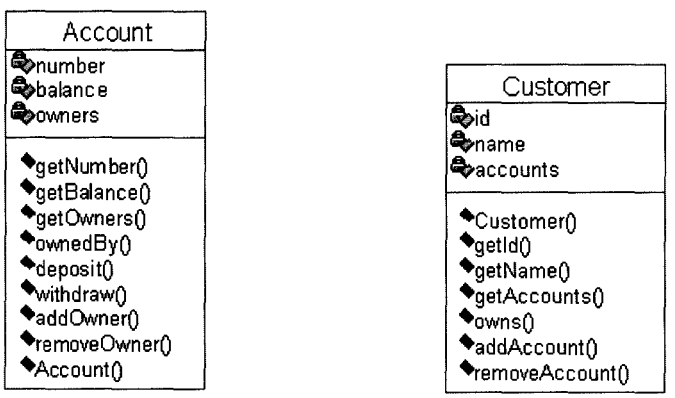
The Object-Oriented banking example is a common demonstration of OOD by attempting to model a bank account and a customer class. We base the case studies and parts of the source code on examples in the documentation distributed with CDE 1.1.1 [6], as well as on those examples appearing in the literature that we surveyed to introduce coordination contracts in Chapter 4. In this example, we do not deal with advanced features, for instance, multi-threaded bank account classes or concurrent transactions, etc.

We have two participant components, *Account* (Figure 6.1(a)) and *Customer* (Figure 6.1(b)). For simplicity reason, the banking example is up to binary relationships. However, when applied, contracts may involve more than

two partners. According to their class diagrams, a customer can invoke several operations on an account. In this Chapter, for ease, we particularly are concerned in basic services as follows:

- `double` getBalance(), this operation returns the balance of an account (see Table 6.4);
- `void` deposit (`double` amount, Customer c), this operation increases the balance held by customer *c* by *amount*, and returns nothing;
- `void` withdraw (`double` amount, Customer c), this operation decreases the balance held by customer *c* by *amount*, and returns nothing (see Table 6.2).

The *withdraw* and *deposit* methods have no restrictions, except that the *amount* sought must not be negative and withdrawals may only be authorized to a customer who owns the account. The ownership can be checked by the *owns*() method in the *Customer* class or the *ownedBy*() method in the *Account* class.



(a) *Account* Class (b) *Customer* Class

Figure 6.1: Case Studies Class Diagrams

6.2 System Change Histories

In Section 5.2 (page 67), we have demonstrated several change histories that may support evolution in an incremental and predictable way. Now we examine these cases with the banking application, which is composed of the

Account class, the *Customer* class and a contract *ContractBank* coordinating the objects of these two classes (Figure 6.2). As a desirable feature of coordination contracts, *Account* and *Customer* are not aware of the existence of *ContractBank*. We suppose systems SYS, SYS_1 and SYS_2 are instantiating the proposed architecture and varying only in the contract *ContractBank*.

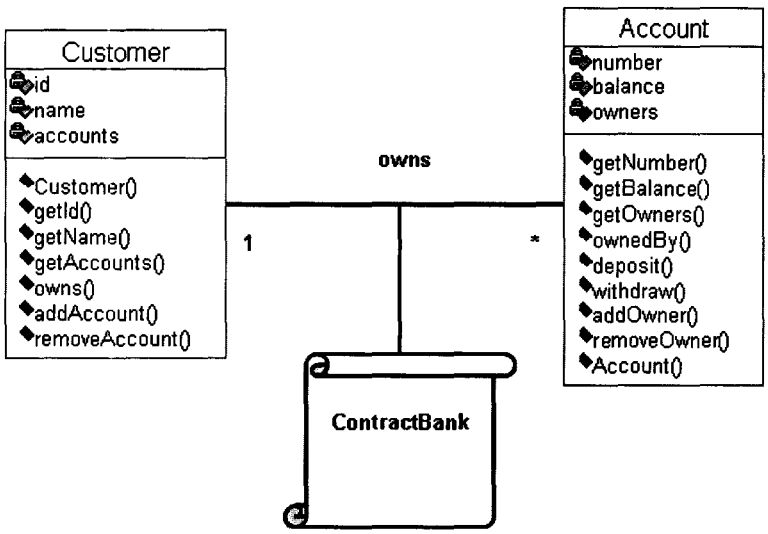


Figure 6.2: Case Studies — A Bank Application

Initially, a contract *ContractBank* either has the *WithdrawRule* as in Table 6.1, or the *BalanceRule* as in Table 6.3, or both rules as in Table 6.5. The functionalities of *WithdrawRule* and *BalanceRule* are pass on method calls, where Table 6.2 and Table 6.4 include the code for original methods *withdraw*() and *getBalance*() in the *Account* class, individually.

Table 6.1: *ContractBank* with *WithdrawRule*

```
contract ContractBank
  participants
    customer : Customer; account : Account;
  coordination
    WithdrawRule:
     when *- >> account.withdraw(amount, c) && (customer == c)
     with (account.getBalance() >= amount)
  ;
end contract
```

Table 6.2: The *withdraw* method in *Account* Class

```
public void withdraw (double amount, Customer c)
        throws AccountException {
  if (amount < 0)
        throw new AccountException(this, amount, c,
              NEGATIVE_AMOUNT);
  if (!ownedBy(c))
        throw new AccountException(this, amount, c, NOT_OWNER);
  balance − = amount;
}
```

Table 6.3: *ContractBank* with *BalanceRule*

```
contract ContractBank
  participants
    account : Account;
  coordination
    BalanceRule:
     when ⋆− >> account.getBalance()
  ;
end contract
```

Table 6.4: The *getBalance* method in *Account* Class

```
public double getBalance() { return balance; }
```

Table 6.5: *ContractBank* with *WithdrawRule* and *BalanceRule*

```
contract ContractBank
  participants
    customer : Customer; account : Account;
  coordination
    WithdrawRule:
     when ⋆− >> account.withdraw(amount, c) && (customer == c)
     with (account.getBalance() >= amount)
    BalanceRule:
     when ⋆− >> account.getBalance()
  ;
end contract
```

Case 1

SYS, SYS_1 and SYS_2 in Figure 6.3 are systems evolving "sequentially" from each other like a relay. Supposedly, the *ContractBank* in SYS has only one

coordination rule for *withdraw*() in Table 6.1. SYS_1 evolves from SYS with this *WithdrawRule*[1] in Table 6.6 by considering an exception, and SYS_2 evolving from SYS_1 by adding or modifying a rule, *BalanceRule*[2] in Table 6.3. The *ContractBank* in the evolved system SYS_2 then takes the form in Table 6.5.

Hence, we are able to combine these two cumulative changes into Figure 6.4, and make a direct evolution process with both of the two coordination rules *WithdrawRule* and *BalanceRule* in the contract(s) from SYS to SYS_2. It is also possible that SYS_2 modifies the *WithdrawRule* based on SYS_1, following the previous change from SYS to SYS_1; see Figure 6.5 and Figure 6.6. The sequence of transitions from Table 6.1 to Table 6.6 then Table 6.8 illustrates an occurrence of the described process.

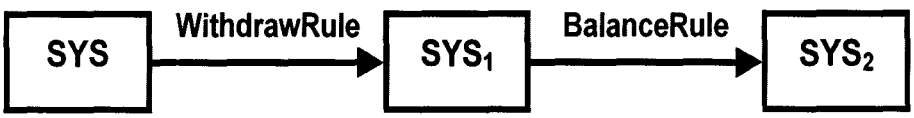


Figure 6.3: Case 1 — Adding/Modifying a Different Rule

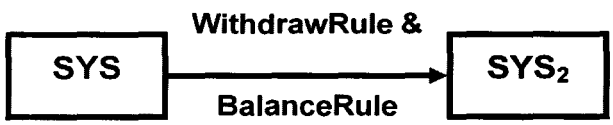


Figure 6.4: Case 1 — Accumulating the Changes of Two Rule

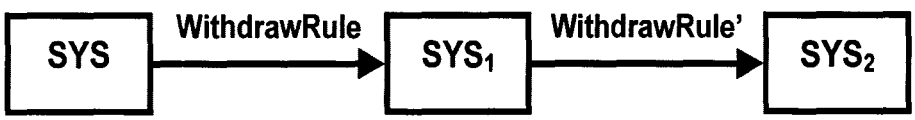


Figure 6.5: Case 1 — Changing the Current Rule

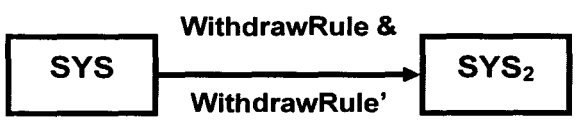


Figure 6.6: Case 1 — Accumulating Changes of the Same Rule

[1]In practice, the rule may be newly added to a contract or modified an existing contract in SYS.

[2]The rule may or may not be in the same contract as the *WithdrawRule*.

Table 6.6: *ContractBank1* with *WithdrawRule1*

```
contract ContractBank1
  participants
    customer : Customer; account : Account;
  coordination
    WithdrawRule1:
     when ⋆− >> account.withdraw(amount, c) && (customer == c)
     with (account.getBalance() >= amount)
     failure {
      throw new AccountException(account, amount, c,
          AccountExceptionTypes.LIMIT_EXCEEDED); }
  ;
end contract
```

Case 2

SYS_1 and SYS_2 in Figure 6.7 are evolving "individually" based on SYS. SYS_1 evolves SYS with the coordination rule *WithdrawRule* in a contract, and SYS_2 evolves SYS with *BalanceRule* in the same or a different contract from *WithdrawRule*. We may not be able to fully justify the predictable relationship between SYS_1 and SYS_2 by means of the behavioral specification matching approach in Section 5.5.

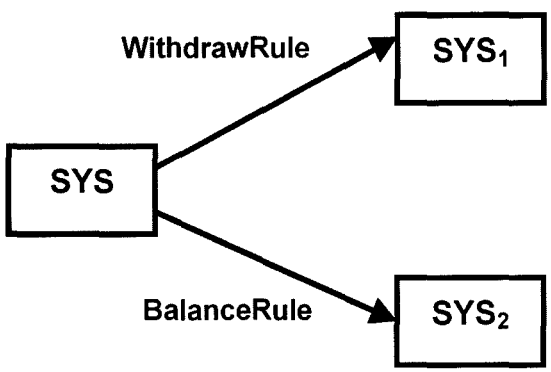


Figure 6.7: Case 2

Furthermore, we argue that even though SYS_1 and SYS_2 may evolve SYS, respectively, with the same coordination rule *WithdrawRule* in the same contract (see Figure 6.8[3]) in a different way, we still may not make a full justification of prediction between SYS_1 and SYS_2 by the relations of *ContractBank*

[3]We name *WithdrawRule* by *WithdrawRule1* and *WithdrawRule2* in Figure 6.8 only to show their any potential difference in versions. In their specifications, the two coordination rules may have the same name.

since we compare the effect of the `do` block and the `failure` clause independently.

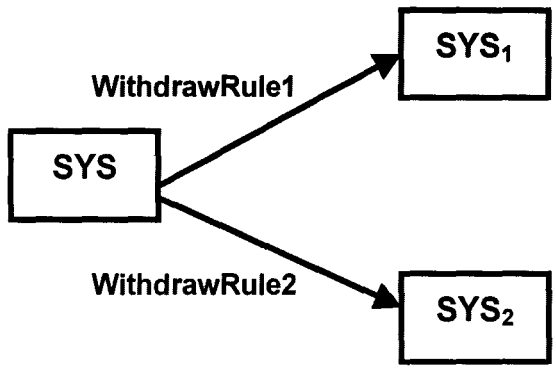


Figure 6.8: Case 2′

Now we demonstrate this scenario with an example. SYS has the initial version of *WithdrawRule* in Table 6.1. Suppose in SYS_1, we evolve to the *ContractBank1* with the *WithdrawRule1* in Table 6.6. The `do` block is omitted so that the original method *withdraw* will be executed instead. If the guard in the `with` clause, "account.getBalance() >= amount", is `false`, the contract raises a defined exception accordingly. However, in SYS_2 in Table 6.7, there is no such `with` clause. Even if "account.getBalance() >= amount" is `false`, the *ContractBank2* in Table 6.7 is able to perform the *withdraw* function and decreases that account by the amount of its remaining balance. Based on this understanding, we conclude that even though the evolved systems SYS_1 and SYS_2 are related to SYS in some sense, the relationship between SYS_1 and SYS_2 in terms of the *WithdrawRule* in *ContractBank* is as yet unclear.

Table 6.7: *ContractBank2* with *WithdrawRule2*

```
contract ContractBank2
  participants
    customer : Customer; account : Account;
  coordination
    WithdrawRule2:
     when *- >> account.withdraw(amount, c) && (customer == c)
     before {System.out.println("Amount asked for: " + amount);}
     do {account._withdraw(Math.min(account.getBalance(),
                                    amount), c);}
  ;
end contract
```

Case 3

SYS_1 and SYS_2 are systems evolved from SYS individually, such as the situation in Figure 6.7. SYS_3 can be achieved in evolving either SYS_1 or SYS_2 in Figure 6.9 by extending the contract(s) with *BalanceRule* and *WithdrawRule*, respectively. Following the scenario in Case 1 (Figure 6.4), we combine these two cumulative changes in Figure 6.10, that is, either accumulating the effects of *WithdrawRule* then *BalanceRule* on SYS, or vice versa. We will cover this case with an example in Section 6.3.3.

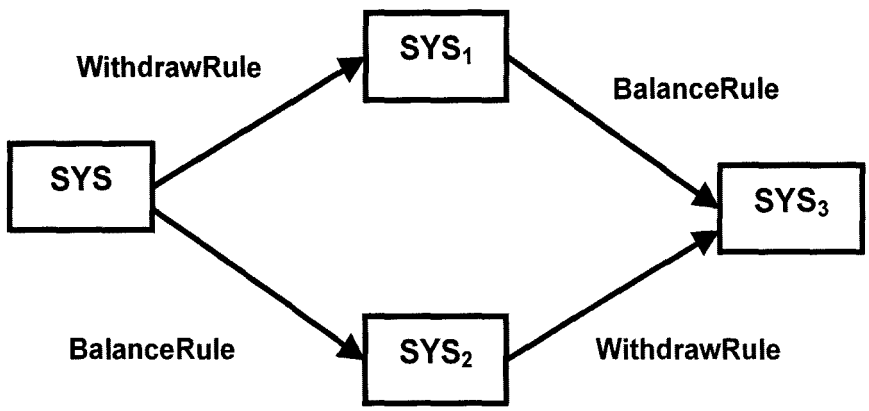


Figure 6.9: Case 3

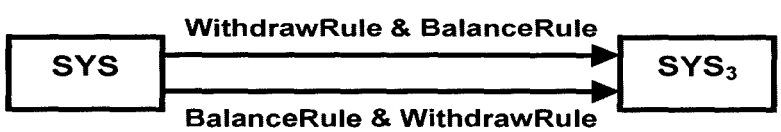


Figure 6.10: Case 3′

6.3 Behavioral Relationships and Specification Matching

In what follows, we will study a variety of contracts applied in the banking application with reasoning their behavioral relationships and the matching rules discussed in Chapter 5. By "changes", we principally refer to the behavioral changes rather than the syntactic ones. If we modify the contract merely syntactically, for example, renaming the rules or contracts, Exact Pre/Post Match succeeds since their preconditions, their postconditions and any other effects are the same. However, Exact Pre/Post Match is a case not likely to occur

often. In addition, to prove the equivalence of two contracts completely, we may need a proper semantics for contracts, which is left for future work.

6.3.1 Changes to the Precondition of Coordination Contracts

To promote a monthly package, a new withdraw rule *WithdrawRule3* in Table 6.8 is substituted for the *WithdrawRule1* in Table 6.6. However, *WithdrawRule3* has a weaker guard than *WithdrawRule1*, since in the `with` clauses, (account.getBalance() >= amount) ⇒ ((account.getBalance() + limit) >= amount), given limit ≥ 0. By the definition in Section 5.4.3 (page 75), the relevant precondition of coordination rules is defined as $pre_{op} \wedge trigger_{call} \wedge with_{call}$. For the case of *WithdrawRule1* and *WithdrawRule3*, they both have the same pre_{op} as the original method *withdraw*() in class *Account*, and the same trigger, "`when` ⋆− >> account.withdraw (amount, c) && (customer == c)".

On the other hand, in the *WithdrawRule3*, other than the `with` clause, the rest parts are intact as compared to the *WithdrawRule1*. Because the `do` block is skipped in both rules, they have the same postcondition of *withdraw*() ($post_{op}$) if the guard is enabled. If the guard fails, they raise of the same exception as defined in the rule body. Consequently, the effect and exceptions of these rules are identical. Therefore, by the definition in Section 5.4.4 (page 76), the postconditions of *WithdrawRule1* and *WithdrawRule3* are the same.

Recognizing these facts and the definitions in Section 5.4.5, we infer that *ContractBank3* with *WithdrawRule3* (Table 6.8) has a weaker precondition than *ContractBank1* with *WithdrawRule1* (Table 6.6), and *ContractBank3* has the same postcondition as *ContractBank1*, i.e., $(ContractBank1_{pre} \Rightarrow ContractBank3_{pre}) \wedge (ContractBank3_{post} \Leftrightarrow ContractBank1_{post})$.

6.3.2 Changes to the Postcondition of Coordination Contracts

In Section 6.1, the method call *getBalance*() is declared in Table 6.4. We initiate a contract *ContractBank* in Table 6.3 for passing on this call merely. To avoid any ambiguity, we rename the *ContractBank* to *ContractBank4*, and *BalanceRule* to *BalanceRule1* in Table 6.9. As we discussed in the beginning of this section, the *ContractBank4* (Table 6.9) is equivalent to the *ContractBank* in Table 6.3 due to the syntactic changes.

To save daily operation costs, we secure the banking accounts with a new rule *BalanceRule2* in Table 6.10, which requires a minimum balance in ac-

Table 6.8: *ContractBank3* with *WithdrawRule3*

```
contract ContractBank3
  participants
    customer : Customer; account : Account;
  attributes
    double limit = 100.0;
  coordination
    WithdrawRule3:
     when ⋆− >> account.withdraw(amount, c) && (customer == c)
     with ((account.getBalance() + limit) >= amount)
     failure {
      throw new AccountException(account, limit+amount, c,
          AccountExceptionTypes.LIMIT_EXCEEDED); }
  ;
end contract
```

Table 6.9: *ContractBank4* with *BalanceRule1*

```
contract ContractBank4
  participants
    account : Account;
  coordination
    BalanceRule:
     when ⋆− >> account.getBalance()
  ;
end contract
```

counts. After returning the balance of the account, we check the current balance and the status of a Boolean variable *lock*. If the balance is less than the required minimum amount, and if it is not locked yet, we then lock the account and prohibit any withdrawal. If the current balance is greater than the minimum amount and the account is locked, then we unlock the account. The new feature will not lock the *deposit*() method; though *withdraw*() is disabled, *deposit*() can be used to unlock the account by increasing the balance.

The two rules, *BalanceRule1* and *BalanceRule2*, have the same precondition, since they have the same precondition as the original method, the same trigger condition and the same guard, i.e., $pre_{op} \wedge trigger_{call_1} \wedge with_{call_1} \equiv pre_{op} \wedge trigger_{call_2} \wedge with_{call_2}$. Hence, we say contracts *ContractBank5* (Table 6.10) and *ContractBank4* (Table 6.9) have an equivalent precondition by the definitions in Section 5.4.3 (page 75) and Section 5.4.5 (page 76).

On the other hand, the two rules have the same postcondition of the method

getBalance() in class *Account*, denoted as $post_{op}$. The effects of the `do` blocks in both rules are the same since they both omit this part, so that the original method is performed as the default. On examination, *BalanceRule2* has an extra `after` block compared to *BalanceRule1*. As a result, the former rule has a stronger postcondition than the latter since it has a stronger joint effect in the `before-do-after` block by the definition in Section 5.4.4 (page 76). No `failure` part is available to compare in both rules.

Taking account of the above facts and the definitions in Section 5.4.5 (page 76), we infer that the contract *ContractBank5* with *BalanceRule2* (Table 6.10) has a stronger postcondition than the contract *ContractBank4* with *BalanceRule1* (Table 6.9), and *ContractBank5* has the same precondition as *ContractBank4*, i.e.,

$$(ContractBank4_{pre} \Leftrightarrow ContractBank5_{pre}) \wedge (ContractBank5_{post} \Rightarrow ContractBank4_{post}).$$

Table 6.10: *ContractBank5* with *BalanceRule2*

```
contract ContractBank5
  participants
    account : Account;
  attributes
    double MIN = 10.0;
    boolean lock = false;
  coordination
    BalanceRule2:
     when *- >> account.getBalance()
     after {
        if (account.getBalance() < MIN && lock == false) {
           lock = true;
        }
        else if (account.getBalance() >= MIN && lock == true) {
           lock = false;
        }
     }
  ;
end contract
```

In Table 6.11, we introduce a customer participant in *ContractBank6* with *BalanceRule3* to print out the account's owner and the account's ID. The customer instances can be created by instantiating the *Customer* class existing in the architecture (Figure 6.2, page 99). Actually, an object of a newly-

introduced class is permissible in the contract as well, for example, an instance that realizes the logging information of the account objects [5].

Applying a similar reasoning in the above subsections, we infer that *BalanceRule3* has a stronger postcondition than *BalanceRule2*, so that *ContractBank6* strengthens the postcondition of *ContractBank5*, as expected. The two rules have the same precondition, so that *ContractBank6* has the same precondition as *ContractBank5*.

Table 6.11: *ContractBank6* with *BalanceRule3*

```
contract ContractBank6
  participants
    customer : Customer; account : Account;
  attributes
    double MIN = 10.0;
    boolean lock = false;
    long number;
  coordination
    BalanceRule3:
     when *- >> account.getBalance()
     after {
        if (account.getBalance() < MIN && lock == false) {
            lock = true;
            number = account.getNumber();
            customer = account.getOwners();
            System.out.println(customer.getName() + " 's account"
                + number + "is locked.");
        }
        else if (account.getBalance() >= MIN && lock == true) {
            lock = false;
            number = account.getNumber();
            customer = account.getOwners();
            System.out.println(customer.getName() + " 's account"
                + number + "is unlocked.");
        }
     }
  ;
end contract
```

6.3.3 Changes to the Precondition and Postcondition of Coordination Contracts

We integrate the *BalanceRule1* (Table 6.9) and the *WithdrawRule1* (Table 6.6) into Table 6.12 to make a new contract *ContractBank7* including both rules. The *BalanceRule1* passes on the original method call *getBalance*() and returns the balance. The *WithdrawRule1* checks if the desired withdrawal amount is greater than the balance, throws an exception if it is not, or it executes the original method *withdraw*() alternatively. According to the change histories we discussed, *ContractBank7* extends *ContractBank1* in Table 6.6 with *BalanceRule1*, and extends *ContractBank4* in Table 6.9 with *WithdrawRule1*.

Table 6.12: *ContractBank7* with *BalanceRule1* and *WithdrawRule1*

```
contract ContractBank7
  participants
    customer : Customer; account : Account;
  coordination
    WithdrawRule1:
     when *- >> account.withdraw(amount, c) && (customer == c)
     with (account.getBalance() >= amount)
     failure {
      throw new AccountException(account, amount, c,
          AccountExceptionTypes.LIMIT_EXCEEDED); }
    BalanceRule1:
     when *- >> account.getBalance()
  ;
end contract
```

The intention of *lock* added in *BalanceRule2* in Table 6.10 is to supervise affected operations like *withdraw*. To reflect such changes on *WithdrawRule3* in Table 6.8, we need the *WithdrawRule4* in Table 6.13. A `before` block is inserted, which checks the status of *lock* before performing the `do` block. By the definition in Section 5.4.4 (page 76), the `before` block is a part of the postcondition for rules.

Following the similar analysis above, we conclude that,

- *WithdrawRule4* in Table 6.13 has a weaker precondition and stronger postcondition than *WithdrawRule1* in Table 6.12.
- *BalanceRule2* in Table 6.13 has a stronger postcondition than *BalanceRule1* in Table 6.12.

Table 6.13: *ContractBank8* with *BalanceRule2* and *WithdrawRule4*

```
contract ContractBank8
  participants
    customer : Customer; account : Account;
  attributes
    double limit = 100.0;
    double MIN = 10.0;
    boolean lock = false;
  coordination
    WithdrawRule4:
     when *- >> account.withdraw(amount, c) && (customer == c)
     with ((account.getBalance() + limit) >= amount)
     before {
        if (lock == true) {
           System.out.println("Balance is Not Enough!
                  This account is LOCKED!");
           System.exit();
        }
     failure {
      throw new AccountException(account, limit+amount, c,
         AccountExceptionTypes.LIMIT_EXCEEDED); }
    BalanceRule2:
     when *- >> account.getBalance()
     after {
        if (account.getBalance() + limit < MIN && lock == false) {
           lock = true;
        }
        else if (account.getBalance() + limit >= MIN && lock == true)
        {
           lock = false;
        }
     }
  ;
end contract
```

By the definition of pre- and postcondition of coordination contracts in Section 5.4.5 (page 76), and the concept of Plug-in Match in Section 5.5.3 (page 83), the relationship between *ContractBank8* and *ContractBank7* satisfies the Plug-in Match, that is,

$$match_{plug-in}(ContractBank8, ContractBank7) = (ContractBank7_{pre} \Rightarrow ContractBank8_{pre}) \wedge (ContractBank8_{post} \Rightarrow ContractBank7_{post}).$$

Since the Plug-in Match is the second strongest match in the strength ordering in Figure 5.11, the relationship between *ContractBank8* and *ContractBank7* also satisfies the inferior matches.

In Section 4.4.2 (page 53), when introducing the CDE-specific language for contracts, we have illustrated a proposed idea of "inheritance" of contracts in Figure 4.6 (page 59) without a formal explanation of the exact meaning. At this moment, we have acquired an understanding of "inheritance" as this kind by the Plug-in Match relationship between contracts.

To represent the third case of change histories in Section 6.2 (page 98) more concretely, we assume that the system with *ContractBank8* is SYS_3 in Figure 6.11. SYS has the initial version of *ContractBank* in Table 6.5. SYS_1 evolves from SYS by *WithdrawRule3* in *ContractBank10* (Table 6.15), and SYS_2 evolves from SYS by *BalanceRule2* in *ContractBank9* (Table 6.14). SYS_3 can be achieved in evolving either from *ContractBank9* by *WithdrawRule4*, or from *ContractBank10* by *WithdrawRule4* and *BalanceRule2*. Comparing Figure 6.11 and Figure 6.9, the difference is in the evolution process from SYS_1 to SYS_3, where changes in *WithdrawRule* and *BalanceRule* are required in the former. The modification of *WithdrawRule3* is performed because *WithdrawRule3* is supposed to be dependent on the new feature of *BalanceRule2* according to the specification.

Given other examples, if the effect of a rule changes but its dependent rules do not reflect those changes correspondingly, we argue that the evolution is still well-defined in the sense that the kind of matching can be characterized. It is the designer's obligation to make sure that contracts are well-designed and implemented.

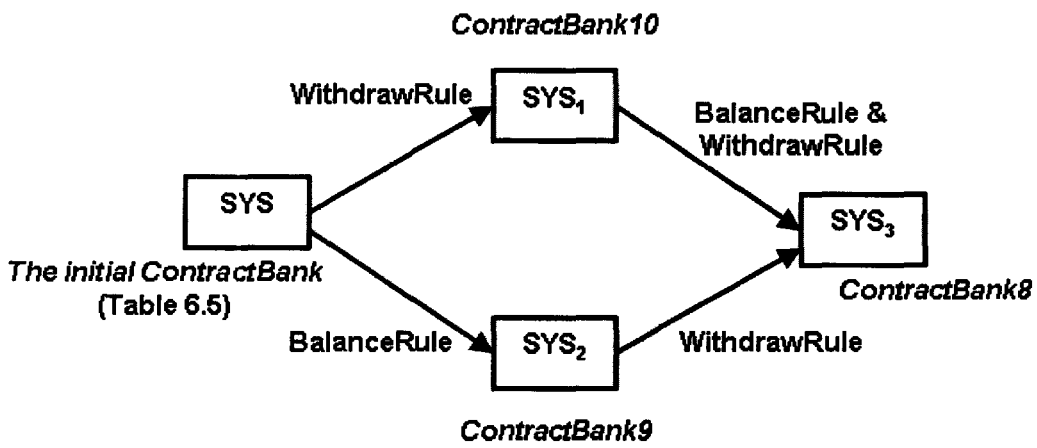


Figure 6.11: A More Specific Example for Case 3

Table 6.14: *ContractBank9* with *BalanceRule2* and *WithdrawRule1*

```
contract ContractBank9
  participants
    customer : Customer; account : Account;
  attributes
    double limit = 100.0;
    double MIN = 10.0;
    boolean lock = false;
  coordination
    WithdrawRule1:
     when *- >> account.withdraw(amount, c) && (customer == c)
     with (account.getBalance() >= amount)
     failure {
      throw new AccountException(account, amount, c,
          AccountExceptionTypes.LIMIT_EXCEEDED); }
    BalanceRule2:
     when *- >> account.getBalance()
     after {
        if (account.getBalance() + limit < MIN && lock == false) {
           lock = true;
        }
        else if (account.getBalance() + limit >= MIN && lock == true)
        {
           lock = false;
        }
     }
  ;
end contract
```

6.4 Summary

In this Chapter, we performed some case studies on a banking example to instantiate the corresponding concepts and approaches in the previous Chapter. In particular, we demonstrated Exact Pre/Post Match and Plug-in Match with this example.

Table 6.15: *ContractBank10* with *BalanceRule1* and *WithdrawRule3*

```
contract ContractBank10
  participants
    customer : Customer; account : Account;
  coordination
    WithdrawRule3:
     when *- >> account.withdraw(amount, c) && (customer == c)
     with (account.getBalance() + limit >= amount)
     failure {
      throw new AccountException(account, amount + limit, c,
          AccountExceptionTypes.LIMIT_EXCEEDED); }
    BalanceRule1:
     when *- >> account.getBalance()
  ;
end contract
```

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The research in this thesis was motivated by an apparent lack of techniques to support predictable Software Evolution based on architectures, especially on architectural connectors. We present an incremental, lightweight approach which addresses the problem by means of coordination contracts. Several contributions toward the resolution of this problem have been made.

This thesis has been concerned with the research position of Software Evolution and Software Architecture. Software Evolution is usually defined by referring to Software Maintenance. Software Maintenance and Software Evolution are the longest and the most expensive phase in the software development life cycle, and usually performed after delivery. To enhance the system's ability to change and save cost, we arrive to the assertion that Software Evolution is unavoidable, and review a software system as an entity under development as well as evolution. As the output of the design process, software architectures describe the structure of a system or a program and its global properties. By elevating the abstraction level to an earlier phase, we define Software Evolution by means of architecture re-configuration in terms of evolving operations on architectural elements, that is, adding, removing, replacing components and/or connectors, according to the required changes.

Some previous work for modeling Software Evolution have been reviewed using two main criteria: the representation of changes in architectures and the mechanism to evolve connectors. The representative work includes Lucena and Alencar's logical framework, Medvidovic et al.'s architectural type theory and transformation techniques, including a UML-based Algebraic Graph Rewriting, and Fiadeiro et al.'s approach. Except for Fiadeiro et al.'s approach, these approaches either have not an explicit representation of connectors or

they have not established an effective mechanism for connectors.

We have also conducted another literature survey with the goal of representing architectural connectors. Connectors deserve to be first-class entities primarily because they are exactly the corresponding elements which reflect the increasingly complex business rules and their interactions in the setting of software architecture. Taxonomies of connector types, notations and techniques for modeling connectors have been discussed extensively. To bridge the gap between architectural level and implementation level of connectors, coordination contracts were introduced by Fiadeiro et al. as a realization of connectors in program architectures. A coordination contract is a modeling and implementation primitive superposing behaviors on participant components that allows "transparent interception" of method calls. The approach transcends the phases of software design and implementation according to the definition. We call our approach multiple dimensional since a three-layer architecture applied on coordination contracts is proposed to separate concerns of components, connectors and configuration during evolution. For the evolution of the component dimension, we assume Medvidovic et al.'s work on C2 components is constructive, which evolve using subtyping theory as a framework for reasoning about evolution.

This thesis's major contribution is to provide a foundation for applying specification matching based methods to contracts to predict software evolution. We borrow specification matching techniques for components and compose them in an original way that is tailored to our specific needs for coordination contracts. Change histories which may relate several evolving systems are characterized to increase the efficiency of the approach. To capture observable behaviors of contracts, we describe pre- and postcondition specification for coordination contracts using a combination of abstract and CDE language for contracts. Behavioral relationships between coordination contracts have been established by a range of matches with various degrees of similarity, such as Exact Pre/Post Match, Plug-in Match, Relaxed Plug-in Match, Guarded Generalized Match and Generalized Match. These matches provide support for preserving the system behaviors in a preferred level and help to control the evolution in a predictable direction.

7.2 Future Work

We have demonstrated an approach for modeling predicable software evolution on connectors using coordination contracts. However, the problem of architecture-based software evolution is by no means completely solved, much remains to be done.

- **Complements to the Approach**
  This approach will be extended in various ways in the future. A number of features that were assumed above, but are not included in this work could be studied and implemented.

  - For the component dimension, although Medvidovic et al.'s subtyping theory to evolve C2 components is proposed to be applied on participant components being coordinated in contracts, details are left to explore.
  - Before we present specification matching, signature matching of contracts is assumed. To match parameters and their types for each rule in contracts is still an open issue.
  - It will be interesting and worthwhile to find out relevant situations that our approach does not fully cover. For instance, even though their specifications mismatch when considering preconditions and postconditions separately or together, two contracts may be related in some sense.
  - Due to the lack of a logic for contracts corresponding to Hoare logic and the weakest precondition calculus, the specification matching approach is short of a precise representation. With such a formal mapping, we can better answer the question and demonstrate more case studies.

- **Correctness of Coordination Contracts**
  In this thesis, we characterize a contract in terms of pre- and postcondition specification because the execution of contracts is basically a set of sequential activities. However, we have not talked about the correctness of coordination contracts yet, so that this thesis does not present a sound way to reason about contracts. Proof obligations might be developed to show the consistency of contracts. The presentation of mathematical semantics for coordination contracts is based on CommUnity, [49, 50, 51] are some references where this topic is discussed. For knowledge of CommUnity and Category Theory, referring to [48] and [52] and the like is advisable.

- **Non-Functional Property (NFP)**
  In Section 4.2 (page 49), we have discussed four levels of contract abstraction. Coordination contracts are classified into the group of synchronization contracts and used primarily to analyze functional properties in this thesis. The "top" level is the quality-of-service contract, which specifies all behavioral properties including even NFPs like availability, throughput, latency and capacity, etc. Since our approach to software evolution

is based on a three-layer architecture which may have NFPs, further study in such aspect seems necessary as well.

- **Scalability**

  In Section 3.3.1 (page 40), we state our standpoint in employing coordination contracts to bridge the gap between specification and implementation of software architectures, so that our approach is at the detailed architecture to source code level (Figure 1.2, page 10). However, systems or applications may further grow in size and complexity because of the incremental approach of evolution. System scale will pose challenges to a broad range of software development issues. Thus, scalability is an open question. Although some case studies have been carried out in Chapter 6, a larger size example to match more complicated contracts is still needed to make the case more convincing.

- **Detecting Invariants**

  Invariants are one of the indispensable constituents in studies of science of programming, which is a set of properties that are true over the observed executions, and prevents changes from violating assumptions for correct behaviors. We do not address the puzzle in our approach, how to discover invariants in coordination contracts. Daikon[1] is an invariant detector that discovers them from the code by static analysis and dynamic analysis in annotated programs. But many popular methods created for programs may not be applied to contracts directly. In addition, the previously held invariants may change when introducing more components to the contract during evolution, which makes the situation more complicated.

- **Software Environments**

  In Section 5.5 (page 78), we suggest FOL or OCL be used to formulate pre- and postconditions of contracts. Therefore, an matchmaker may be integrated in CDE 1.1.1 to prove relationships between contracts automatically. In addition, an evolution manager is proposed to control system change histories.

[1] `http://pag.csail.mit.edu/daikon/`

Bibliography

[1] J2EE API specifications. In http://java.sun.com/javaee/.

[2] JAVA API specifications. In http://java.sun.com/javase/.

[3] Software architecture definitions. In http://www.sei.cmu.edu/architecture/definitions.html.

[4] *IEEE Standard 1219: Standard for Software Maintenance.* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[5] Personal communication from Dr. Tom Maibaum, 2005–2007.

[6] CDE 1.1.1. http://www.atxsoftware.com/cde/.

[7] N. Aguirre and T.S.E. Maibaum. A temporal logic approach to component-based system specification and reasoning. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, FL, USA, 2002.

[8] Jonathan Aldrich. *Using Types to Enforce Architectural Structure.* PhD thesis, University of Washington, August 2003.

[9] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM Press.

[10] Paulo S. C. Alencar and Carlos Jose Pereira de Lucena. A logical framework for evolving software systems. In *Formal Aspects of Computing*, volume 8, pages 3–46, 1996.

[11] Robert Allen and David Garlan. A formal basis for architectural connection. In *ACM Trans. Softw. Eng. Methodol.*, volume 6, pages 213–249. ACM Press, 1997.

[12] Luis Andrade, Jose Fiadeiro, Joao Gouveia, Georgios Koutsoukos, Antonia Lopes, and Michel Wermelinger. Coordination patterns for component-based systems. In *Proc. of the 5th Brasilian Symposium on Programming Languages*, 2001.

[13] Luis Andrade, Jose Fiadeiro, Antonia Lopes, and Michel Wermelinger. Theory and practice of coordination technologies. In *A Tutorial at Formal Methods Europe 2002*, July 2002.

[14] Luis Filipe Andrade and Jose Luiz Fiadeiro. Evolution by contract. In *Proceedings of the ECOOP'00 Workshop on Object-Oriented Architectural Evolution*, 2000.

[15] Luis Filipe Andrade and Jose Luiz Fiadeiro. Coordination technologies for Web-Services. In *OOPSLA Workshop on Object-Oriented Web Services*, 2001.

[16] Luis Filipe Andrade and Jose Luiz Fiadeiro. Agility through coordination. In *Inf. Syst.*, volume 27, pages 411–424, Oxford, UK, 2002. Elsevier Science Ltd.

[17] Luis Filipe Andrade and Jose Luiz Fiadeiro. Architecture based evolution of software systems. In Marco Bernardo and Paola Inverardi, editors, *Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003*, volume 2804 of *LNCS*, pages 148–182, Bertinoro, Italy, 2003. Springer-Verlag.

[18] Luis Filipe Andrade and Jose Luiz Fiadeiro. Composition contracts for service interaction. In *Journal of Universal Computer Science*, volume 10, pages 375–390, 2004.

[19] Luis Filipe Andrade, Jose Luiz Fiadeiro, Joao Gouveia, and Georgios Koutsoukos. Separating computation, coordination and configuration. In *Journal of Software Maintenance*, volume 14, pages 353–369, New York, NY, USA, 2002. John Wiley & Sons, Inc.

[20] Luis Filipe Andrade, Jose Luiz Fiadeiro, Joao Gouveia, Antonia Lopes, and Michel Wermelinger. Patterns for coordination. In G.Catalin-Roman and A.Porto, editors, *COORDINATION'00*, pages 317–322, 2000.

[21] Farhad Arbab. What do you mean, coordination? In *March '98 issue of the Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, March 1998.

[22] Ralph-Johan Back, Luigia Petre, and Ivan Porres Paltor. Analysing UML use cases as contracts. In *"UML" '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference*, pages 518–533. Springer Berlin/Heidelberg, 1999.

[23] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[24] Stephanie Balzer, Patrick Th. Eugster, and Bertrand Meyer. Can aspects implement contracts? In *RISE 2005 (Rapid Implementation of Software Engineering Techniques)*, Heraklion, Greece, 2005. Springer.

[25] Marco Antonio Barbosa and Luís Soares Barbosa. Specifying software connectors. In Zhiming Liu and Keijiro Araki, editors, *ICTAC*, volume 3407 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2004.

[26] Leonor Barroca and Jose Luiz Fiadeiro. Coordination contracts as connectors in component-based development. In *6th Bienneal World Conference on Integrated Design & Process Technology*, June 2002.

[27] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison-Wesley, 2nd edition, 2003.

[28] Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins. Making components contract aware. In *Computer*, volume 32, pages 38–45, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[29] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 2nd edition, 2005.

[30] Jean-Michel Bruel. Service-oriented implementation of component associations. In *Proceedings of the 2004 Canadian Conference on Computer and Software Engineering Education*, number 189–198. University of Calgary, March 2004.

[31] T. Bures and F. Plasil. Communication style driven connector configurations extended version of "scalable element-based connectors". volume 3026 of *LNCS*, 2004.

[32] Yonghao Chen and Betty H. C. Cheng. A semantic foundation for specification matching. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 5, pages 91–109. Cambridge University Press, NY, 2000.

[33] Selim Ciraci and Pim van den Broek. Modeling software evolution using algebraic graph rewriting. In *Workshop on Architecture-Centric Evolution (ACE 2006)*, ACE, Workshop at the 20th European Conference on Object-Oriented Programming ECOOP 2006 July 3-7, 2006, Nantes, France, 2006.

[34] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond.* Addison-Wesley Professional, first edition, 2002.

[35] Philippe Collet. Functional and non-functional contracts support for component-oriented programming (position paper). In David H. Lorenz and Vugranam C. Sreedhar, editors, *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 19–21, Tampa Bay, Florida, 2001. Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115.

[36] CommUnity Workbench 1.4. http://ctp.di.fct.unl.pt/~ co/cw14/index.htm.

[37] E. J. Chikofsky Cross and J. H. Reverse engineering and design recovery: A taxonomy. In *IEEE Software*, volume 7, pages 13–17, 1990.

[38] Ricardo de Mendonça da Silva, Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília Mary F. Rubira. An architectural approach for fault-tolerant component composition based on exception handling. Technical Report IC-04-02, 2004.

[39] Software Evolution Definition. http://www.program-transformation.org/transform/softwareevolution.

[40] K.K. Dhara and G.T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering (ICSE'96)*, pages 258–267, 1996.

[41] Edsger Wybe Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[42] Edsger Wybe Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[43] Hong Duan. A comparative study of pre/postcondition and relational approaches to program development. Master's thesis, McMaster University, 2004.

[44] Stephane Ducasse and Tudor Girba. Modeling software evolution by treating history as a first class entity. In *Workshop on Software Evolution through Transformation, SETRA 2004, with ICGT2004*, volume Electronic Notes in Theoritical Computer Science, Rome, Italy, 2004. ENTCS ELSVIER.

[45] Alexander Egyed, Nikunj R. Mehta, and Nenad Medvidovic. Software connectors and refinement in family architectures. In *IW-SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families*, pages 96–106, London, UK, 2000. Springer-Verlag.

[46] Huw Evans and Peter Dickman. Zones, contracts and absorbing changes: An approach to software evolution. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 415–434, New York, NY, USA, 1999. ACM Press.

[47] Jean-Marie Favre, Reiko Heckel, and Tom Mens. Setra 2006: 3rd workshop on software evolution through transformations: Embracing the change. Natal, Brazil, 2006.

[48] Jose Luiz Fiadeiro. *Categories for Software Engineering.* Springer-Verlag, 1998.

[49] Jose Luiz Fiadeiro and Luis Filipe Andrade. Interconnecting objects via contracts. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 38.*, pages 182–183. IEEE Computer Society, 2001.

[50] Jose Luiz Fiadeiro and Antonia Lopes. Semantics of architectural connectors. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 505–519, London, UK, 1997. Springer-Verlag.

[51] Jose Luiz Fiadeiro and Antonia Lopes. Algebraic semantics of coordination or what is in a signature. In *AMAST '98: Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology*, pages 293–307, London, UK, 1999. Springer-Verlag.

[52] Jose Luiz Fiadeiro and Tom Maibaum. Categorical semantics of parallel program design. In *Sci. Comput. Program.*, volume 28, pages 111–138, Amsterdam, The Netherlands, 1997. Elsevier North-Holland, Inc.

[53] Peter Fingar. Component-based frameworks for e-commerce. In *Commun. ACM*, volume 43, pages 61–67. ACM Press, 2000.

[54] Bernd Fischer and Gregor Snelting. Reuse by contract. In *Proc. ESEC/FSE-Workshop on Foundations of Component-Based Systems*, pages 91–100, 1997.

[55] Robert W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical aspects of computer science*, pages 19–32. American Mathematical Society, 1967.

[56] Robert W. Floyd. The paradigms of programming. *Commun. ACM*, 22(8):455–460, 1979.

[57] Patrice Gahide, Noury Bouraqadi, and Laurence Duchien. Promoting component reuse by integrating aspects and contracts in an architecture model. In *Workshop on Aspects, Components, and Patterns for Infrastructure Software. 1st International Conference on Aspect-Oriented Software Development (AOSD 2002).* University of Twente, Enschede, The Netherlands, April 2002.

[58] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[59] David Garlan. What is style? In *Proc. First International Workshop Software Architecture*, 1995.

[60] David Garlan. Higher-order connectors. In *Proceedings of Workshop on Compositional Software Architectures*, Monterey, California, USA, 1998.

[61] David Garlan. Software architecture: A roadmap. In *ICSE - Future of SE Track*, pages 91–101. ACM Press, 2000.

[62] David Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In Marco Bernardo and Paola Inverardi, editors, *Third International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures, SFM 2003*, pages 1–24, 2003.

[63] David Garlan and A.J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. In *Proc. of the Third Int. Conf. on the Unified Modeling Language*, 2000.

[64] David Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component based systems. In G. T. Leavens Sitaraman and M., editors, *Foundation of Component-Based Systems.* Cambridge University Press, 2000.

[65] John C. Georgas, Eric M. Dashofy, and Richard N. Taylor. Architecture-centric development: a different approach to software engineering. *Crossroads*, 12(4):6–6, 2006.

[66] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[67] J. Gouveia, G. Koutsoukos, M. Wermelinger, L. Andrade, and J.L. Fiadeiro. Coordination contracts for Java applications. In *ICSE 2002. Proceedings of the 24th International Conference on Software Engineering, 2002*, pages 714–714, 2002.

[68] Joao Gouveia, Georgios Koutsoukos, Luis Filipe Andrade, and Jose Luiz Fiadeiro. Tool support for coordination-based software evolution. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, volume 38, pages 184–196, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[69] Joao Gouveia, Georgios Koutsoukos, Michel Wermelinger, Luis Filipe Andrade, and Jose Luiz Fiadeiro. The coordination development environment. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 323–326, London, UK, 2002. Springer-Verlag.

[70] David Gries. *The Science of Programming.* Springer Verlag, New York.

[71] David Gries and Fred B. Schneider. *A logical approach to discrete math.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[72] Reiko Heckel, Tom Mens, and Michel Wermelinger. Workshop on software evolution through transformations: Towards uniform support throughout the software life-cycle. In *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, pages 450–454, London, UK, 2002. Springer-Verlag.

[73] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *OOPSLA/ECOOP '90: Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–180, New York, NY, USA, 1990. ACM Press.

[74] Dan Hirsch, Sebastian Uchitel, and Daniel Yankelevich. Towards a periodic table of connectors. In *COORDINATION '99: Proceedings of the*

Third International Conference on Coordination Languages and Models, page 418, London, UK, 1999. Springer-Verlag.

[75] C. A. R. Hoare. An axiomatic basis for computer programming. In *Commun. ACM*, volume 12, pages 576–580, New York, NY, USA, 1969. ACM Press.

[76] Paola Inverardi and Henry Muccini. Coordination models and software architectures in a unified software development process. In *Coordination Languages and Models: Fourth International Conference, COORDINATION 2000*, pages 323–328. Springer Berlin/Heidelberg, 2000.

[77] James Ivers, Paul Clements, David Garlan, Robert Nord, Bradley Schmerl, and Jaime Rodrigo Oviedo Silva. Documenting component and connector views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Carnegie Mellon University, 2004.

[78] Mehdi Jazayeri. On architectural stability and evolution. In *Ada-Europe '02: Proceedings of the 7th Ada-Europe International Conference on Reliable Software Technologies*, pages 13–23, London, UK, 2002. Springer-Verlag.

[79] Lin Gu Kevin Sullivan and Yuanfang Cai. Non-modularity in aspect-oriented languages: Integration as a crosscutting concern for aspectj. In *AOSD '02: Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 19–26, New York, NY, USA, 2002. ACM Press.

[80] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[81] Jussi Koskinen. Software maintenance costs, http://www.cs.jyu.fi/~koskinen/smcosts.htm. Updated Sept. 28, 2004.

[82] Georgios Koutsoukos, Joao Gouveia, Luis Filipe Andrade, and Jose Luiz Fiadeiro. Managing evolution in telecommunication systems. pages 133–140. Kluwer, B.V., 2001.

[83] Georgios Koutsoukos, T. Kotridis, Luis Filipe Andrade, Jose Luiz Fiadeiro, Joao Gouveia, and Michel Wermelinger. Coordination technologies for business strategy support: A case study in stock trading. In *Proc. of the ECOOP Workshop on Object-Oriented Business Solutions*, pages 41–52, 2001. Invited paper.

[84] Philippe Kruchten. Architectural blueprints—the "4+1" view model of software architecture. In *IEEE Software*, volume 12, pages 42–50, 1995.

[85] Kevin Lano and Jose Luis Fiadeiro. Extending UML with coordination contracts. In *Software and Systems Modeling*, volume 5, pages 110–120, 2006.

[86] Kevin Lano, Jose Luiz Fiadeiro, and Luis Andrade. *Software Design Using Java 2.* Palgrave macmillan, 1st edition, 2002.

[87] M. M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag.

[88] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change.* Academic Press Professional, Inc., 1985.

[89] M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 70–74, New York, NY, USA, 2001. ACM Press.

[90] M.M. Lehman. Software's future: Managing evolution. In *IEEE Software*, volume 15, pages 40–44, 1998.

[91] Barbara H. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-Oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM Press.

[92] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. In *ACM Trans. Program. Lang. Syst.*, volume 16, pages 1811–1841, New York, NY, USA, 1994. ACM Press.

[93] Antonia Lopes, Michel Wermelinger, and Jose Luiz Fiadeiro. Higher-order architectural connectors. In *ACM Trans. Softw. Eng. Methodol.*, volume 12, pages 64–104, New York, NY, USA, 2003. ACM Press.

[94] C. J. P. Lucena and P. S. C. Alencar. A formal description of evolving software systems architectures. In *Sci. Comput. Program.*, volume 24, pages 41–61, Amsterdam, The Netherlands, 1995. Elsevier North-Holland, Inc.

[95] Nenad Medvidovic. *Architecture-Based Specification-Time Software Evolution.* PhD thesis, University of California, Irvine, 1999.

[96] Nenad Medvidovic, Eric Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. In *Information and Software Technology*, volume 49, pages 12–31, 2007.

[97] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling software architectures in the Unified Modeling Language. In *ACM Trans. Softw. Eng. Methodol.*, volume 11, pages 2–57, New York, NY, USA, 2002. ACM Press.

[98] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A language and environment for architecture-based software development and evolution. In *ICSE '99: Proceedings Of the 21st International Conference On Software Engineering*, pages 44–53, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[99] Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor. A type theory for software architectures. Technical Report UCI-ICS-98-14, Dept. of Information and Computer Science, University of California, Irvine, April, 1998.

[100] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Trans. Softw. Eng.*, volume 26, pages 70–93. IEEE Press, 2000.

[101] Nikunj Mehta and Nenad Medvidovic. Understanding software connector compatibilities using a connector taxonomy. In *First Workshop on Software Design and Architecture 2002*, 2002.

[102] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 178–187, New York, NY, USA, 2000. ACM Press.

[103] Tom Mens, Kim Mens, and Roel Wuyts. On the use of declarative meta programming for managing architectural software evolution. In *Proceedings of the ECOOP'2000 Workshop on Object-Oriented Architectural Evolution*, 2000.

[104] Tom Mens and Michel Wermelinger. Formal foundations of software evolution: Workshop report. In *SIGSOFT Softw. Eng. Notes*, volume 26, New York, NY, USA, 2001. ACM Press.

[105] Bertrand Meyer. Applying "design by contract". In *Computer*, volume 25, pages 40–51, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[106] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[107] Tommi Mikkonen. The two dimensions of an architecture. In *A position paper in First Working IFIP Conference on Software Architecture*, San Antonio, Texas, USA, February 22-24 1999.

[108] Richard Mitchell and James McKim. Extending a method of devising software contracts. In *TOOLS '99: Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages*, page 234, Washington, DC, USA, 1999. IEEE Computer Society.

[109] Ana Moreira, Luis Filipe Andrade, and Jose Luiz Fiadeiro. Evolving requirements through coordination contracts. In J. Eder and M.Missikoff, editors, *CAiSE 2003*, volume 2681 of *LNCS*, pages 633–646. Springer-Verlag Berlin Heidelberg, 2003.

[110] Jason E. Robbins Nenad Medvidovic, Peyman Oreizy and Richard N. Taylor. Using Object-Oriented typing to support architectural design in the C2 style. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32, New York, NY, USA, 1996. ACM Press.

[111] Oscar Nierstrasz. Software evolution as the key to productivity. In A. Knapp M. Wirsing Balsamo and S., editors, *Radical Innovations of Software and Systems Engineering in the Future*, volume 2941 of *LNCS*, pages 274–282. Springer-Verlag, 2004.

[112] OBLOG. http://www.oblog.pt/.

[113] OCL. In *http://www.omg.org/technology/documents/modeling_spec_catalog.htm.*

[114] Cristovao Oliveira and Michel Wermelinger. The CommUnity workbench user manual for version 1.4. 2005.

[115] OMG. OMG Unified Modeling Language specification, 2003.

[116] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Software Engineering, 1998. Proceedings of the 1998 (20th) International Conference*, pages 177–186, Kyoto, Japan, 1998. IEEE Computer Society.

[117] Jens Palsberg and Michael I. Schwartzbach. Three discussions on object-oriented typing. In *SIGPLAN OOPS Mess.*, volume 3, pages 31–38, 1992.

[118] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. In *Commun. ACM*, volume 15, pages 1053–1058, New York, NY, USA, 1972. ACM Press.

[119] David Lorge Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd International Conference on Software Engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[120] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[121] John Penix and Perry Alexander. Toward automated component adaptation. In *The Ninth International Conference on Software Engineering and Knowledge Engineering*, pages 535–542. Knowledge Systems Institute, June 1997.

[122] John Penix and Perry Alexander. Efficient specification-based component retrieval. In *Automated Software Engineering*, number 6, pages 139–170, 1999.

[123] John Penix, Phillip Baraona, and Perry Alexander. Classification and retrieval of reusable components using semantic features. In *Knowledge-Based Software Engineering Conference, Proceedings, 10th*, pages 131–138, 1995.

[124] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. In *SIGSOFT Softw. Eng. Notes*, volume 17, pages 40–52, New York, NY, USA, 1992. ACM Press.

[125] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, MA, USA, 1991.

[126] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, Cambridge, MA, USA, 2002.

[127] Kenneth H. Rosen. *Discrete mathematics and its applications (2nd ed.).* McGraw-Hill, Inc., New York, NY, USA, 1991.

[128] Roshanak Roshandel, Andre Van Der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae—a system model and environment for managing architectural evolution. *ACM Trans. Softw. Eng. Methodol.*, 13(2):240–276, 2004.

[129] Roshanak Roshandel and Nenad Medvidovic. Relating software component models. USC Technical Report USC-CSE-2003-504, Center for Systems and Software Engineering, University of Southern California, Los Angeles, CA, March 2003.

[130] Medha Shukla Sarkar, Dorothea Blostein, and James R. Cordy. GXL — a graph transformation language with scoping and graph parameters. In *Proc. TAGT'98 - Theory and Applications of Graph Transformation*, November 1998.

[131] Kamran Sartipi. *Software Architecture Recovery based on Pattern Matching.* PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2003.

[132] H. J. Schneider. A review for specification matching of software components. Feb 1998.

[133] Johann Schumann and Bernd Fischer. NORA/HAMMR: making deduction-based software component retrieval practical. In *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, pages 246–257, Washington, DC, USA, 1997. IEEE Computer Society.

[134] Mary Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical report, Pittsburgh, PA, USA, 1994.

[135] Mary Shaw and Paul Clements. Toward boxology: Preliminary classification of architectural styles. pages 50–54. ACM Press, 1996.

[136] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, Washington, DC, USA, 1996. IEEE Computer Society.

[137] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[138] Katz Shmuel. A superimposition control construct for distributed systems. In *ACM Trans. Program. Lang. Syst.*, volume 15, pages 337–356. ACM Press, 1993.

[139] Anthony J.H. Simons. The theory of classification, part 4: Object types and subtyping. In *Journal of Object Technology*, volume 1, pages 27–35, November-December 2002.

[140] Douglas R. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[141] Ian Sommerville. *Software Engineering.* Addison Wesley, 7th edition, 2004.

[142] Clemens Szyperski. *Component Software — Beyond Object-Oriented Programming.* Addison-Wesley and ACM Press, first edition, 1998.

[143] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., and Jason E. Robbins. A component- and message-based architectural style for GUI software. In *ICSE '95: Proceedings of the 17th International Conference on Software Engineering*, New York, NY, USA, 1995. ACM Press.

[144] Herbert Toth. On theory and practice of assertion based software development. In *Journal of Object Technology*, number 2, pages 109–130. ETH Zurich, Chair of Software Engineering, 2005.

[145] Axel van Lamsweerde. Formal specification: A roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM Press.

[146] Michel Wermelinger and Jose Luiz Fiadeiro. Algebraic software architecture reconfiguration. In *Proc. ESEC/SIGSOFT FSE 1999*, pages 393–409. Springer-Verlag, 1999.

[147] Michel Wermelinger and Jose Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. In *Sci. Comput. Program.*, volume 44, pages 133–155. Elsevier North-Holland, Inc., 2002.

[148] Michel Wermelinger and Cristovao Oliveira. The CommUnity workbench. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 713–713, New York, NY, USA, 2002. ACM Press.

[149] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. In *ACM Trans. Softw. Eng. Methodol.*, volume 6, pages 333–369, New York, NY, USA, 1997. ACM Press.

Index