I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Reuse of software designs, experience and components is essential to making substantial improvements in software productivity, development cost, and quality. However, the many facets of reuse are still rarely used in the various phases of the software development lifecycle because of a lack of adequate theories, processes, and tools to support consistent application of reuse concepts. There is a need for approaches including definitions, models and properties of reuse that would provide explicit guidance to a software development team in applying reuse. In particular there is a need to provide abstractions that clearly separate the various functional concerns addressed in a software system. Separating concerns simplifies the identification of the software components that can benefit from reuse and can provide guidance on how reuse may be applied.

In this thesis we present an extended model related to the separation of concerns in object-oriented design. The model, called *views*, indicates how an object-oriented design can be clearly separated into objects and their corresponding interfaces. In this model objects can be designed so that they are independent of their environment, because adaptation to the environment is the responsibility of the interface or view. The view can be seen as expressing the semantics for the "glue" that joins components or objects together to create a software system. Informal versions of the *views* model have already been successfully applied to operational and commercial software systems. The objective of this thesis is to provide the *views* notion with a theoretical foundation to address reuse and separation of concerns.

After clearly defining the *views* model we show the formal approach to combining the objects, interfaces (views), and their interconnection into a complete software system. The objects and interfaces are defined using an object calculus based on temporal logic, while the interconnections among object and views are specified using category theory. This formal framework provides the mathematical foundation to support the verification of the properties of both the components and the composite software system. We then show how verification can be mechanized by converting the formal version of the *views* model into higher-order logic and using PVS to support mechanical proofs.

Acknowledgements

To my wife, Karin, for her love, patience, and support.

To my children, Amanda and Daniel, for their love and for giving me reasons to persevere.

To my parents, Joaquim and Enilze, for their boundless support throughout my academic life.

To my supervisor, Professor Donald D. Cowan, for his constant encouragement and enduring patience.

To Professor Paulo Alencar for his precious guidance and advice.

To all the members of my dissertation committee for agreeing to serve as examiners.

To Daniel German, for his friendship throughout my graduate studies.

To all my friends at the University, who served as a source of encouragement throughout the years.

Contents

1 Introduction **1**

1.1 Motivation 1

1.1.1 Reuse 2

1.1.2 Object-Oriented Modeling 3

1.1.3 Formal Methods 3

1.2 Problem Statement 5

1.2.1 Expressiveness of Modeling Constructs 5

1.2.2 Representation of Object Models 7

1.3 Proposed Solution 9

1.3.1 Extending the Set of Modeling Constructs 9

1.3.2 Formal Specification of Object Models 11

1.4 Contributions 12

1.5 Related Work 12

1.6 Thesis Overview 17

2 The *Views* Approach in Modeling **18**

2.1 Separation of Concerns 19

2.1.1 The ADV Modeling Approach 21

2.1.2 Applications of the ADV Model 22

2.2 Modeling with *Views* 22

2.2.1 Properties of a *Views* Relationship 25

2.2.2 Reuse 27

2.2.3 Examples of *Views* Usage 28

2.3 UML Relationships 30

2.4 Extending the UML Notation 32

3 A Formal Theory for the *Views* Relationship **34**

3.1 An Introduction to Categories 36

3.2 Object Calculus Theories 41

3.3 An Interpretation Theory for a Class 43

3.4 Combining Object Theories with Morphisms 46

3.5 An Interpretation Theory for a Relationship 49

3.6 Formalization of UML Relationships 52

3.6.1 Association 53

3.6.2 Aggregation 61

3.6.3 Generalization 63

3.7 Properties of the *Views* Relationship 64

3.7.1 Self Dependencies 64

3.7.2 Acyclic Structural Dependencies 65
3.7.3 Cardinality Constraints 66
3.7.4 Creation/Destruction of Objects and Relationship 69
3.7.5 Viewed Singularity and Viewer Multiplicity 72
3.7.6 Viewer and Viewed Visibility 75
3.7.7 Attributes Consistency 76
3.7.8 Action Mappings 80
3.8 Case Study: Dual Interface Clock 82

4 Verification **94**
4.1 Formal Specification of Object Models 95
4.1.1 Semi-Formal Specifications 95
4.1.2 Formal Specification of the *Views*-Based Application 97
4.1.3 Related Work 98
4.2 Using a Verification System 98
4.2.1 The PVS Specification Language 99
4.2.2 The PVS Prover 101
4.3 Object Theories 103
4.3.1 A Generic Class Theory 104
4.3.2 Specification of Object-Oriented Theories 107
4.4 Relationship Theories 114
4.4.1 Formalization of an Association 115
4.4.2 Inheritance and Subtyping 124

4.4.3 *Views* 125

4.4.4 Using *Views* to Relate Object Theories 128

4.5 Colimit Theories 131

4.5.1 A General Colimit Theory 132

4.5.2 The Colimit of the Whole 134

4.6 Proving System Properties 135

4.6.1 Framework Properties 138

4.6.2 *Views*-Related Properties 140

4.6.3 UML-Related Properties 147

4.6.4 Domain-Specific Properties 148

4.7 Other *Views*-Based Systems 152

5 Conclusion **154**

5.1 Summary 154

5.2 Future Work 155

A The PVS Environment **157**

A.1 The PVS Language 157

A.2 PVS Prover Commands 159

A.2.1 Verification Commands 160

A.2.2 Control and Structural Commands 162

Bibliography **163**

List of Tables

3.1 Binary relation conditions 59

3.2 Different forms of aggregation relationship 62

3.3 Conditions on actions of object and relationship instances 72

List of Figures

1.1 Stack models 6

1.2 Yet another stack model 11

2.1 A multi-interface stack model 25

2.2 A simple interfacing model 28

2.3 An interface model for distinct concerns 29

2.4 Metamodel for UML relationships 31

2.5 The extended metamodel for the Core subpackage 32

3.1 Categories **2** and **3** 38

3.2 (a) A cone and (b) a cocone for a diagram 40

3.3 The pullback categorical construct 41

3.4 Morphisms between instances, class manager and class theories . . . 48

3.5 Morphisms forming a composite of two object theories 49

3.6 A general relationship 50

3.7 The colimit of object and relationship theories 52

3.8 A multiple viewers example 74

3.9 *Views* consistencies in a clock application 77

3.10 Vertical consistency through attribute morphisms 79

3.11 A multiple viewers example 81

3.12 Interconnection between clock system theories 83

3.13 Specification of the Viewed Counter object 84

3.14 Specification of the Analog Clock Viewer object 85

3.15 Specification of the Digital Clock Viewer object 88

3.16 Specification of the *Views* relationship 90

3.17 Specification of M_{AR} class manager signature 91

3.18 A morphism between the Analog Viewer and a class manager theory 92

3.19 A morphism between *Views* and a class manager theory 93

4.1 Model of a banking application 95

4.2 A model for the composition of theories 104

Chapter 1

Introduction

1.1 Motivation

The sophistication level of current software systems requires the involvement of people with variable knowledge levels throughout development. Interface design, requirement analysis, specification, software architecture and programming are examples of tasks usually performed by different personnel. Yet, results of each phase of the development process should, ideally, be reliable, understandable and resilient to changes. The attainment of these characteristics, however, is time consuming, expensive, and requires experienced developers.

Reuse is one current approach capable of making substantial improvements in software productivity, development cost, and quality. It contributes to the improvement of the software development process in several ways. Reliability is increased, as reusable assets are usually tested and assessed before available. Understandability and adaptability are enhanced, as working with components provides higher degrees of abstraction. In addition, dealing with software components, rather than

programming code, diminishes the complexity of construction processes and decisions. However, effective reuse requires more than locating assets, assessing their relevance, and adapting them to particular needs. Reuse effectiveness depends on building software assets that are reusable by design.

Despite being recognized as a viable solution for the urgent needs of the software industry, the many facets of reuse are still rarely used in the various phases of the software development lifecycle because of a lack of adequate theories, processes, and tools to support consistent application of reuse concepts. There is a need for approaches including definitions, models and properties that would provide explicit guidance in the application of reuse.

In the following subsections of this motivation, we interrelate three research areas contributing to this thesis: reuse, object-orientation, and formal methods.

1.1.1 Reuse

In an extensive research, Mili, Mili, and Mili state that after decades of research, software reuse seems to be the only realistic approach to achieve a much needed improvement in software quality and production [MMM95]. Reusable components provide levels of abstraction that can be effectively applied in the development of increasingly complex software [Pen93]. However, effective software reuse involves software that is reusable by construction. Higher levels of reuse are obtained with the careful design of reusable architectures, and with the introduction and institutionalization of reuse in the development organization [Wen94].

Software reuse should not be limited to the implementation artifacts. In fact, reuse can be achieved in any stage of the software development and at different levels of abstraction. Reusable elements include architectures, design patterns, domain

models, development processes and decisions, and many other aspects involved with a software system. Our current interests are related to reuse in earlier stages of the software lifecycle.

1.1.2 Object-Oriented Modeling

Since SIMULA was introduced by Dahl and Nygaard [DN70], object-orientation has continuously grown. In the past decade, the growth pace has noticeably intensified and several object-oriented modeling and programming languages have become one major target for software engineering research. Developers view object-orientation as the answer to improve software understandability, quality, and reusability. This perspective is basically supported by the concept of *information hiding*.

However, achieving quality and reuse cannot be guaranteed by a simple switch of development paradigm. In this sense, Wasmund argues that reuse is a software engineering discipline rather than a technology [Was94]. In other words, this author advocates that more than programming paradigms and tools are needed to achieve high degrees of reusability. In addition, some characteristics of object-oriented technologies may introduce the false idea that systems developed according to object-oriented paradigms have the intrinsic property of being reusable. This is a myth that holds only in cases where reuse is explicitly planned as a goal.

1.1.3 Formal Methods

Currently, most uses of formal methods are in safety and security critical systems where formal methods are a possible way to achieve the needed high level of assurance [Rus95a]. The reason being that, despite several attempts to simplify formalization activities, formal specification is still considered a complex and intricate

task. However, as formal methods popularity grows, tools are expected to simplify and integrate to mainstream development methodologies some of the tasks which keep users away from using formal methods.

Meyer [Mey99] argues that the formal methods reputation of being complex is not entirely justified, and formal methods have already achieved a number of successes. He adds that the use of formal techniques may surprise many doubters in the future. One of the reasons for this foreseeable success is its connection with reuse. As Meyer says, *reusable components need strong warranties, and formal-methods costs can be justified economically by the economies of scale permitted by reuse* [Mey99]. Complementary, Biggerstaff and Richter write that the four fundamental processes of reusability are finding, understanding, modifying, and composing components [BR87]. In order to engineer automated support for reusability and provide a high level of assurance, these four processes have to be formalized [BR87, Pen93].

Formalization is also important to the development of the object oriented paradigm. Many authors [LB98b, DH99, BC95, WRC97, EFLR98] work to add rigour to currently popular object-oriented modeling language in an attempt to overcome its limitations. However, the purpose of these attempts are sometimes misunderstood. One of several myths on formal methods [Hal90, BH94] is that formal specifications replace the need for other informal techniques, such as natural language requirements and testing. The reality is that formal specifications are an important complement to other informal or semi-formal techniques. There might be cases when formal methods are in a sense "over-kill", while in other situations they might be crucial. In addition, the formal specification phase of the software lifecycle is aimed at the early detection of errors, when they are less expensive to fix [Boe87].

1.2 Problem Statement

Within current analysis and/or design methods a single type of modeling construct can be used with several different purposes that affect their semantics. More specifically, relationships between classes and/or objects are being applied without a clear representation of its semantical meaning. This situation will inevitably result in concept miscommunications, which may prove costly throughout the development process.

The problem is twofold. First, the variety of purposes and semantics required in the modeling of specific domains is usually not supported by object-oriented methods. Distinct concepts are often represented under a single notational construct. Second, most of the popular modeling languages do not provide precise and unambiguous definitions for the modeling constructs supported. As a result, later interpretations of the model may be incomplete or incorrect.

1.2.1 Expressiveness of Modeling Constructs

In related works [AM94, Civ93, WdJS95], the authors acknowledge the problem of different semantic uses and misuses of widely accepted object-oriented relationships. For instance, specialization, which is often confused with the inheritance implementation mechanism, is considered too flexible and general to encourage a disciplined employment, as discussed by Armstrong and Mitchell [AM94]. These authors also compared inheritance to the *goto* construct, whose lack of expressive power created problems for developers and maintainers.

Specialization is a class relationship in which the behavior of a superclass is shared by all of its subclasses. In a proper specialization, an operation of the subclass that corresponds to an operation in the superclass has the responsibility

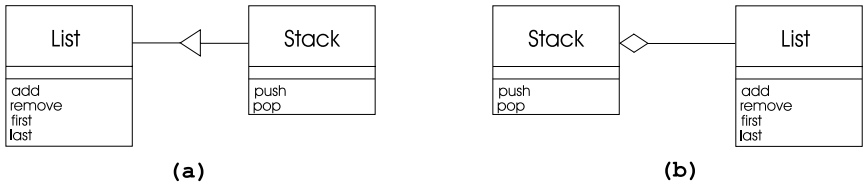


Figure 1.1: Stack models

to provide equivalent services, and possibly more. However, in a typical misuse of this relationship, specialization is used as a technique to reuse behavior partially, even when the related classes are inherently different. This can lead to all sorts of problems, including a deterioration in understandability.

Rumbaugh *et al.* [RBP+91] use the implementation of a *Stack* class based on an already existing *List* class to illustrate one bad usage of inheritance. An object model based on the Object Modeling Technique (OMT) describes such an example in Figure 1.1. The figure shows two models of a *Stack* using specialization and aggregation relationships. In the model of Figure 1.1(a), the *Stack* class inherits both desirable (e.g., adding or removing elements from the top of the list) and undesirable (e.g., adding or removing elements from arbitrary positions in the list) operations. The desirable operations are used, while the undesirable ones are masked or just ignored. This may lead to serious misconceptions and possibly unexpected behavior.

An alternative implementation model, which is shown in Figure 1.1(b), is also proposed by Rumbaugh. In this second model, the two classes involved are instantiated as distinct objects, with *Stack* delegating its behavior to appropriate operations of *List*. This model guarantees that a *push* operation requested to *Stack* is delegated to *List* and properly executed. Undesirable operations of *List* are not accessible through the *Stack* interface.

Despite aggregation providing an apparent reasonable solution, the semantic interpretation of this model goes beyond delegation, which was the original purpose of the relationship construct used. As shown in Figure 1.1(b), the model indicates that a stack object *is composed of* a list, which is not an accurate representation for this particular definition of a list. Composition usually implies dynamic dependencies which may not be convenient for this particular specification of a stack. Understandability and reusability of this model are considerably affected.

We think that a more expressive modeling language would help to remediate circumstances like the one just mentioned. A language that provides an extensive set of abstraction mechanisms offers more options to convey the desired ideas properly. However, the creation of a complex and cumbersome notation would not help developers at all. So, this trade-off situation instigates the analysis of a few questions, such as what are the most important and used concepts that should be supported by a language? How to extend modeling languages with these concepts? What is the significance of these concepts in each development phase? As of today, researchers have not found clear answers to these questions, as different problems have different needs.

1.2.2 Representation of Object Models

Meyer identifies several categories of deficiencies commonly found in informal representations [Mey85]. The author uses these representation pitfalls in the argument that formal specifications constitute an important step between requirements and design phases of the software development regardless of the size of the system. In fact, the author uses a small editor specification to illustrate his points. In a similar fashion, we identify pitfalls in the small semi-formal stack example shown in Figure 1.1.

One of the most common pitfalls identified by Meyer is *ambiguity*, which allows different interpretations for a single model. Ambiguity results from the lack of a complete and precise definition for all the constructs supported by a modeling language. Another common pitfall is called *silence*, which indicates the absence of specification for a feature. For instance, the whole-part relationship in OMT (i.e. aggregation) sometimes implies a lifetime dependency of the *parts* on the *whole*. Other times, however, it does not. The representation of this information is not supported by the OMT graphical notation.

The Unified Modeling Language (UML) is another semi-formal language that attempts to overcome some of the limitations of OMT. However, as most of the semantics of this language is defined in natural language, there are several topics for which the definition is ambiguous. For example, UML defines a *composite* aggregation as a relation in which the deletion/copy of the *whole* implies the deletion/copy of the *parts*. However, the language is not clear whether a *part* may exist prior to the *whole*.

In Figure 1.1(b), instances of the *List* class represent the *parts*, while *wholes* are instances of the *Stack* class. With the information provided in the diagram, very little can be assumed about existence dependencies among the objects involved. While other semi-formal notations, such as UML, may describe models with an improved rigour, only formal languages used carefully are able to assure unambiguity in specifications. This is an essential factor to the development of specification and verification tools.

1.3 Proposed Solution

The object-oriented paradigm has matured for a few decades. However, object-oriented development methods are still relatively recent [Boo91, CY91, CD94, JCJO92, RBP+91, WBWW90]. New or extended methods are still expected to improve the support to software modeling with better abstraction techniques, evaluation mechanisms, and management. In current modeling techniques, we verify that each of the widely used relationships, such as specialization, aggregation, and association, is general enough to cover a few different concepts. Some of these semantic uses may even lead to equivocate interpretations, as previously indicated in the *Stack* example of Figure 1.1. We consider this as an indication that software specification could benefit from additional primitive constructs and concepts which are not supported by most methods yet. The addition of these new modeling elements should extend the basis to represent abstractions and, consequently, improve the quality of specifications.

1.3.1 Extending the Set of Modeling Constructs

While there is a wide consensus that software reuse potentially enables significant software productivity, quality, and cost improvement [Lim94, McC97, Rei97], most software development methods do not include support for reuse [McC97]. Thus, there is a need for explicit definitions about how to practice reuse as part of the development process. These definitions include models and properties of reuse mechanisms that can clarify and provide guidance for the software developers that want to adopt reuse.

In this thesis we deal with a reuse technique called separation of concerns [Aks96] that can be applied in object-oriented design. Separation of concerns is

a well-established principle in software engineering that attempts to hide complexity through abstractions [Par72b, Pre92] that carefully segregate different aspects of a set of related algorithms such as those encompassed in an object. Abstraction is the object-oriented technique adopted by humans to overcome its limited capacity to deal with complexity. As software complexity grows, so does the need for more powerful abstractions.

Specialization through inheritance is an approach that is suggested to achieve separation of concerns. However, inheritance forces all the concerns to be embedded somewhere in the class structure, and unless very carefully used, does not allow the object to be distinguished from all of its related special conditions. In addition, aggregations and associations are two other types of relationship which are not suitable to represent separation of concerns. While aggregations are characterized as strong relationships with tight coupling between the whole and its parts, associations are too flexible and do not provide mechanisms to guarantee consistency among related concerns.

The *views* modeling approach was created to promote a disciplined interconnection of modules representing different concerns [ACLN98a, ACLN98b]. The basic construct of this model is the *views* relationship, which defines the pattern of interaction between objects representing distinct concerns. The *views* relationship provides a framework for interface modeling which is not supported by any other language. It establishes a connection between interface and application that guarantees consistency while allowing a loose coupling between those parts.

Figure 1.2 indicates a stack model based on a *views* relationship. As detailed in the following chapters, such diagram indicates that a stack object is an interface or a *view* for a list object. This interpretation reflects a more accurate meaning for the stack application.

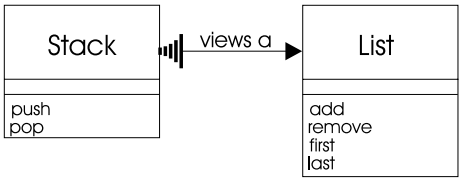


Figure 1.2: Yet another stack model

1.3.2 Formal Specification of Object Models

While Figure 1.2 uses a different relationship for the stack example, it is still prone to ambiguous interpretations. The reason being that this extended UML graphical notation is not formal. Several researchers promote the use of a formal textual notation to complement the information provided by graphical semi-formal notations. While semi-formal notations support the easy communication of concepts to users, formal notations provide the precise specifications which is often important to convey information unambiguously.

We use object calculus theories based on logic in conjunction with a categorical framework to describe the properties of the object constructs supporting our modeling approach. This formal framework was chosen because of the availability of tools to develop and verify formal specifications, such as logic-based environments. In addition, these formalisms are not dependent on any particular specification language. Thus, other formal specification languages or similar approaches can benefit from the theories presented in this thesis.

1.4 Contributions

The thesis of this work is that augmenting the accuracy of abstractions supported by extended modeling methods allows the definition of a formal framework that supports the specification and verification of properties characterizing object-oriented relationship constructs.

As part of the process of attaining the above described goal, a number of contributions can be identified from this thesis. These contributions are succinctly described as follows:

- definition of an object-oriented modeling approach that promotes a disciplined interconnection of modules representing different concerns in a software system;
- extension of the core concepts and notation of a modeling language to support the representation of the *views*-related constructs;
- description and use of a formal framework in the specification of relationship properties in object models; and
- use of a formal verification environment based on logic to mechanize the validation of different types of properties used in object system specifications.

1.5 Related Work

With the concepts of information hiding [Par72b] and the notion of module specification [Par72a], Parnas introduced some cornerstones of modern software design. In a sense, the work of Parnas established the roots of our current work. Other

early precursors of our ideas were De Remer and Kron [DK76], which defined a module interconnection language to support programming-in-the-large.

In the past few years, a number of architectural models and programming approaches have investigated the support to the separation of different concerns in distinct specification modules in order to achieve higher degrees of reuse. Goguen investigated the general interface concept together with the reuse and interconnection of software components [Gog86]. Similar to the *views* approach, he uses formal languages and mappings of types and operations to interconnect and maintain consistency among objects. He also used category theory to put object theories together [Gog89]. However, Goguen does not define a relationship theory among object specifications, thus making the properties of his design mechanism quite different from our approach. The Common Object Request Broker Architecture (CORBA), which is supported by the Object Management Group, defined an open standard for application interoperability [Dig91]. This standard is based on a client/server interaction model that separates application interfaces from their implementations. These interfaces are specified in a neutral Interface Definition Language. More recently, Kiczales *et al.* [KLM+97] described a new programming paradigm called Aspect-Oriented Programming (AOP). AOP provides the basis for the identification, isolation, composition and reuse of the several concerns, which are known as *aspects*, contained in the programs.

The MVC model [KP88] was one of the first implementations to address separation of concern issues specifically. Currently, several visual development environments [IBM94, Dig92, Syb96] simplify the programming task by making available a library of reusable interface (visual) and application specific (non-visual) objects. The interface objects are interconnected to the application by mechanisms which are specific to the particular programming paradigm supported by the environment.

In addition, the separation of concerns allowed by these mechanisms is mostly directed at the user interface part of the system.

Currently, the Java programming language represents one of the popular paradigms to the development of user interfaces. The Java interface mechanism is event-driven. The event handling model of Java 1.1 is based on the concept of an *event listener*. An object interested in receiving certain events is called an *event listener*, while the one generating events is called an *event source*. This event source object keeps a list of all the listener objects interested in being notified when certain events occur. Such a concept may be very useful in the implementation of a mechanism that maintains the consistency between interface objects and their respective applications. In addition, the AWT class library of Java provides an implementation of the *Observer* design pattern [GHJV95], which describes a mechanism for maintaining the consistency between interface and application objects.

Modeling of user interface concepts has been one of the research topics addressing the need for additional and rigorous modeling elements. Other researchers, however, try to improve the expressiveness of modeling languages in other ways. Civello separates roles and meanings of *whole-part* associations in distinct constructs [Civ93]. He argues that the resulting models are easier to understand and maintain with the additional semantics represented. In another attempt to add semantics to modeling elements, Steyaert *et al.* [SLMD96] define *reuse contracts* based on specialization. These contracts document the way an asset is related to its superclass, thus allowing a better understanding of the circumstances in which an object is specialized.

In yet another attempt to add rigour to modeling languages, Snoeck and Dedene formally define a new relationship called *existence dependency* [SD98]. Such a relationship captures some of the semantics usually associated with aggregation

relationships. While the authors argue that the semantics of *whole-part* relations are usually insufficiently defined, the new relationship is simple, unambiguous, and enables checking for semantic integrity and consistency between structural and behavioral aspects of object models.

While some authors add new elements to modeling languages, other researchers focus on the precise definition of the semantics of already popular modeling concepts. Bourdeau and Cheng describe a method for deriving algebraic specifications directly from diagrams defined with the object model notation of OMT [BC95]. Wang *et al.* present a formal model for both the object and dynamic models of OMT [WRC97]. They also integrate the two models, which enables to check for inter- and intra-model consistency.

UML is emerging as a de-facto standard for object-oriented modeling [Par97]. As a result, one current research focus is on the specification of a more rigorous semantics of its notations. This is, however, a very difficult task, as the UML language is very large. Evans *et al.* use Z [Dil90] to define the abstract syntax of a subset of the UML static model notation formally [EFLR98]. In another related work, Lano and Bicarregui describe part of structural and dynamic notations of UML using temporal logic theories [LB98b, LB98a].

Views-based mechanisms represent an extension to typical modeling languages. In addition, *views* properties can also be customized and extended with properties suitable to particular applications. As UML supports the extension of the core concepts of the language, it is being used as the basis for the *views* approach development. Thus, similar to the formalization provided by Lano and Bicarregui [LB98b, LB98a], we also formalize some of the UML relationship constructs. This formalization allows the subsequent verification of properties based on UML concepts.

The adopted formal system for the *views* approach includes the use of a categorical framework together with object calculus theories based on logic. This framework is based on an approach developed by Fiadeiro and Maibaum [FM91, FM92], which use a framework based on temporal logic [MP91, Bar87]. While object calculus theories model the object theories of the system, the interconnection of the components is achieved with category theory.

The work of Bicarregui, Lano and Maibaum [BLM97a, BLM97b] also inspired the formalism adopted for the *views* constructs. Based on the formal framework structured by Fiadeiro and Maibaum [FM91], these authors defined interpretation theories for object instances, classes, and associations according to the concepts of an object-oriented methodology called Syntropy [CD94]. The formal notion of association relationships and its properties provided the foundation for the specification of the properties of *views* relationships.

DeLoach and Hartrum use an object-oriented algebraic specification language called O-SLANG to specify theories for the concepts introduced by OMT [DH99]. In such work, the authors specify association, aggregation, and inheritance theories using algebraic specifications that connect class theories. This approach also uses category theory to map elements of the theories, as it was also inspired by the work of Goguen. Note that the banking system case study formally described in Chapter 4 using the PVS specification language was initially based on the algebraic specification presented by the aforementioned authors [DH99].

While the formalism introduced in Chapter 3 of this thesis was based on the object calculus described by Fiadeiro and Maibaum, the availability of a specification and verification environment compelled us to translate the temporal logic based formulae into the higher-order logic language of PVS. In fact, an identical problem motivated the work of Maharaj and Bicarregui [MB97]. These authors describe the

translation of a VDM-SL specification into the PVS language using the methods described by Agerholm [Age96].

1.6 Thesis Overview

In this thesis we develop of a formal framework to support a precise specification and extension of object-oriented modeling languages. Among several modeling constructs being explored, the interface interconnection mechanism called *views* is given particular attention throughout this thesis. The properties associated with this mechanism are formally defined and verified to support reuse by means of a disciplined separation of the distinct concerns contained in a software system.

In Chapter 2, the basic concepts of the *views* approach of modeling are introduced. The set of relationships supported by UML is extended to support the representation of *views* relationship constructs, which represent the basic concept of the *views* approach.

Chapter 3 introduces formal interpretation theories for objects and relationships. The formalization of the set of relationships supported by UML extended with *views* is the goal of the chapter. Particular focus is on the *views* relationship construct.

Chapter 4 focuses on the mechanization of a logic-based system that uses formal concepts introduced in Chapter 3. The specification is developed in a formal environment, which provides typechecking tools for a higher-order logic specification language and a powerful proof checker that allows the verification of properties of the formal specifications. A number of *views* and UML relationship properties are formally stated and verified in this chapter.

In Chapter 5, the major contributions of this thesis are discussed, and topics for future related research are suggested.

Chapter 2

The *Views* Approach in Modeling

Despite a recent effort to create a standard notation for object-oriented modeling, several other notations are still commonly used in the specification of object systems. These notations, however, do not agree on a basic set of concepts, including relationships. Some of the most popular types of relationship supported by object-oriented modeling languages are known as *specialization*[1], *association*, and *aggregation*. These relationships are part of methods such as OMT [Rum88], Object-Oriented Design [Boo91], Syntropy [CD94], and the relatively recent Unified Modeling Language (UML).[2]

While these previously mentioned relationships are capable of representing most object-oriented systems, they may not be expressive in particular situations. For instance, note that the specification of interconnections between loosely coupled modules in a software model is not explicitly addressed by any of these modeling relationships. While association does not have the expressive power to characterize particular types of relationships explicitly, specialization is a refinement relation-

[1]This construct is also referenced as generalization.

[2]UML was accepted by OMG as the standard notation for software architecture.

ship between classes and aggregation interconnects objects which are often tightly related.

Separating objects with different concerns has been the focus of a number of authors [Aks96, HL95, CL95]. Some of them propose implementation models that separate user interface concerns from application-specific objects [KP88]. Other authors investigate pattern mechanisms of interaction among modules implementing different concerns [GHJV95]. Alternatively, our contribution is focused on a methodical study of the semantic properties of a modeling relationship that allows the separation of objects of different concerns in a software model.

2.1 Separation of Concerns

Separation of concerns is a well-established principle in software engineering that hides complexity by means of abstraction mechanisms [FSJ99]. According to the Oxford English Dictionary, a concern is *a relation of connection or active interest in an act or affair.* Alternatively, Czarnecki *et al.* defines a concern as a domain used as a decomposition criterion for a system or another domain with that concern [CES97].

The term *concern* has different meanings across software engineering. In some of its connotations, a concern may refer to elements of design that cross-cut the basic functionality of the system. For instance, memory access patterns may be considered in some cases as one specific concern [KLM+97]. Other notions of concern might be related to more general concepts such as performance and quality. However, in this thesis we will be using the same connotation of a concern as the one given by Fayad *et al.* [FSJ99]. These authors mention that *current frameworks involve a basic concern and a number of special-purpose concerns. The basic concern*

is represented by [...] algorithms that provide the essential functionality relevant to an application domain, and the special purpose concerns relate to other software issues, such as user interface presentation, control, timing, synchronization, distribution, and fault tolerance.

Different concerns can be identified during analysis, design, implementation, and refactoring. Objects with similar concerns are connected by a common interest on a particular domain of the problem description, which may be of structural, functional, or behavioral nature. Distinct concerns should be loosely coupled and as orthogonal as possible. While there are guidelines for the distinction of concerns, the identification of the boundaries of a concern is still an arbitrary task. As Dijkstra states: *The crucial choice is, of course, what aspects to study in isolation, how to disentangle the original amorphous knot of obligations, constraints and goals into a set of concerns that admit a reasonably effective separation* [Dij76].

A significant barrier to the reuse of both designs and implementations of software objects and modules is the fact that they internalize knowledge about their surrounding environment. For example, a typical module or object of an application often knows about its user interface, specifically details of how its data structures will be displayed, how the user will interact with the application, or what objects on the screen correspond to activations of components of the module. Similarly, a module or object knows too much about the services required from other objects or modules. For example, a module will know too much about naming conventions in a file system, or about the names of modules or functions from which it acquires services. Such depth of specialized knowledge seems counter not only to reuse but to good engineering practice in general [CL95].

One of the first models to address the separation of concerns in object-oriented technology was the *Model-View-Controller* (MVC) [KP88]. This is a programming

paradigm which was originally developed for use in Smalltalk-80 systems. The decomposition of interface behavior implemented by this model has also inspired many other models [BC91, Dig91, Hil92, Mye91]. Even though these are excellent implementation models, it is often difficult to map these strategies into other programming environments, or to make them applicable to other software development phases.

2.1.1 The ADV Modeling Approach

The *Abstract Data View*[3] approach [CILS93a, CL95, ACLN95] was developed in an attempt to overcome the limitations inherent in the separation of concern models which are based on specific programming paradigms. This approach is an object-oriented design model which *bridges the gap between the internal world of application objects and its requirement for knowledge of the external world* [CL95]. The basic constructs of the ADV approach are the *Abstract Data View* (ADV) and the *Abstract Data Object* (ADO), which represent, respectively, interface objects (views and interactions) and application objects which are independent of the interface. These types of object support a disciplined approach to design which attempts to separate concerns.

The separation of concerns introduced by the ADV approach divides the "world" into two types of objects. These types are the ADVs and ADOs, and they characterize the concern of an object in a software model as either interface or application. Although we can find many structural similarities in both object concepts, it is important to observe that there is a clear separation between capabilities of ADOs and ADVs. An ADO has no knowledge of its surrounding environment (ADVs), thus

[3]The modeling approach was later renamed to *Abstract Design View*

ensuring independence of the application from its interface. On the other hand, an ADV does know about its associated ADO and can query or modify its state by means of its public interface or a mapping between the ADV and related ADO [ACLN95].

2.1.2 Applications of the ADV Model

Initially ADVs were used to capture the user interface concern of interactive software systems. Later, the model was extended to general interfaces that could capture other external concerns such as a timer or a network. A further extension captured other special purpose concerns such as control, timing, and distribution.

ADVs have been used in various software system designs to support user interfaces for games and a graph editor [CBI+92], to interconnect modules in a user interface design system (UIDS) [LCP92], to support concurrency in a cooperative drawing tool, to design and implement both a ray-tracer in a distributed environment [PLC93], and to design a scientific visualization system for the Riemann problem. A research prototype of the VX·REXX [Wat93] system was motivated by the idea of composing applications in the ADV/ADO style. In addition, we have shown in [CILS93b] and [CLV93] how ADVs can be used to compose complex applications from simpler ones in a style which is similar to some approaches to component-oriented software development and megaprogramming [WWC92].

2.2 Modeling with *Views*

In this thesis we present a different approach for the separation of concerns which was inspired by the ADV model. This new approach, which is called *views*, is

centered on an object-oriented modeling relationship which is identified by the same name. The *views* approach attempts to address most of the important contributions of the ADV model. While it promotes a disciplined interconnection of objects with distinct concerns, it does not characterize the "worlds" of concern according to different types of objects. This means that objects representing one kind of concern are conceptually identical to the objects representing other concerns. Alternatively, the ADV approach uses ADOs and ADVs to characterize the type of concern of a given object.

In the *views* modeling approach, the disciplined separation of concerns is supported by the properties of the interconnecting relationship theory. This theory is external to the objects being interconnected and does not directly change their specifications. This approach is closer to concepts used in currently popular modeling languages, and we believe it makes a smoother transition from these traditional languages to the *views*-based ones. In fact, a *views*-based modeling language may be developed as an extension of some of the current languages in an attempt to improve their expressiveness.

As already mentioned, *views* aims the interconnection of modules representing different concerns [ACLN98a, ACLN98b]. The basic construct of this model is the *views* relationship, which defines the pattern of interaction among objects of different concerns. These objects are usually referenced as *viewer* and *viewed* objects, depending on the roles they perform in the relationship. Viewer roles are often assigned to objects characterizing an interface part of the model. Alternatively, viewed roles are usually performed by objects with domain specific concerns.

Note, however, that the "viewer" and "viewed" terms do not characterize intrinsic properties of the objects. Rather, they represent roles performed by an object in one particular *views* relationship. Thus, an object could be assigned a viewer

role with respect to one relationship at the same time it performs a viewed role for another *views* relationship. Nonetheless, constraints on the number of *views*-related roles performed by a single object exist. For instance, an object cannot perform more than one viewer role in the whole system.

A disciplined separation of responsibilities should lead to wide and consistent reuse of specifications for both viewer and viewed components. However, promoting reuse is not the only major issue related to separation of concerns. Modifications on software often have dangerous and costly consequences on quality, coherence or knowledge of entities that depend on a modified entity [BR94]. Separating concerns into independent modules decreases the danger of such changes.

A viewer object is conceived to represent either a user interface or an adaptation of the public interface of a viewed object that modifies the way this object is accessed by others. Thus, a viewer should have elements inside its specification to access and monitor the current viewed properties. These elements are connected to the viewed object by means of a mapping mechanism. Such a mechanism associates actions and attributes of the viewer object with the corresponding actions and attributes in the viewed object. Mapping is used here only as a modeling concept. It may be implemented in several different ways.

Note that, by construction, a viewer object is aware of the public interface of the viewed object, and so preserves encapsulation. The viewed object, however, should have no knowledge about any internal property of a viewer object. Moreover, the state of these two objects should be consistent at all times by means of the mapping mechanism. Thus, a change in the state of a viewed object will effect corresponding changes in the state of viewer objects related to that object.

The separation of concerns provided by the *views* approach allows the specification of several viewers for a single viewed object. For example, the stack model

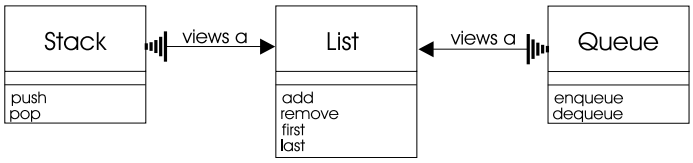


Figure 2.1: A multi-interface stack model

illustrated in Figure 1.2 is being accessed by a single viewer, which is the *Stack* class. An extension of this model could add another interface to the data repository represented by the *List* class. For instance, Figure 2.1 illustrates a model in which the data contained in *List* objects are accessed either through *Stack* or *Queue* interfaces. The state of related instances of all three classes will be kept consistent by means of the *views* relationship properties.

2.2.1 Properties of a *Views* Relationship

While the process of separating concerns still depends on arbitrary decisions of the developer, *views* provide mechanisms and directions which disciplines modeling. In this section, we focus on the characteristics of the relationship construct on which the modeling approach is based.

The term *relationship* will be used throughout this thesis as a theory which interconnects classes. Interconnections among objects will be explicitly referred to as *relationship instances*. An instance of a *views* relationship is a binary construct that associates exactly one viewer object with one other object instance that plays a viewed role in this relationship instance. A relationship, however, may have several instances interconnecting several viewer and viewed objects.

We start the characterization of the relationship with constraints on the identity and type of the objects being interconnected. Such constraints define *views* as

a relationship between different objects of different classes. The rationale for this property is that the construct was conceived to model the interaction among distinct concerns, and distinct concerns are invariably represented by distinct classes. Additional reasons for the preceding characteristic are presented in the next chapter when the property is formally stated.

The purpose of a viewer object is to observe and, if necessary, change the state of a viewed object. A viewer should always have its state consistent with the object it views. Consequently, a viewer depends on the existence of another viewed object to perform its responsibilities effectively. This dependency leads to a property constraining the lifetime of viewer objects. Such property states that an object playing a viewer role in a *views* relationship is always *views*-related to another object playing a viewed role. In other words, the lifetime of a viewer will be contained in the lifetime of the viewed object.

This previous property also affects the possibility of creating *views*-cycles in a model. The lifetime dependency between viewer and viewed objects would have to be respected by every object in the cycle. As a consequence, all of the objects in this cycle would have to be created and destroyed at the same time, thus creating a circular prerequisite problem. Therefore, *views*-cycles are not allowed in a model.

The cardinality of a *views* relationship is another important property defined by this approach. As mentioned in the previous section, the *views* relationship allows the association of zero, one, or many viewer objects of one particular class for each viewed object instance. Conversely, we define that for each object instance playing a viewer role in a views relationship there is exactly one related object playing a viewed role. Such constraints eliminates potential consistency problems of one viewer monitoring multiple objects states. In addition, this rule does not impose any limitation to the modeling process, as a set of viewers may always be composed

into more complex views.

The property that constrains the number of objects related to a viewer object to one is also extended for the system as a whole, and not only for the relationships independently. This means that an object will be allowed to play a viewer role in at most one *views* relationship in the whole system. This same object, however, may play as many viewed roles as allowed by the system specification.

All of the properties described in this section will be formally stated in the next chapter by temporal logic axioms. In addition, theorems derived from those axioms will complement the set of properties.

2.2.2 Reuse

Improvements of an order of magnitude are still needed to extricate the software industry from the well-known *software crisis* [HSL91]. Boehm argues that instead of finding ways of writing code faster, we need to write less hand-crafted software [Boe87]. While *automatic programming* is still a promise, *reuse* appears to be the only realistic solution for software quality and productivity improvements. Wasmund characterizes reuse as a software engineering discipline rather than a technology [Was94]. As stated by the author, *the object-oriented technology does not automatically yield high reuse rates, but it is an enabling platform for high degrees of reuse if explicitly planned for and appropriate actions taken* [Was94].

In particular, the *views* approach is related to reuse because it can be used to separate the user interface from the application part of a system or act as an interface for distinct modules of a system. In both cases, reusability is a consequence of the separation of concerns developed in the structure of the system. By preventing modules of the application from knowing about the surrounding environment, the approach eliminates a drawback for reuse.

2.2.3 Examples of *Views* Usage

The *views*-based modeling of a general interface may take several forms. Each of these forms relies on the properties of the *views* relationship construct and describe a pattern of interaction among different parts of the system. The most intuitive of the interfacing models characterizes the interaction between the user interface and domain-specific parts of a system. However, a set of *views* constructs can also be used to model the interaction among two or more different modules of the system. These interaction mechanisms will be illustrated in this section.

As mentioned in previous sections, one of the barriers for reuse is that objects of the application domain sometimes internalize environment knowledge. In many cases, separation of concerns can be achieved with mechanisms as simple as a single object-oriented relationship construct. The basic idea is to interconnect interface and application, in such a way that the application object can be "viewed" or "monitored" by different objects without ever knowing about the identity or the internal structure of these objects.

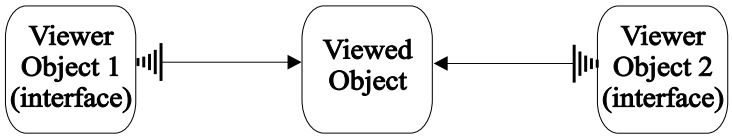


Figure 2.2: A simple interfacing model

One simple interfacing example is illustrated in Figure 2.2, where the *viewers* represent two distinct user interfaces, a *viewed* object represents the domain-specific part of the application, and the arrow indicates the existing *views* relationship between the objects. Note that the represented mechanism describes modeling concepts, and it does not indicate any direction for the implementation phase of the software lifecycle.

Another typical application of *views* relationships is used to characterize the interaction among distinct modules of the system. In such models, both source and target modules in the interaction are described by viewed objects. Alternatively, viewers represent all the interface properties among the interacting modules. The communication flow between the two interacting modules in this model is bi-directional, even though these (viewed) objects do not need a direct reference for each other. All the required information about the target object is hidden by the viewer objects in the interface. This *views*-based model is represented in Figure 2.3.

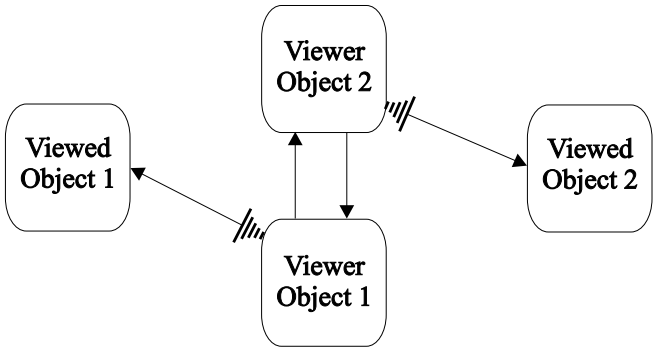


Figure 2.3: An interface model for distinct concerns

Similarly to examples described for the ADV theory [CL95], the model in Figure 2.3 may be used for several interfacing purposes, including domain transformations and synchronization. For instance, in a case where *Viewed Object 2* corresponds to a double-ended queue while *Viewed Object 1* is an object that requires interaction with a stack structure, the viewer objects should act as an interface which transforms an existing domain (i.e. the double-ended queue) into the required domain (i.e. the stack). In such case, the *views* approach relates to reuse by adapting an existing domain into a required one.

2.3 UML Relationships

UML is currently the most popular language for specifying the artifacts of a software system. The language integrates a number of notations and guidelines, and it represents a standard for object-oriented modeling. UML is independent of programming languages and development processes.

Currently, the semantics of UML are mostly defined as a combination of class diagrams and natural language. No formal specification of the UML semantics has been derived yet, as the UML authors claim that such formalization would add significant complexity without a clear benefit. However, we argue that formal methods could help to identify and represent some unclear concepts we found in the current UML specification. Some of these unclear concepts are mentioned in the following chapter. In addition, some researchers [Mey85, DBH95] consider formal specifications as an important complement, rather than a replacement, to informal or semi-formal specifications.

Despite providing extensive documentation on the semantics of UML, the current definition of the language concepts still allows ambiguous interpretations, as will be shown in the next chapter. Thus, we believe that UML may benefit from the use of formal languages by means of precise and concise definitions of modeling concepts. However, the complete UML is too large, and its formalization is not the major objective of this thesis. The metamodel describing the semantics of the language contains approximately 90 metaclasses and 100 meta-associations divided in several logical packages. Therefore, we will limit our focus to some of the concepts which are more closely related to the properties of the *views* approach.

The *views* modeling concepts are basically centered on a relationship construct. Thus, in accordance to the *views* approach goals, our main interest in UML is on

the characteristics of the relationships constructs supported. These relationships are described in the *Core* subpackage. *Core* is considered the most fundamental of the subpackages composing the *Foundation* package, which provides the infrastructure for UML [Rat97]. The Core subpackage defines the basic constructs for the development of structural models. The semi-formal metamodel describing the semantics of this subpackage is shown in Figure 2.4.

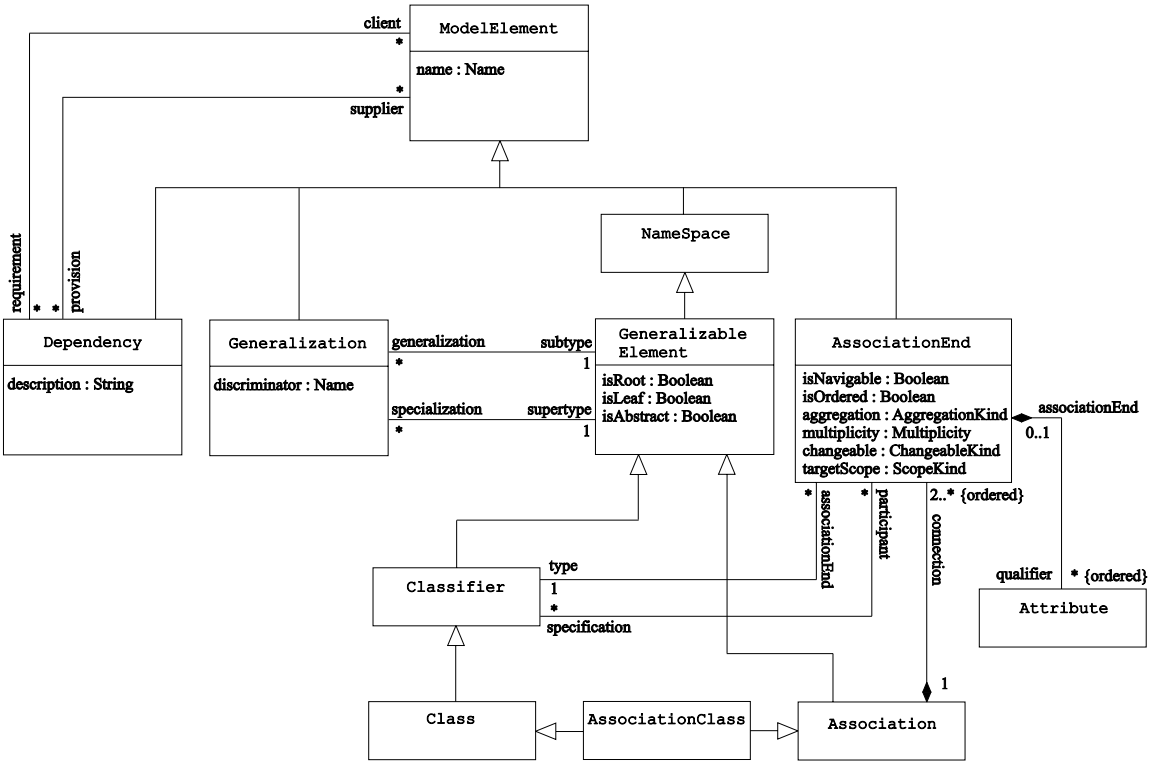


Figure 2.4: Metamodel for UML relationships

Despite some of the metaclasses in the metamodel under consideration not being explicitly considered in our formalization effort, the most important concepts defined in the Core subpackage will be rigorously treated in the following chapters. Most notably, the metaclasses to be formally investigated include *Class*, *Generalization*, *Association*, *AssociationClass*, and *AssociationEnd*. Note that the attributes

in this latter metaclass represent the semantics for every association relationship in UML models (aggregations included). The only other UML relationship construct is called generalization.

2.4 Extending the UML Notation

One of the goals of UML is to support the extensibility and specialization mechanisms to extend the core concepts of the language. In fact, UML authors expect that UML will be tailored as new needs are discovered. This perception represents a nice fit for the objectives of the *views* approach, which aims to augment the expressiveness of an already existing core of modeling concepts.

To support the extension of UML with the *views* construct, we create a graphical representation that is compatible with the other graphical elements of the UML notations. This relationship representation was already illustrated by the arrows in Figure 2.2. The tail of the arrows are connected to classes/objects performing the viewer role in the relationships. Alternatively, the head of the arrows are connected to a class/object performing a viewed role.

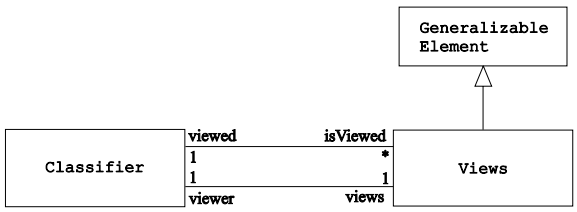


Figure 2.5: The extended metamodel for the Core subpackage

The extension of the UML Core subpackage metamodel resulting from the addition of *views* relationship constructs to the modeling language is shown in Fig-

ure 2.5. This metamodel defines two associations between the *Classifier*[4] and the *Views* metaclasses: one for the viewed and another for the viewer objects. The former indicates that one viewed object may be "monitored" by several viewers. The second meta-association indicates that for each viewer instance there is exactly one corresponding *views* relationship instance. Note that this meta-association does not limit the number of *views* relationship (or viewer) instances.

The extended structural notation for UML models will be used throughout this thesis in conjunction with the other graphical constructs supported by the modeling language.

[4]A classifier is an element describing behavioral and structural features. Common classifiers are classes, data types, and interfaces

Chapter 3

A Formal Theory for the *Views* Relationship

In this chapter we introduce formal interpretation theories for the *views* relationship, which is modeled between a *viewer* and a *viewed* object [ACN98], and for some UML constructs. We also introduce interpretation theories for the objects being interconnected by the relationship. Each of these theories may be seen as a set of statements we want to make about something. However, as most theories contain too many facts for them to be explicitly listed, in practice, only a subset of these statements are described by means of axioms. The remainder of the facts may be inferred using a deductive system. While the set of all statements is normally referred to as *theory*, the specification with the explicitly listed axioms is termed *theory presentation*.

The objects and the *views* relationship are presented as smaller and separate theories which should be combined to form a composite (system) theory. As argued by Burstall and Goguen [BG77], the structure of a specification is more connected to the way a specification as a theory can be expressed as a combination of smaller,

more tractable theories. Therefore, the purpose of a specification language is to provide tools for putting small theories together to make larger specifications. Category theory is used to combine object and relationship theories in the same way that *given a category of widgets, the operation of putting a system of widgets together to form some super-widget corresponds to taking the co-limit of the diagram of widgets that shows how to interconnect them* [Gog89].

The adopted formal system includes the use of a categorical framework together with object calculus theories based on logic. The categorical framework illustrates how the characteristics of interface objects and their relationship with other objects in the system can be combined into a logic-based formalism. The object calculus theories model the components of the system in terms of signatures and logic axioms. This framework was chosen as the formal underlying description because of the availability of tools to develop and analyze formal specifications, such as logic-based environments. These formalisms are not dependent on any particular specification language. Thus, other formal specification languages or similar approaches can include the theory of interfaces of objects.

Following an approach developed by Fiadeiro and Maibaum [FM91, FM92], we use a categorical framework based on temporal logic [MP91, Bar87] to define theories for the system objects which are involved in a relationship. The relationship itself is another component of the system which is described as a separate theory that interconnects two objects. This theory is first introduced in general terms, and later refined with the addition of new axioms which characterize the *views* relationship in particular. While object calculus theories model the components of a system, the whole system (or a composite component) results from the interconnection of the components by means of morphisms. These interconnected components form a category.

In the first few sections of this chapter we outline the principles of the formal framework adopted. Nevertheless, we limit our discussion of this framework to the basic principles required to support our theory of objects and interfaces. The latter sections of the chapter characterize the *views* approach and illustrate the related concepts with a case study.

3.1 An Introduction to Categories

Before we go any further with the description of object theories and how they are interconnected, we introduce a few basic concepts necessary to the understanding of the framework adopted in the *views* formalization process. While the following sections introduce the object calculus theory for object and relationship specifications, this section is intended to present concepts that give us the ability to define interconnections among a number of theories, and reason about the combination of these theories as a single composite theory (or the whole system). These concepts are defined by the *category theory*.

Category theory is an abstract mathematical theory used to describe the external structure of various mathematical systems [Sri90]. Our use of the category theory is to define interconnections among theory presentations. A category [BW90] is a graph with rules for composing arrows and nodes in order to generate another composite node. Moreover, this composite node may be used as a component node of a yet more complex system. In our specification framework, the *arrows* in a category represent *morphisms*, while *nodes* represent the *object theories*.

The rules for object composition are given by four functions. But before we describe these functions, we present a few definitions which are used in the specification of the properties of these four functions.

Definition 1 *The* **domain** *of a function f is the set of all elements which are mapped to something by f. If f is defined by $f : S \to T$ its domain is represented by the set S.*

Definition 2 *The* **codomain** *of a function f is the set of all elements which have something mapped to them by f. If f is defined by $f : S \to T$ its codomain is represented by the set T.*

The first two functions of a category specification associate with each morphism f of the category its domain $dom(f)$ and codomain $cod(f)$, both of which are objects of the category. The expression $f : S \longrightarrow T$ is used to indicate that f is a morphism with domain S and codomain T. The collection of all arrows with domain S and codomain T in a category $\mathbf{C}$ is written $\mathbf{C}(S, T)$.

Definition 3 *If $f : S \to T$ and $g : T \to U$ are two functions, then the* **composite** *function $g \circ f : S \to U$ is defined to be the unique function with domain S and codomain U for which $(g \circ f)(x) = g(f(x))$ for all $x \in S$.*

Definition 4 *We call* **identity** *the function $id : S \to S$ for which $id(x) = x$ for all $x \in S$.*

The next two functions associate with each object C of a category a morphism $\mathbf{id_C}$ called the **identity morphism** and a function of composition. This composition function associates another morphism $f \circ g$ with any pair (f, g) of morphisms such that $dom(f) = cod(g)$. These four functions are required to satisfy the following axioms:

$dom(f \circ g) = dom(g)$

$cod(f \circ g) = cod(f)$

$(f \circ g) \circ h = f \circ (g \circ h)$ (associative law)

$dom(id_A) = cod(id_A)$

$id_A \circ f = f$ (identity law)

$f \circ id_A = f$ (identity law)

Pierce describes a number of illustrative examples of categories [Pie91]. Category **0** has no nodes and no arrows. The identity and associativity laws are trivially satisfied. Category **1** has one node and the identity arrow. The composition of this arrow with itself can only be itself, thus satisfying the category axioms. Category **1+1** is composed of two nodes and two identity arrows. Category **2** has two nodes, two identity arrows, and another arrow connecting the nodes, as shown in Figure 3.1. It is easy to verify that the six category axioms are satisfied. Category **3** is the other example shown in Figure 3.1, where f, g, and h are the only non-identity arrows in that category.

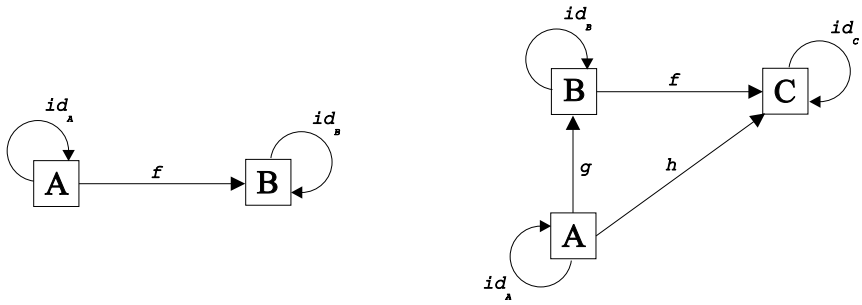


Figure 3.1: Categories **2** and **3**

In the previous paragraphs we characterized a category. However, another important use of category theory in our formalism is in the representation of the

composition of a number of theories into a more complex theory. In this respect, pushouts and colimits are two concepts of the category theory which will be of significant importance in this chapter. But, before we describe these concepts, a few other definitions need to be introduced.

According to Pierce, a **diagram** in a category **C** is a collection of vertices and directed edges consistently labeled with nodes and arrows of **C** [Pie91]. In other words, if an arrow f has domain A and codomain B, the edge in the diagram must be labeled as f and its endpoints labeled as A and B.

Definition 5 *A diagram in a category* **C** *is said to* **commute** *if all the paths from a vertex A to another vertex B in the diagram are equal. For instance, the diagram labeled as Category* **3** *which was shown in Figure 3.1, commutes if $f \circ g = h$, as $f \circ g$ and h are the only two paths between vertices A and C.*

Definition 6 *A* **cone** *for a diagram* **D** *in a category* **C** *is a* **C***-node A and arrows $f_i : A \rightarrow D_i$, where D_i represents vertices of* **D***, such that for each arrow g in* **D***, the diagram commutes.*

Figure 3.2(a) illustrates a cone A. The notation $\{ f_i : A \rightarrow D_i \}$ is used to refer to that cone.

Definition 7 *A cone for a diagram* **D** *is called* **universal** *if every other cone of the same diagram has a unique arrow to it. This universal cone is called the* **limit** *of the diagram* **D***.*

The dual of a category **C** is represented by $\mathbf{C^{op}}$, where $\mathbf{C^{op}}$ is a category with the same nodes of **C**, but the arrows are reversed. As a consequence, most of the categorical concepts are described in pairs, e.g. limit/colimit, cone/cocone, pullback/pushout, product/coproduct, etc.

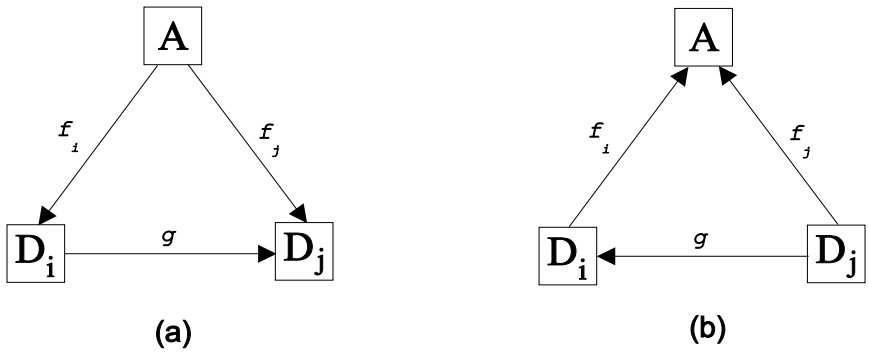


Figure 3.2: (a) A cone and (b) a cocone for a diagram

Definition 8 *The dual of a cone, i.e., a* **cocone**, *in a category* **C** *is a* **C**-*object* A *and a collection of arrows* $\{ f_i : D_i \rightarrow A \}$ *such that* $f_i \circ g = f_i$ *for each* g *in the category diagram. A* **colimit** *for this diagram is the universal cocone, such that every other cocone in the diagram has a unique arrow to it.*

Figure 3.2(b) shows the dual of the cone in Figure 3.2(a), which is the cocone $\{ f_i : D_i \rightarrow A \}$.

Having defined the general notions of *limit* and *colimit* of a diagram, we now describe some specific instances of these notions. More specifically, we introduce the example of universal/co-universal construction[1] called pullback/pushout. Other important examples of universal constructions, such as products/coproducts and equalizers/coequalizers, are described in related publications [Pie91, BW90, Mit65].

The **pullback** of a pair of arrows $f : A \rightarrow C$ and $g : B \rightarrow C$ is a node P and the pair of arrows $f' : P \rightarrow B$ and $g' : P \rightarrow A$ such that the diagram commutes, i.e. $f \circ g' = g \circ f'$. In addition, if a node X and two arrows $i : X \rightarrow A$ and $j : X \rightarrow B$ form another cone of the diagram, then there is a unique arrow $k : X \rightarrow P$ that

[1] A **universal construction** describes a class of nodes and arrows that share a common property.

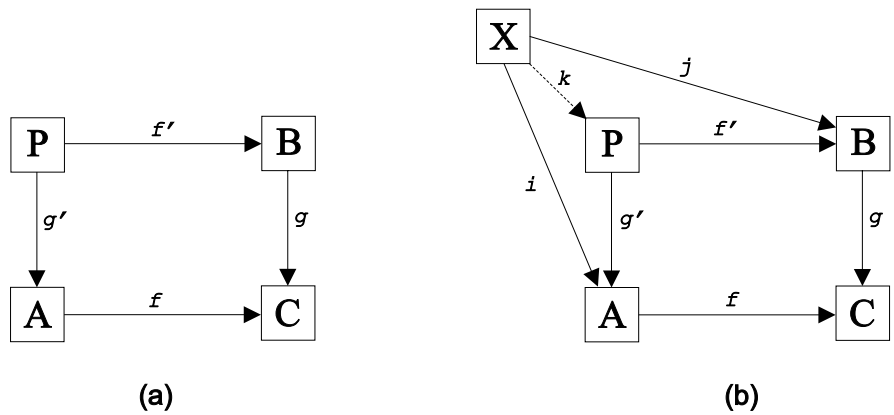


Figure 3.3: The pullback categorical construct

can be added to this diagram such that the diagram still commutes, i.e. $i = g' \circ k$ and $j = f' \circ k$. Figure 3.3 illustrates this pullback diagram.

Pushouts represent the dual notion of pullbacks. Thus, a pushout is obtained by reversing the direction of all arrows in a pullback diagram. To spell this definition out, we use the category diagram illustrated in Figure 3.5. In such diagram, node C and arrows h and k represent a cocone. In addition, this cocone will represent a pushout diagram if for any other cocone diagram with node E and arrows $h' : A \to E$ and $k' : B \to E$, there is a unique arrow $j : C \to E$ such that the diagram commutes. Barr and Wells [BW90] calls this pushout as an *amalgamated sum* of the objects A and B. *Pushouts are the way you identify part of one node with a part of another* [BW90].

3.2 Object Calculus Theories

An object theory describing a component of the system consists of a pair (θ, Φ), where θ is the component signature, and Φ is a set of θ-formulae. The component

signature defines the specific vocabulary symbols which are useful to describe components, while the θ-formulae represent the set of axioms used in the component description.

An object (or relationship) signature θ consists of three distinct parts. $\mathcal{S}$ is a set of constant symbols, $\mathcal{A}$ is a set of attribute symbols and $\mathcal{G}$ represents a set of action symbols. The set of constant symbols is composed by sorts and functions and contains the information that is state independent. An object can use, for example, booleans, natural numbers, or sequences of characters as constant symbols. If a natural number is to be included as a sort that belongs to $\mathcal{S}$, then a set of functions that operate on them is also provided. The attributes represent the state-dependent information of an object. They describe the data that can change as time passes through the actions of the objects. Attributes are the observable properties of an object. Finally, the set of actions accounts for changes in attribute values and interaction with other objects. These sets define the boundaries of an object description and are complemented by axioms which specify the behavior of the object.

From a given signature θ of an object $\mathbf{A}$, we can inductively construct the set Φ of well-formed θ-formulae relative to the component $\mathbf{A}$. A θ-formula is a term built from θ-terms, the quantifiers $\forall$ and $\exists$, and some temporal logic operators. Thus, for any signature, we first construct the θ-terms, and later the θ-formulae.

θ-Terms and θ-Formulae. Given an object signature $\theta =< \mathcal{S}, \mathcal{A}, \mathcal{G} >$, for every sort $s \in \mathcal{S}$, terms, atomic formulae and well-formed formulae are, respectively, defined as follows:

$$t_s \ ::= \ c \ | \ x_s \ | \ a(t_{s_1}, \ldots, t_{s_n}) \ | \ f(t_{s_1}, \ldots, t_{s_n}) \ | \ \bigcirc t_s$$

$$\phi_{atom} \ ::= \ (t_s \ =_s \ t_s) \ | \ g(t_{s_1}, \ldots, t_{s_n}) \ | \ \boxed{\text{BEG}}$$

$$\phi ::= \phi_{atom} \mid (\neg\phi) \mid (\phi \rightarrow \phi) \mid (\bigcirc\phi) \mid (\Diamond\phi) \mid (\Box\phi) \mid (\phi \mathbf{U} \phi) \mid (\forall x_s \phi) \mid (\exists x_s \phi).$$

where the symbols c, x_s, a, f, and g denote constants, variables of sort s, attributes, functions, and actions, respectively. ■

The special temporal logic operators used previously are $\boxed{\text{BEG}}$ denoting "a predicate that is true exactly at the first moment", $\bigcirc$ denoting "at the next method initiation time" ($\bigcirc\phi$ holds in a state when ϕ holds in the next state), $\Diamond$ denoting "sometime in the future" ($\Diamond\phi$ holds when ϕ holds in some future state), $\Box$ denoting "always in the future", and $\mathbf{U}$ denoting "until" ($\phi\mathbf{U}\psi$ holds when ψ will hold sometime in the future and ϕ holds between now and then).

The *future* operators $\Diamond$ and $\Box$ may also be derived from $\mathbf{U}$ as:

$$\Diamond\phi = true\mathbf{U}\phi$$

$$\Box\phi = \neg\Diamond\neg\phi$$

3.3 An Interpretation Theory for a Class

In this section, we introduce an interpretation theory for a generic class based on an object calculus described by Bicarregui, Lano, and Maibaum [BLM97a]. Such authors describe an object class as an interpretation theory in temporal logic: a signature θ and a set of axioms Φ. The theory of a class $\mathbf{A}$ is given by a combination of two distinct theories: *class instances* and *class manager*. A typical class instance theory A_i represents the theory for every object of this class. This theory introduces sorts for the type of each attribute ($\mathcal{S}$), the attributes ($\mathcal{A}$), and the actions ($\mathcal{G}$). The following is an example of signature of a theory for a generic instance of $\mathbf{A}$:

$\mathcal{S} = \{Type_1, Type_2\}$

$\mathcal{A} = \{attr_1 : Type_1, attr_2 : Type_2\}$

$\mathcal{G} = \{act_1, act_2\}$

Note that one important constraint to be satisfied by each object description in the object-oriented approach is given by the locality axiom. This axiom guarantees the encapsulation of attributes in an object. In other words, the attributes (state) of an action may be modified only by actions which are local to the object. Such locality requirement is specified by the following axiom.

For every signature $\theta =< \mathcal{S}, \mathcal{A}, \mathcal{G} >$

$$Locus_\theta : \quad ((\bigvee_{g \in \mathcal{G}} (\exists x_g) g(x_g)) \ \vee \ (\bigwedge_{a \in \mathcal{A}} (\forall x_a)(\bigcirc a(x_a) = a(x_a))))$$

where for each symbol u, x_u is a tuple of distinct variables of the appropriate sorts.

■

This axiom means that either one of the actions $act_i \in \mathcal{G}$ of the object is performed, or else all the attributes $attr_i \in \mathcal{A}$ will remain invariant. As an example, the locality requirement for an object of class **A**, whose signature was previously defined, is given by:

$$act_1 \vee act_2 \vee (\bigcirc attr_1 = attr_1 \wedge \bigcirc attr_2 = attr_2)$$

The second theory for the specification of a class is called class manager. A class manager theory M controls the creation and destruction of instances of a class. For a general class type X, the theory introduces a sort for identifiers of objects called $@X$. The $@X$ sort is a set of identifiers for any possible instance of X, which includes currently existing and non-existing – which are unborn or dead – instances of X. The set of currently existing instances is defined by an attribute

$\overline{X}$ of M. The class manager theory also specifies actions to create and kill objects of X. The following is the signature for M:

$\mathcal{S} = \{@X\}$

$\mathcal{A} = \{\overline{X} : \mathbb{F}@X\}$

$\mathcal{G} = \{create : @X, kill : @X\}$

According to the theory M, a class instance can only be added or removed from the list of existing objects by means of execution of the create or kill actions. Objects will be added or removed from $\overline{X}$ if pre-conditions to the execution of corresponding actions are met. These pre-conditions should avoid the creation of already existing objects, as well as the deletion of non-existing objects. The pre- and post-conditions to the actions in theory M are summarized in the following temporal logic axioms

$$create(x) \Leftrightarrow x \notin \overline{X} \wedge x \in \bigcirc\overline{X} \tag{3.1}$$

$$kill(x) \Leftrightarrow x \in \overline{X} \wedge x \notin \bigcirc\overline{X} \tag{3.2}$$

In addition to these axioms, an initialization rule states that the initial set of existing instances of X is empty. This is formalized by:

$$\mathbf{BEG} \Rightarrow \overline{X} = \emptyset \tag{3.3}$$

While the previous axioms describe the effects of the occurrence of *kill* and *create* actions on the attributes, they also indicate that these two actions interfere with each other. As a consequence, the axioms restrict their ability to occur concurrently. This may be proved by deriving $\neg(create \wedge kill)$ as a theorem of the description of objects.

Theorem 1 *create and kill do not occur concurrently.*

From Axioms 3.1 and 3.2 we have

$$create(x) \Rightarrow x \notin \overline{X}$$

$$kill(x) \Rightarrow x \in \overline{X}$$

Consequently, from the above rules and a propositional calculus axiom we derive

$$\neg(create(x) \wedge kill(x))$$

Following the formal object theory in [BLM97a], we use morphisms to combine the class manager and class instance theories with the theory of the class, as illustrated in Figure 3.4. As a result, a **self** identifier — which is the name an object refers to itself — is mapped by the morphisms to global identifiers, such as x_i, inside the class theory. In addition, each attribute and action symbol of an instance will have an extra parameter for identification. For example, an attribute *attr* of a class instance x_i is conveniently identified as $x_i.attr$. A rigorous specification for the combination of every two object theories is shown in the next section.

3.4 Combining Object Theories with Morphisms

As previously described, a category describing a complex system is composed of object theories and morphisms to interconnect these theories. While in the previous section we described some temporal logic axioms for the specification of typical object theories, we now concentrate on the concepts of morphisms as a means to synchronize objects in a category.

Morphism Between Theory Presentations. A morphism of object theory presentations $\sigma : < \theta_1, \Phi_1 > \longrightarrow < \theta_2, \Phi_2 >$ is an object signature morphism $\sigma : \theta_1 \longrightarrow \theta_2$ such that every axiom in $< \theta_1, \Phi_1 >$ is translated as a theorem of

$< \theta_2, \Phi_2 >$. In addition, the locality axiom of the first should also be a theorem of the second. These two conditions are stated as

$$(\Phi_2 \Rightarrow_{\theta_2} \sigma(\phi)) \text{ holds for every } \phi \in \Phi_1;$$

$$(\Phi_2 \Rightarrow_{\theta_2} \sigma(Locus_{\theta_1})),$$

where the formula on the right-hand side of the second rule is the translation of the locality requirement. ■

A signature morphism is used in the previous definition to relate the language of two different object signatures. Essentially, this morphism identifies the data, the attributes, and the actions of the two different signatures.

Signature Morphism. A morphism between signatures $\sigma :< \mathcal{S}, \mathcal{A}, \mathcal{G} > \longrightarrow < \mathcal{S}', \mathcal{A}', \mathcal{G}' >$ is defined by a trio of the total functions $\sigma_{\mathcal{S}} : \mathcal{S} \longrightarrow \mathcal{S}'$, $\sigma_{\mathcal{A}} : \mathcal{A} \longrightarrow \mathcal{A}'$, $\sigma_{\mathcal{G}} : \mathcal{G} \longrightarrow \mathcal{G}'$. However, for brevity, we can state that a morphism is given by $\sigma : \theta \longrightarrow \theta'$. ■

A morphism between theory presentations (or a description morphism) is a signature morphism that defines a theorem-preserving translation between the two theory presentations, and also preserves the translation of the locality axiom. These morphisms can be used to express a system as an interconnection of its parts, that is, as a diagram. This diagram is a directed multigraph in which the nodes are labeled by theory specifications, and the edges by the specification morphisms. Figure 3.4 illustrates an initial diagram that shows the morphisms between each object instance $\mathbf{A_i}$ of a class theory $\mathbf{A}$ and the class theory $\mathbf{A}$ itself, which was described in the previous section. This diagram will be later complemented with the addition of other theories, thus composing a more complex system.

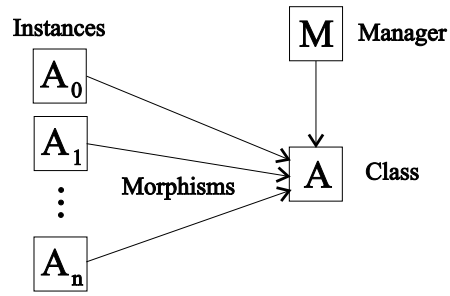


Figure 3.4: Morphisms between instances, class manager and class theories

A diagram of specifications can be collapsed to a single specification theory by taking the colimit of a diagram. This colimit may be then used to infer properties of the system as a whole. Informally, the colimit of a diagram is the disjoint union of all specifications (attributes, actions and axioms), together with the identification of some attributes and action symbols that receive the same name. For example, if two attributes $attr_A$ and $attr_B$, have been identified they receive the same name $attr$ in the resulting colimit.

We now show how combinations of object theories can be achieved with categories. As previously described, morphisms are used to express relationships between the component objects, and the composite object is obtained by constructing the colimit of the diagram that represents the interaction among the objects. We illustrate the construction of the colimit with a case in which we synchronize two object theories A and B. These two theories interact through a common subcomponent S which synchronizes the interaction. Synchronization in this case identifies actions in A with actions in B. We create an object theory S and two morphisms f and g such that $f : S \longrightarrow A$ and $g : S \longrightarrow B$. This combination is represented by the following diagram:

$$A \longleftarrow^{f} S \longrightarrow^{g} B.$$

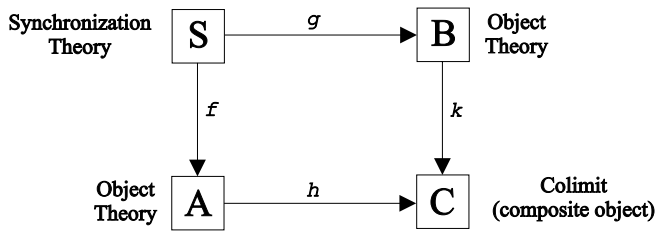


Figure 3.5: Morphisms forming a composite of two object theories

In order to obtain the object describing the combination of A and B, we build the colimit of this diagram. In this particular case, as we are only dealing with two elements, the colimit is called a *pushout*. Pushouts are just examples of the combination of object theories by assembling them in a diagram and connecting them through the appropriate interfaces. The pushout of such a diagram consists of another theory C of the category together with two morphisms represented by

$$A \longrightarrow^{h} C \longleftarrow^{k} B.$$

where h and k are morphisms such that:
(a) $h \circ f = k \circ g$ (i.e. only one copy of S is in C), and
(b) C is minimal in the sense that for an arbitrary object E, such that $h' : A \longrightarrow E$ and $k' : B \longrightarrow E$, there is an unique morphism $j : C \longrightarrow E$ such that $j \circ h = h'$ and $j \circ k = k'$.

Figure 3.5 illustrates a complete diagram of the combination of two object theories to generate a composite object C.

3.5 An Interpretation Theory for a Relationship

As previously defined, an interpretation theory for a class combines class instances and class management theories. In this section, we build an object calculus theory

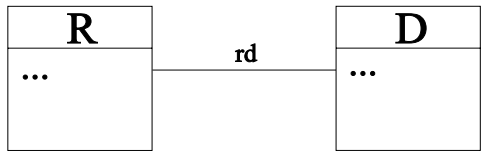


Figure 3.6: A general relationship

that relates objects defined by two class theories, namely R and D. Such a relationship theory may be described as a "middle theory" whose purpose is to identify subcomponents of the two given class theories that need to be synchronized.

An object relationship theory is also expected to be dependent on the structure (domain) of the class theories it relates. In fact, a relationship theory will contain parts of the specification of each of the theories it is synchronizing. Nevertheless, the only purpose of this section is to characterize the properties of a general relationship theory which is independent of the structure of the objects it relates. Thus, for more complex and complete relationships, such as the *views* relationship, additional properties will have to be specified, including the domain-specific ones.

We now describe a general theory for a relationship between objects of classes R and D, as illustrated in Figure 3.6. The identifiers of objects in these classes are, respectively, in $@R$ and $@D$. A signature for this general relationship theory follows.

$$\mathcal{S} = \{@R, @D\}$$

$$\mathcal{A} = \{rd : \mathbb{F}(@R \times @D)\}$$

$$\mathcal{G} = \{link : @R \times @D, unlink : @R \times @D\}$$

Note that rd is an attribute that represents the set of currently existing relationship instances, and $link$ and $unlink$ are the only actions capable of creating or removing an instance of a relationship. Similarly to the axioms for the creation

and deletion of objects, the pre- and post-conditions for *link* and *unlink* are

$$link(r,d) \Leftrightarrow (r,d) \notin rd \wedge (r,d) \in \bigcirc rd \tag{3.4}$$

$$unlink(r,d) \Leftrightarrow (r,d) \in rd \wedge (r,d) \notin \bigcirc rd \tag{3.5}$$

$$\mathbf{BEG} \Rightarrow rd = \emptyset \tag{3.6}$$

The previous axioms do not assume any condition on the state of the related objects. However, one condition for the existence of a relationship is that the objects involved are currently alive. This is stated by:

$$\forall r \in @R, d \in @D \cdot (r,d) \in rd \Rightarrow r \in \overline{R} \wedge d \in \overline{D} \tag{3.7}$$

which, together with Axioms 3.4 and 3.5, yields the following post- and pre-conditions for the relationship theory actions:

$$link(r,d) \Rightarrow r \in \bigcirc\overline{R} \wedge d \in \bigcirc\overline{D} \tag{3.8}$$

$$unlink(r,d) \Rightarrow r \in \overline{R} \wedge d \in \overline{D} \tag{3.9}$$

In addition, Axioms 3.4 and 3.5 also yield concurrency restrictions which are similar to the one presented in Theorem 1. This restriction is presented in Theorem 2 and the proof, which is omitted, is identical to the one presented for Theorem 1.

Theorem 2 *link and unlink do not occur concurrently.*

Combining object and relationship theories. In the previous section we described the formal approach to the combination of object theories. In Figure 3.5 we showed a simple example where two object theories A and B are combined to form a composite object C, which contains the theory of both A and B. Now, some complexity is added to the subsystem as we introduce the theory of a relationship.

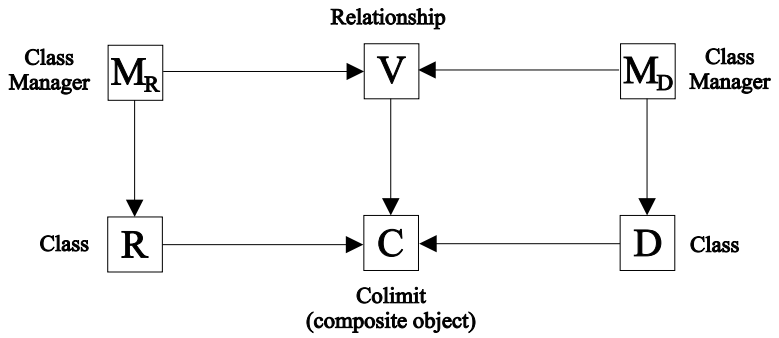


Figure 3.7: The colimit of object and relationship theories

Figure 3.7 shows the diagram where two object class theories R and D and one relationship theory V are interconnected by morphisms to derive the composite theory C. C is interpreted as the colimit of theories R, D, and V. Note that the theories responsible for synchronizing both class theories R and D with the relationship theory V are the class manager theories M_R and M_D. These class manager theories may also be interpreted as the "glue" that combines objects and relationship theories. The generic structure of a class manager was described in Section 3.3.

We now investigate some of the UML modeling properties and derive formal theories for the supported relationships. We follow these theories with an interpretation theory for the *views* relationship, which relates viewer ($@R$) and viewed ($@D$) objects.

3.6 Formalization of UML Relationships

Currently, there is not a formal specification of the UML semantics. As the UML authors claim, *the current description is not a completely formal specification of the language, because to do so would have added significant complexity without clear*

benefit [Rat97]. Thus, the semantics of the language is mostly defined in a combination of UML class diagrams and natural language. A formal language (*Object Constraint Language*) is only used to define some well-formedness rules for the UML constructs.

The current semantics specification of UML (as in [Rat97]) is certainly more detailed than the previous modeling languages from which it originated. However, their use of natural language in the definition of semantics still allows ambiguous interpretations from developers [LB98a]. As we show in this section, UML may benefit from the use of formal languages by means of precise and concise definitions of modeling concepts.

The complete UML is large. The metamodel describing the semantics of the language contains approximately 90 metaclasses and 100 metaassociations divided in several logical packages. These metaclasses and metaassociations specify semantics for both the structural and behavioral object models. Because of the size and complexity of the language, it is not among our goals to specify all the modeling concepts in UML formally. In fact, only a small portion of the relationship concepts contained in the UML structural model are formally specified in this section.

3.6.1 Association

In UML, an association represents a semantic relationship among classifiers. Classifiers are the elements of UML that represent behavioral and structural features of a system. Among others, the classifier concept includes classes, data types, and interfaces. While all of these classifier types may be defined as logic theories, we will concentrate only on the concept of classifiers as classes. We will use both terms interchangeably.

An association is composed of two or more association ends. An association end is the endpoint of an association that is attached to a single classifier. The typical association relationship is called binary and has two association ends. While the attribute properties we will define next may be easily extended for associations with more than two ends, we will restrict our formalization effort to binary associations.

In this section, we will formalize the properties related to the association of a given class A with a class B.[2] The association theory between A and B will be called ab. The instance of an association theory ab that interconnects class instances a_i and b_i is represented as (a_i, b_i) and it is called a link. Note also that all the properties defined in the previous section for the general relationship still apply to association theories.

We now investigate the six attributes defined in the UML metamodel for the *AssociationEnd* metaclass, which composes an association relationship.

The Aggregation Attribute

Aggregation is the object-oriented concept that represents a "whole-part" relationship. Some authors consider aggregation as a relationship type distinct of association [DBH95, Lan95]. UML, however, defines aggregation as an attribute of the association end that may assume three different values: *none*, *aggregate*, and *composite*. For simplicity, we represent the aggregation attribute of the end of the association instance (a_i, b_i) that is attached to the class instance a_i as $aggregation(a_i)$. Thus, a binary association theory will contain one attribute for each end of the association:

$$aggregation : A \rightarrow \{none, aggregate, composite\}$$

[2] We renamed classes R and D from Section 3.5 to A and B, respectively, to avoid confusion with the *views* properties defined in the next section.

$$aggregation : B \rightarrow \{none, aggregate, composite\}$$

Whenever the attribute value of one end of the association differs from *none*, it means that the relationship is an aggregation and the object at that end represents the "whole". Consequently, the object at the other end of the relationship instance represents the "part". By definition, the attribute value of the "part" end of the relationship should be *none*. This constraint is defined by the following UML well-formedness rule:

$$(a, b) \in ab \wedge aggregation(a) \neq none \Rightarrow aggregation(b) = none \tag{3.10}$$

UML defines aggregation as a special form of a binary association that specifies a whole-part relationship. A *composite* aggregation implies a strong form of a relationship where a "part" is included in at most one composite object (i.e. the "whole") at a time, even though the owner may be changed over time. Alternatively, the *aggregate* attribute value implies a weaker type of aggregation. In such a case, the "part" may be shared among several "wholes". These "whole" objects may also change over time.

The above definition of aggregation is presented in the UML Semantics manual in natural language [Rat97]. While the authors provide a detailed description of the concept, there is still margin for ambiguous interpretations. For instance, the distinction between a common association relationship (i.e. *aggregation* attribute value is *none*) and a "shared" aggregation relationship (i.e. *aggregation* value is *aggregate*) is not completely clear with respect to binding. For instance, we do not have a precise definition whether a "part" may exist without being associated to a "whole". Therefore, we assume that *being always bound* is a characteristic of both the *composite* and the *aggregate* relationships. Such a binding concept assures

that a "part" object can only exist if interconnected to a "whole" object. This is formally stated as:

$$\forall a \in \overline{A} \cdot aggregation(a) \neq none \Rightarrow \Box(b \in \overline{B} \iff \exists a_1 \in \overline{A} \cdot (a_1, b) \in ab) \quad (3.11)$$

Note that the above formula allows the "whole" object to change over time. Thus, a "part" may be bound to distinct "wholes" at different stages of its lifetime. Note also that the antecedent of the previous formula (i.e. $\forall a \in \overline{A} \cdot aggregation(a) \neq none$) is just a condition on the *aggregation* attribute of the association end attached to class A. The value of $aggregation(a)$ should not vary for every $a \in \overline{A}$.

The previous property defines the difference between the two forms of aggregation and a typical association. In addition, the two following rules will formally distinguish the *aggregate* form of aggregation from the *composite* form. The first formula guarantees that there will be at most one "whole" object associated with each "part" in a composite aggregation. It is stated as:

$$\forall a \in \overline{A} \cdot aggregation(a) = composite \wedge (a_1, b) \in ab \wedge (a_2, b) \in ab \Rightarrow a_1 = a_2 \quad (3.12)$$

The second formula assures that, in a composite aggregation, whenever the "whole" is killed, then so are its aggregated parts. It is shown as:

$$\forall a \in \overline{A} \cdot aggregation(a) = composite \wedge (a, b) \in ab \wedge kill(a) \Rightarrow kill(b)$$

Similarly to the axiomatization of aggregation properties described in this section, Lano and Bicarregui also use temporal logic axioms to specify the semantics of UML aggregation relationships [LB98a]. However, these authors assume that in a composite aggregation "parts" cannot be deleted whilst the "whole" continues to exist. Furthermore, their work is not extended to the other association attributes.

The Changeable Attribute

This attribute specifies constraints to the creation and removal of association instances for an object after such an object is created. Similarly to the *aggregation* attribute, an association theory contains one *changeable* attribute for each end of the association. Therefore, a binary association theory will have the following attributes:

$$changeable : A \rightarrow \{none, frozen, addOnly\}$$

$$changeable : B \rightarrow \{none, frozen, addOnly\}$$

When the changeable attribute associated with one end of the relationship has value *none*, it implies that there are no constraints to the creation or removal of association instances which involve the objects at that end. In other words, if $changeable(a) = none$ for any $a \in \overline{A}$ then instances of an association ab, such as (a, b_i), may be created and removed at any time of the lifespan of object a.

Alternatively, if the changeable attribute of one end has value *frozen* then no association instance may be added to an object at that end of the association after its creation. So, an object $a \in \overline{A}$ will never be associated with another object $b \in \overline{B}$ if this association link was not created at the same time as a. This property may be formally stated as:

$$changeable(a) = frozen \wedge link(a, b) \Rightarrow create(a) \tag{3.13}$$

or, alternatively, as:

$$changeable(a) = frozen \wedge a \in \overline{A} \wedge (a, b) \notin ab \Rightarrow (a, b) \notin \bigcirc ab$$

We believe that whenever the attribute $changeable(a)$ has value $frozen$, all links involving an object $a \in \overline{A}$ will be, not only created, but also destroyed together with

one of the objects in the link. In other words, a will be alive until one of its links is destroyed. UML, however, does not state this destruction property explicitly, even though we believe this is the intended semantics. So, in case this *freezing* property holds both for creation and destruction, we have to add the following rule to the association theory:

$$changeable(a) = frozen \wedge unlink(a, b) \Rightarrow kill(a)$$

AddOnly is the other possible value for the *changeable* attribute. In this case, association instances may be added at any time to an object, but once created, this link will only be destroyed together with the object with changeable attribute set to *addOnly* in this relationship instance. The semantics associated with the *addOnly* value is the same as the semantics defined in the previous rule for the *frozen* case. Thus, the constraint rule for the *addOnly* value is defined as:

$$changeable(a) = addOnly \wedge unlink(a, b) \Rightarrow kill(a) \tag{3.14}$$

Remember that **U** is the temporal logic operator "until".

The IsOrdered Attribute

The order in which instances of an association interconnect an object to others may be relevant to model some characteristics of a system. While the specific order of the association links is determined only by the operations creating these relationship instances, UML associations provide a Boolean attribute which specifies whether ordering is relevant in one particular association end. *IsOrdered* is such an attribute, and it indicates that partial ordering relations exist on the set of association instances.

Reflexivity	$\forall s \in ab \cdot (s, s) \in R$
Irreflexivity	$\forall s \in ab \cdot (s, s) \notin R$
Symmetry	$\forall s_1, s_2 \in ab \cdot (s_1, s_2) \in R \Rightarrow (s_2, s_1) \in R$
Asymmetry	$\forall s_1, s_2 \in ab \cdot (s_1, s_2) \in R \Rightarrow (s_2, s_1) \notin R$
Antisymmetry	$\forall s_1, s_2 \in ab \cdot (s_1, s_2) \in R \wedge (s_2, s_1) \in R \Rightarrow s_1 = s_2$
Transitivity	$\forall s_1, s_2, s_3 \in ab \cdot (s_1, s_2) \in R \wedge (s_2, s_3) \in R \Rightarrow (s_1, s_3) \in R$

Table 3.1: Binary relation conditions

A partial order relation on the set of links ab is a set of pairs (s_i, s_j), where $s_i, s_j \in ab$, that follow some of the conditions defined in Table 3.1. The symbol "$<$" is commonly used to represent partial orderings which are irreflexive, asymmetric, and transitive relations. Alternatively, the symbol "$\leq$" is commonly used to represent partial orderings which are reflexive, antisymmetric, and transitive relations [Sah81]. We shall use R(a) to denote a partial ordering on the links involving the object instance a, where $a \in \overline{A}$. R(a) should respect the same set of conditions defined for the "$<$" or "$\leq$" type of partial ordering.

To specify the constraints introduced by the *isOrdered* attribute formally, we first define the set SP(a) of all association instance pairs as:

$$SP(a) = \{((a, b_1), (a, b_2)) \mid (a, b_1) \ and \ (a, b_2) \in ab \wedge b_1 \neq b_2\}$$

So, for the cases where the *isOrdered* attribute at one end of the relationship is set to *true*, we define the following rule:

$$isOrdered(a) = true \Rightarrow \exists R(a) \cdot R(a) \subseteq SP(a) \wedge R(a) \ is \ a \ partial \ ordering \ relation$$

The Multiplicity Attribute

Multiplicity is the association end attribute that specifies constraints on the number

of object instances that may be associated with one single object at the other end of the relationship. The value of this attribute specifies a range of non-negative integers. The size of this range may vary from one integer to an infinite number of integers.

Formally, we define the attribute $multiplicity(a)$ as being an element of sort $RANGE$, where $RANGE$ is the powerset of the set of integers in the interval $[0, \infty)$. In addition, we define the set S(a) of all association instances in ab which involve the object a, where $a \in \overline{A}$, as:

$$S(a) = \{(a, b_i) \mid (a, b_i) \in ab\}$$

Hence, the constraint rule to be added to the association theory is defined as:

$$\forall a \in \overline{A} \cdot sizeof(S(a)) \in multiplicity(a)$$

where *sizeof* is a function that returns a nonnegative integer denoting the size of a set. $RANGE$ is the domain of this function.

The IsNavigable Attribute

When placed at one end of the association, this attribute indicates whether the object at the other end of the relationship will be granted access to the data of the object connected at this end. The object will be accessible (i.e. *isNavigable* value is *true*) if the objects associated contain references to elements of that object.

IsNavigable is defined as a Boolean attribute in the relationship theory. In addition, as this attribute indicates directions to be followed at implementation time, no additional constraints will be specified in the formal theory for the *isNavigable* attribute.

The TargetScope Attribute

The *targetScope* attribute is an extension from UML 1.0 [Cor97] incorporated in the language in Version 1.1 of UML [Rat97]. It represents a form to store meta-information. If the value of the *targetScope* attribute is *instance*, then a normal association is being represented. However, if the value of *targetScope* is *classifier*, it means that the association is a relationship involving a class itself, rather then the instance of the class.

This attribute allows the representation of features supported by certain programming languages (e.g. Java), which provide mechanisms to access meta-information at runtime. However, our formal framework does not support the representation of meta-information. Therefore, we define *targetScope* as an attribute of the theory without adding any specific constraints to the theory. This information may still be available at implementation time.

$$targetScope : A \rightarrow \{instance, classifier\}$$

$$targetScope : B \rightarrow \{instance, classifier\}$$

3.6.2 Aggregation

The concept of *aggregation* has a variety of meanings in object-oriented methods. Sometimes, this meanings are not even precisely defined inside a particular method. While the UML semantics for aggregation was defined in the previous section by means of an association end attribute, we now illustrate a few other types of aggregation according to a classification mechanism defined by Lano [Lan95].

Lano used three attributes to generate different types of aggregation. *Binding* is the attribute defining whether a "part" can exist without being contained by one

Unbound	Replaceable	Shareable	aggregation(a)	changeable(a)	changeable(b)
No	No	No	composite	frozen	frozen
No	No	Yes	aggregate	frozen	frozen
No	Yes	No	composite	none	frozen
No	Yes	Yes	aggregate	none	frozen
Yes	No	No	none	frozen	none
Yes	No	Yes	none	frozen	none
Yes	Yes	No	none	none	none
Yes	Yes	Yes	none	none	none

Table 3.2: Different forms of aggregation relationship

specific "whole". *Replaceability* specifies whether the "whole" can exist without having one specific value as a "part." Finally, *sharing* defines whether two or more "wholes" can share a common part.

Table 3.2 presents the results of distinct combinations of value for the three attributes analyzed by Lano. While the three columns on the left side of the table represents Lano's attributes, the three columns on the right are UML attribute values used to create equivalent relationships. For instance, one of Lano's aggregation which is not unbound, replaceable, or shareable is equivalent to a UML association with $aggregation(a) = composite$, $changeable(a) = frozen$, and $changeable(b) = frozen$.

Note that the last four rows of the table represent associations which are not aggregations. Furthermore, the last two relationships are called *exclusive association* and *general association*, respectively. While the difference between these two relationships is not shown in the UML attributes of the table, such a difference would be noticeable if the *multiplicity* attribute of UML was also illustrated.

3.6.3 Generalization

Inheritance is another type of object-oriented relationship which is presented in many forms among different methods. Some formal languages provide "ad-hoc" inheritance in which methods and attributes of the inherited class may be redefined, or even removed by the class inheriting a specification. In UML, the mechanism of inheritance is characterized by a generalization relationship between two *GeneralizableElements*. In our formalization effort, *classes* will be the only *GeneralizableElements* addressed.

The authors of UML characterize generalization as a taxonomic relationship connecting a generalized version of a class (i.e. a supertype) to a more specialized version of that class (i.e. a subtype). Since generalization is a subtyping relationship, an object instance of a supertype may be substituted by an object instance of the subtype without changing the expected behavior of the system. This desired effect is called *substitution property*, and is defined by Liskov [Lis88].

One immediate interpretation of the substitution property for a relationship between a superclass A (i.e. a supertype) and a subclass B (i.e. a subtype) is that every instance of the subclass is also an instance of the superclass. This is formally stated as:

$$\overline{B} \subseteq \overline{A}$$

We formally define inheritance of a class B from a class A by means of a morphism from A to B such that the class sort of B is a subsort of the class sort of A. Such a morphism guarantees that all sorts, attributes, actions, and axioms of the superclass A are embedded in the subclass B. Thus, all operations and attributes that apply to A also apply to B.

The multiple inheritance concept may be defined by means of colimit operations,

as shown in [DBH95]. For example, if a class B multiply inherits properties from classes A_1 to A_n, then a theory for a class A_{col} may be created as the colimit of the theories for A_1 to A_n extended with the class sort A_{col}. Thus, when B inherits from A_{col}, it is actually inheriting the properties of all A_i classes, where $0 < i \leq n$.

3.7 Properties of the *Views* Relationship

Each different kind of object-oriented relationship "glues" objects together in a particular way. The semantic properties of these relationships introduce static and dynamic constraints which characterize the type of interaction between the related objects. These constraints determine how one action occurring in the object in one end of the relationship affects the object in the other end. Our current interest is to specify the properties and constraints over a *views* relationship theory V between viewer and viewed objects. Note that $V = \{\mathcal{S}_V, \mathcal{A}_V, \mathcal{G}_V\}$. The signature of theory V is equivalent to the signature of the general relationship described in the previous section.

3.7.1 Self Dependencies

One simple rule for the *views* relationship is that viewer and viewed objects should have different identities. In the context of object-oriented analysis the representation of relationships among different objects is the point of interest. It is not relevant to say that one object *views* itself. This implies that viewer actions and attributes will not be mapped to elements inside an object's own structure. This trivial constraint is formally stated by:

$$link(r, d) \Rightarrow r \neq d \tag{3.15}$$

Property 1 *The* views *relationship is irreflexive.*

While Property 1 establishes a constraint for an individual object, it does not constrain an object to view another object of the same class. In this regard, lets assume that an object class X *views* itself. Thus, every object instance of X would *view* another object instance of this same class X. As viewer objects cannot exist without having an already existing viewed object to reference, as will be formally stated in later sections, this relationship would create a circular prerequisite which would be cumbersome to fulfill. All the objects in this circular dependency structure would have to be created and destroyed at the same time. Hence, we state that an object class cannot have a *views* relationship to itself. This is formalized as:

$$link(r, d) \Rightarrow R \neq D \tag{3.16}$$

where R and D are the class types of the viewer object r and viewed object d, respectively.

Property 2 Views *is a relationship between objects of different classes.*

Note that Property 1 is always valid when Property 2 holds. As the latter property requires that objects related by *views* relationships belong to different classes, consequently, these same objects will have different identities.

3.7.2 Acyclic Structural Dependencies

For reasons similar to the ones stated in the previous subsection, *views*-cycles in the static structure of models will also create circular prerequisites which are difficult to fulfill and limit the benefits of using the relationship. Thus, we require the *views*

dependencies in a model to be acyclic. Based on a definition given by Partsch [Par90], this property is stated as:

$$\neg\exists C \cdot C \subseteq \bigcup_{i=1}^{n} \overline{X}_i \wedge C \neq \emptyset \wedge (\forall r \cdot r \in C \Rightarrow \exists d \cdot d \in C \wedge (r, d) \in rd)$$

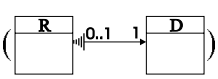
where $\overline{X}_i$ is an attribute that represents the set of currently existing instances of a class X_i, and $\bigcup_{i=1}^{n} \overline{X}_i$ is the union of all instances of the n classes in the model.

Property 3 *The structural dependencies of a given model have no* views-*cycles.*

3.7.3 Cardinality Constraints

Cardinality is a constraint over the number of instances of a class during the execution of a system. As an example, for a typical association, the cardinality of the class in one end of the relationship may be either fixed at any positive integer, or variable inside a non-negative range. The cardinality of a class is a design decision to be taken during the development of a system.

While the cardinality at both ends of an association could be over many possible ranges, other object relationships may establish more rigorous constraints. *Views* is one of these relationships with more stringent cardinality constraints, and some of the allowed types of this relationship are described next.

Optional Unary Viewer (

).

In this type of relationship, the viewed object d may be related to zero or one viewer objects. However, when the viewer object is alive, i.e. $r \in \overline{R}$, it must be related to d. These constraints, in addition to the Axioms 3.8 and 3.9, result in the following conditions on the actions of V:

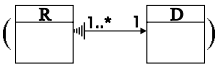
$$link(r_1, d) \wedge link(r_2, d) \Rightarrow r_1 = r_2 \tag{3.17}$$

$$link(r, d_1) \wedge link(r, d_2) \Rightarrow d_1 = d_2 \tag{3.18}$$

$$link(r, d) \Leftrightarrow r.create_R \tag{3.19}$$

$$unlink(r, d) \Leftrightarrow r.kill_R \tag{3.20}$$

The optional unary viewer relationship may be seen as an injective function $f : \overline{R} \rightarrowtail \overline{D}$, as for every r in $\overline{R}$ there is one single related d in $\overline{D}$.

Multiple Viewers (

).

At one time during execution, there may be several viewers attached to one viewed object in a multiple viewer relationship. However, in this case a viewed object d cannot exist without a viewer. This is stated as

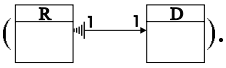
$$\forall d \in \overline{D} \cdot (\exists r \in \overline{R} \cdot (r, d) \in rd)$$

In terms of conditions on the actions of V, we have that Axioms 3.18, 3.19, 3.20 are also valid, together with

$$link(r, d) \wedge rd = \emptyset \Rightarrow d.create_D \tag{3.21}$$

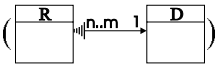
$$unlink(r, d) \Rightarrow \bigcirc rd \neq \emptyset \vee d.kill_D \tag{3.22}$$

This kind of relationship may be seen as a surjective function $f : \overline{R} \twoheadrightarrow \overline{D}$.

One-to-one View (

).

This is a combination of the two previously defined types of *views*. Consequently, the pre- and post-conditions may be obtained by combining the two previously defined sets of axioms. In addition, the one-to-one view is both injective and surjective, thus making it a bijective function.

The General Views Relationship (

).

The three previously described types of *views* relationship represent modeling constructs that may be common during development. These types represent particular cases of a general *views* relationship that we now describe.

In the general *views* case, the cardinality of a viewed class is exactly one, meaning that one viewed object exist for each related viewer object. Alternatively, the cardinality associated with the viewer class is in a range $[n, m]$, where n and m are integers so that $0 \leq n \leq m \leq \infty$. This means that, depending on the relationship specification, any natural number of viewers may be related to each instance of a viewed object. The cardinality requirements for the actions of V are expressed by Axioms 3.8 and 3.9 and Axioms 3.18, 3.19, 3.20.

Property 4 *For each object instance playing a viewer role in a* views *relationship there is exactly one related object playing a viewed role in this relationship.*

We also infer from Axioms 3.18 and 3.19 that rd, which represents the set of currently existing instances of V, may be seen as a **function** from $\overline{R}$ to $\overline{D}$. With the addition of Axiom 3.20, we may actually infer that rd is a **total funtion** from $\overline{R}$ to $\overline{D}$.

From this statement, we identify one lifetime constraint between the related objects. This constraint is that an instance of a viewer class **R** will only exist if related to some object of a viewed class **D**, which is formally stated by

$$\forall r \in \overline{R} \cdot (\exists d \in \overline{D} \cdot (r, d) \in rd)$$

Property 5 *An object playing a viewer role in a* views *relationship is always* views-*related to another object playing a viewed role.*

3.7.4 Creation/Destruction of Objects and Relationship

In a system, objects do not exist in isolation. They interact and cooperate among themselves to accomplish more complex tasks. In some cases, an object will be meaningless without others. For instance, if an object is created to monitor changes in a given subsystem, this object will be unable to fulfill its responsibilities if the mentioned subsystem is not active.

Views is a relationship that creates a unidirectional dependency between objects. Such a dependency is illustrated by the lifetime pre- and post-conditions for the related objects.

Viewer Object Lifetime Conditions. From the cardinality constraints previously specified, we can infer that the creation of a viewer object implies the creation of an instance of the *views* relationship in which this object participates, and vice-versa. Additionally, a lifetime constraint implies that if a viewer object is killed then the instance of the *views* relationship associated with this object is also destroyed. The inverse is also true, i.e. if an instance of a *views* relationship is destroyed, the viewer object in this relation is killed. These four conditions are stated in Axioms 3.19 and 3.20.

The above conditions yield the two following properties:

Property 6 *An instance of a* views *relationship associates exactly one viewer object with one other object instance that plays a viewed role in this relationship instance.*

Property 7 *The destruction of a* views *relationship instance implies the destruction of the viewer object it was relating, and vice-versa.*

Note that Property 4 differs from Property 6 in the sense that the first one refers to *views* as a relationship theory, while the latter refers to single instances of the *views* relationship.

While there is a strong correlation between the existence of a *views* relationship and its viewer elements, the same is not true for the viewed elements. In this regard, the only condition for the creation or destruction of a *views* relationship at a certain time is the existence of the viewed object at this time and, consequently, during the existence of the relationship. These conditions are defined in Axioms 3.8 and 3.9.

Note that, as *views* and the viewer objects are strongly correlated, the conditions on the viewed object lifetime which result from the creation/destruction of a *views* relationship are also valid for the creation/destruction of a viewer object. Thus, for any $(r, d) \in rd$, we have:

$$r.create_R \Rightarrow d \in \bigcirc\overline{D}$$

$$r.kill_R \Rightarrow d \in \overline{D}$$

These two axioms are consistent with what was said in Property 5, that is, a viewer is always related to a viewed object.

Viewed Object Lifetime Conditions. As viewer objects may be modeled as optional (see Section 3.7.3), a viewed object may exist for a period of time without being related to any viewer. Therefore, the creation of a viewed object does not require any pre-condition related to the existence of the corresponding *views* relationship or the viewers.

On the other hand, the destruction of a viewed object implies the destruction of every related viewer object. For instance, suppose that a viewed object d, which is part of a *views* relationship (r, d), is *killed*. The previous assertion, which involves

elements of theories that are combined by the category theory approach, is expressed as the following theorem:

Theorem 3 $\vdash \forall r \in \overline{R}, d \in \overline{D} \cdot (r, d) \in rd \wedge d.kill_D \Rightarrow r.kill_R$.

Our proof starts with the inference from rule 3.2 that:

$$d.kill_D \Rightarrow d \notin \bigcirc \overline{D}$$

and the contrapositive of Axiom 3.7, which is:

$$r \notin \bigcirc \overline{R} \ \vee \ d \notin \bigcirc \overline{D} \Rightarrow (r, d) \notin \bigcirc rd$$

From these 2 previous axioms, we infer that:

$$d.kill_D \Rightarrow (r, d) \notin \bigcirc rd$$

Combining this result and the hypothesis $(r, d) \in rd$, from Axiom 3.5 we have:

$$unlink(r, d)$$

Axiom 3.20 implies:

$$r.kill_R$$

Property 8 *The destruction of an object playing a viewed role in a* views *relationship implies the destruction of the relationship itself and the viewer object participating in this relationship.*

The above conditions are expressed in Table 3.3, which describes the lifetime-related conditions for the occurrence of each of the actions specified in the object and relationship theories.

		pre-conditions			*post-conditions*		
		Views	*Viewer*	*Viewed*	*Views*	*Viewer*	*Viewed*
Views	*created*	$\notin rd$	**created**	none	$\in rd$	none	**exists**
	destructed	$\in rd$	**killed**	**exists**	$\notin rd$	none	none
Viewer	*created*	**created**	$\notin \overline{R}$	none	none	$\in \overline{R}$	**exists**
	killed	**killed**	$\in \overline{R}$	**exists**	none	$\notin \overline{R}$	none
Viewed	*created*	none	none	$\notin \overline{D}$	none	none	$\in \overline{D}$
	killed	**killed**	**killed**	$\in \overline{D}$	none	none	$\notin \overline{D}$

Table 3.3: Conditions on actions of object and relationship instances

3.7.5 Viewed Singularity and Viewer Multiplicity

So far, we have analyzed the *views* relationship as an interconnection mechanism between two distinct classes. A complete system, however, is usually composed of several classes connected by several relationships of different types and semantics. In fact, it is usual that one single class is related to other classes in the system by means of a few different relationships. Sometimes, there are different occurrences of the same type of relationship. For instance, one class A may be related in an association with another class B. At the same time, class A may be associated with class C. In this case, we have two occurrences — namely ab and ac — of the same type of relationship (i.e., association).

While there is an unlimited number of associations for which a given class may take part, other relationships may have different constraints at one or both ends of the relationship. For instance, in many specification languages [Lan95, RBP+91], the definition of aggregation[3] implies that one object will not play the contained

[3] Aggregation relates one container object to a contained one. It is also known as the *whole/part* relationship.

role for aggregation relationships more than once. In other words, an object will be contained in at most one container object. In a *vehicle* specification system, for example, a *wheel* object is contained in at most one *car* object, even though *wheel* may be the container of *tire* and *bolt* objects. The *views* relationship defines similar constraints on the objects being related.

As previously mentioned, it is a responsibility of the viewer object to monitor the behavior of a viewed object according to rules defined by a *views* relationship. Such a monitored behavior is specified by properties and constraints defined in the relationship theory. Thus, the behavior monitoring performed by a viewer object depends, not only on the attribute values of the object being viewed, but also on the rules specified by the relationship theory. Now, suppose that an object r of a class R *views* an object d according to a *views* theory V_{rd}. In case r would be allowed to *view* another object e of a different class, r would also be constrained by another *views* theory V_{re}. In such case, the *views* theories V_{rd} and V_{re} may impose incompatible constraints to the theory of R. Hence, to guarantee the semantic integrity of the theories involved, an additional constraint on the *views* modeling approach is that an object r will play the role of a viewer for at most one *views* relationship theory. This property is rigorously stated by:

$$\forall r, d, e \cdot (r, d) \in rd \wedge (r, e) \in re \Rightarrow rd = re$$

or, using the relationship actions, by the axiom:

$$link(r, d) \wedge link(r, e) \Rightarrow rd = re \tag{3.23}$$

The above rule implies that, if the premises are true, then the relationships rd and re are the same, which also implies that d is an object of the same class as e. Putting this rule together with Axiom 3.18 (or Property 4), we also have that

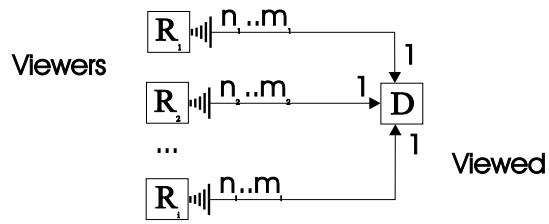


Figure 3.8: A multiple viewers example

$d = e$. The additional meaning is that an object r is limited to *view*, not only a single class of viewed objects, but also a single instance of objects playing a viewed role. These rules characterize the property that we call *viewed singularity*.

Property 9 *An object is allowed to play a viewer role for at most one* views *relationship.*

The previously stated property, defined by Axiom 3.23, also generates consequences that may be stated similarly to the cardinality constraint (i.e., Axiom 3.18). The difference of the current result is that, in the hypothesis – which is stated in the left-hand side of Axiom 3.18, – d_1 and d_2 are not necessarily objects of the same class. Thus, for our current purposes, d_1 and d_2 are replaced by d and e, which yields:

$$link(r, d) \wedge link(r, e) \Rightarrow d = e \tag{3.24}$$

Alternatively, at the other end of the relationship, a viewed object d may have several viewers referring to it. There may be not only viewer objects of the same class, but also viewers from different class structures, which implies the existence of different *views* relationship theories. This property is called *viewer multiplicity* and is illustrated in Figure 3.8.

3.7.6 Viewer and Viewed Visibility

In the categorical approach described in this chapter, morphisms provide the formal basis to express interconnections among objects in a system, as explained in Section 3.4. Fiadeiro and Maibaum [FM92] state that two objects of the system interact by sharing some other object, i.e. by having a common sub-component in which they synchronize through morphisms. Similarly, the *views* interconnection between viewer and viewed objects is also specified by sharing object signatures. The identification of these shared signatures should be founded on the visibility rules characterizing the *views* approach to modeling. These rules determine which part of an object is "visible" or "shared" by both objects.

The signatures in the specification structure of a class may be informally separated according to their specific responsibilities. For instance, while part of an object signature deals with attributes and actions determining specific object behavior, other elements of the signature will be used to model the interaction activities with other objects.

In a subsystem composed of two object classes interconnected by a *views* relationship, the specification structure of a viewed class has only signatures which are relevant to the application being defined. In other words, it has no attribute or action which was specifically intended to access any kind of information maintained by the viewer object. For instance, in a situation where the viewer is a user interface object, the viewed object should not have attributes notifying it about the viewer's interface-related information. On the other hand, viewer class specifications have not only application-specific signatures (behavior specific), but also have signatures that allow the viewer object to monitor or change the state of a viewed object (interconnection signatures).

We say that each viewer has a sub-object that is identified, or interconnected,

with all or part of the viewed object properties to which it is related. As a consequence, this sub-object imposes upon the viewer a pattern of behavior that mimics the behavior of a corresponding viewed object.

Such identification is formalized as shown in Figure 3.7, which describes the combination of relationship (V), viewer (R), and viewed object (D) theories. From this diagram, we observe that the class manager theory M_D synchronizes D and V by means of morphisms. We say that M_D contains a sub-object which is "shared" by both D and V, as it contains a theory that is identified with parts of D and V. On the other side of the relationship, M_R synchronizes V and the viewer class R. Thus, V is synchronized to both the viewer R and the viewed object D. This relationship theory allows the identification of the signatures of related objects and the specification of new axioms based on these signatures.

3.7.7 Attributes Consistency

Another characteristic of the *views* relationship is that it supports the specification of axioms that constrain the values of the attributes in the related objects. This property will guarantee that the states of the objects involved are always consistent among themselves.

Two levels of consistency must be expected when several viewer objects are related to one single viewed object. First, consistency must always exist between each viewer and its related viewed object to assure that the viewer correctly represents the state of a viewed object. A second level of consistency is achieved as a consequence of the first one, that is, all viewers of the same object should be consistent among themselves. Consistency between the viewer object and its related viewed part is called *vertical consistency*, while consistency among the different viewers

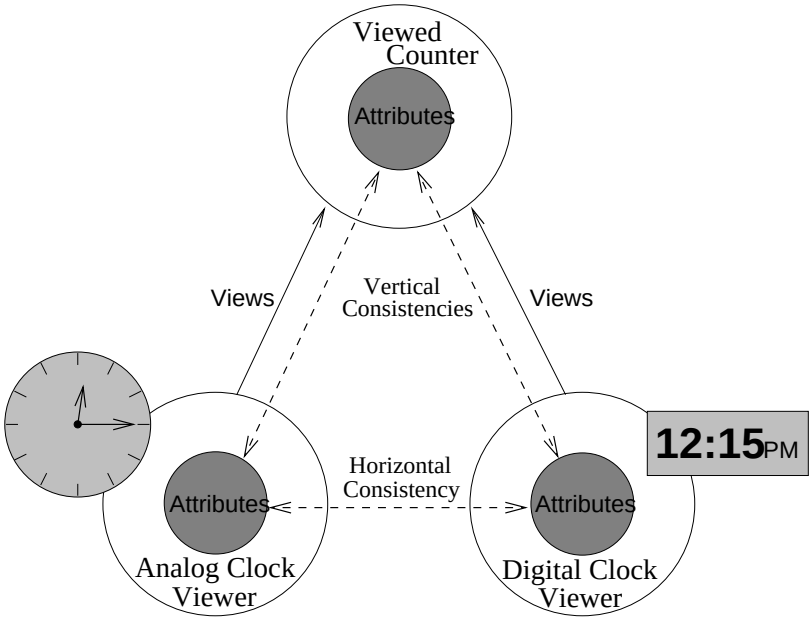


Figure 3.9: *Views* consistencies in a clock application

is called *horizontal consistency*. These consistency properties must be guaranteed by the specification of the *views* relationships being defined among the involved objects.

Figure 3.9 illustrates these consistency properties using a clock application model which contains a counter object with two distinct viewers: a digital clock viewer and an analog clock viewer. In this example, vertical consistency ensures that each viewer object shows the values specified in the attributes of the corresponding viewed object, while horizontal consistency guarantees that the different viewers, i.e., the analog and digital clocks, always show the same time.

Vertical Consistency

We say that two related objects are vertically consistent if their states are coherent with respect to the type of *views* relationship established among them. These

states are represented by the attribute values of the viewer and viewed objects. For instance, the analog clock viewer of Figure 3.9 is consistent with the viewed counter only if the attributes associated by means of the *views* relationship hold consistent values at any time. In other words, the viewed counter attribute that is mapped by *views* to the analog clock viewer should have equivalent value as the time shown, which is *12:15*.

Figure 3.10 shows some attribute mappings between the Analog Viewer and Viewed Counter objects (see also Figure 3.9). Such a diagram is a simplified instance of the general case shown in Figure 3.7, where objects and relationship theories are combined into a subsystem theory. In this particular case, the theories of the viewer and viewed objects are interconnected by a *views* relationship theory. The class manager theories M_{AR} and M_D are used to synchronize the relationship theory with the viewer and viewed object theories, respectively. Note also that a similar diagram may be obtained for the Digital Viewer and Viewed Counter objects.

In this diagram, two attribute morphisms from the class manager theory M_D, which are $t_D \rightarrow time_D$ and $t_D \rightarrow time$, specify the consistency between the viewed object ($time_D$) and the relationship ($time$) attributes. In addition, two attribute morphisms from the class manager theory M_R, which are $t_{AR} \rightarrow time_{AR}$ and $t_{AR} \rightarrow time$, define the consistency between the viewer object and the relationship attributes. These four morphisms guarantee that $time_{AR}$ and $time_D$ will have consistent values at all times.

It is important to mention that consistency between related object states is a constraint property represented by attribute morphisms, as illustrated in Figure 3.10. These morphisms specify that two attributes of related objects (viewer and viewed) always hold the same value. However, the attribute morphisms do

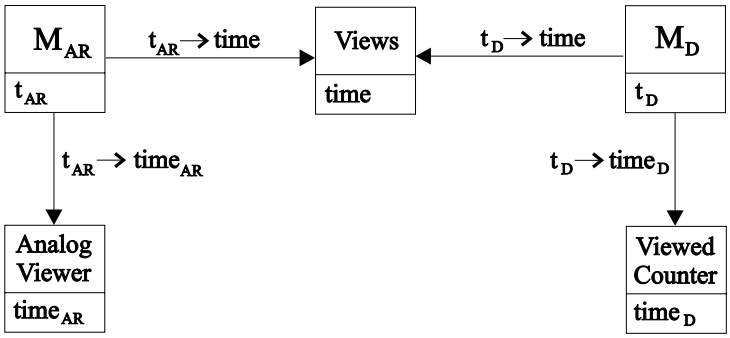


Figure 3.10: Vertical consistency through attribute morphisms

not describe how attribute $time_{AR}$, which is shown in the analog clock viewer, is updated as consequence of a change in the value of attribute $time_D$. In the next subsection, we show how such consistency is achieved by describing how attributes that are modified by actions remain consistent.

Horizontal Consistency

This kind of consistency is among viewers of one viewed object. We say that two viewer objects are horizontally consistent if each of them have attributes that are mapped to one or more common attributes of the same viewed object. In fact, horizontal consistency is a direct consequence of the vertical consistency between each viewer and viewed object.

Still using the application of Figure 3.9 as example, we see that the analog clock viewer is vertically consistent with the viewed counter state, as the morphisms between the time-related attributes $time_D$ and $time_{AR}$ have shown. For identical reasons, the digital viewer should be vertically consistent with the viewed counter. Consequently, the time-related attributes of both the analog and the digital viewers will be consistent among themselves.

3.7.8 Action Mappings

In the previous subsection we used morphisms to obtain attribute (state) consistency between viewer and viewed objects. However, morphism of attributes only is forbidden by the category of theory presentations, because if we isolate a set of attributes as a sub-object there will be no action to modify their values. This property is a consequence of the locality requirement described in Section 3.3. As shown in [FM92], the locality property implies that attributes cannot be separated from the actions that update them, thus imposing a discipline in the way we can interconnect the object theories by means of morphisms.

These conditions imply that the morphism $M_D \longrightarrow D$, which was represented in Figure 3.10, will consist not only of the attribute morphism $t_D \rightarrow time_D$, but also of a set of action morphisms involving all the actions of D which modify the value of the attribute $time_D$. In addition, the specification of the viewer object R should also contain actions that will be identified through morphisms with those actions of D which are synchronized by the morphism $M_D \longrightarrow D$. Consequently, when an action act_D of D modifies a given attribute att_D of this same object, each viewer object R that is monitoring the object D will also execute an action act_R, which is identified with the action act_D, and the corresponding attribute att_R in viewer R will also be modified.

Concurrency Constraints

The formalism we adopt for the specification of the *views* modeling approach supports concurrency of actions. This concurrency allows us to specify that the viewed and the corresponding viewer attributes are consistent at all times, as we illustrated in Figure 3.10 using the $time_D$ and $time_{AR}$ attributes. In that case, an action of the

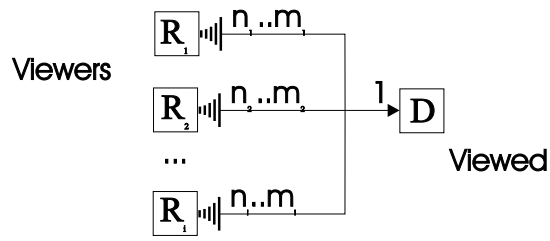


Figure 3.11: A multiple viewers example

object D would modify the attribute $time_D$ at the same time another synchronized action of the object R would update the attribute $time_{AR}$.

While this kind of action concurrency keeps the attribute values of the objects consistent, conflicting behavior may also result from simultaneous execution of actions. For instance, suppose that while an action of a digital viewer object tries to set the time attribute of the viewed counter, another action of the analog viewer object also tries to set the same time attribute to a different value. This kind of conflict may be resolved by adding some conditions to the relationship theories connecting the different objects.

Our approach to address concurrency conflicts among different viewer objects *viewing* one common part – i.e. a subset of the attributes – of a viewed object is to define the interactions of all the possibly conflicting objects with a viewed object in a single relationship theory. As this relationship theory contains actions which are synchronized with the potentially conflicting actions in related object theories, it is then responsible for defining axioms that constrain the concurrent execution of these conflicting actions. Figure 3.11 illustrates this approach by interconnecting several viewers R_i with a viewed object D by means of a single *views* relationship theory V.

Note that the object model of Figure 3.8 is very similar to the diagram shown in Figure 3.11. The difference is that the former had several *views* relationship

theories, each connecting a pair of objects. Such an approach is suitable when there are no potential conflicts among actions of the different viewers. For example, there may be cases where there are no common attributes in the attribute sets being "monitored" by the distinct viewers. Alternatively, the latter approach uses one single relationship theory for all the objects involved. This approach is generally suitable when the same viewed attributed may be modified by several viewers.

The single relationship and several viewers approach of Figure 3.11 does not introduce any limitation to the modeling process. All the previously specified properties relating to the viewed class are valid in both modeling approaches.

In the case study presented in Section 3.8 we illustrate action mappings and the elimination of potential conflicts by means of axioms constraining the concurrent execution of some viewer actions.

3.8 Case Study: Dual Interface Clock

In the previous sections we have provided a formal description of the concepts inherent in the *views* approach to modeling. While the relationship properties were defined, not much emphasis was given to the actual specification of systems which are based on the *views* approach. We specified part of a modeling language, but the actual use of such language was not of immediate concern. Therefore, our current objective is to complement the modeling language definitions with the specification of a case study composed of a few objects interconnected by *views* relationships.

The case study to be formally specified is the dual interface clock which was used and briefly described in previous sections. This simple system has an application object (Viewed Counter) which is responsible for keeping the correct time of the day, and two different types of user interface for this application. The first interface

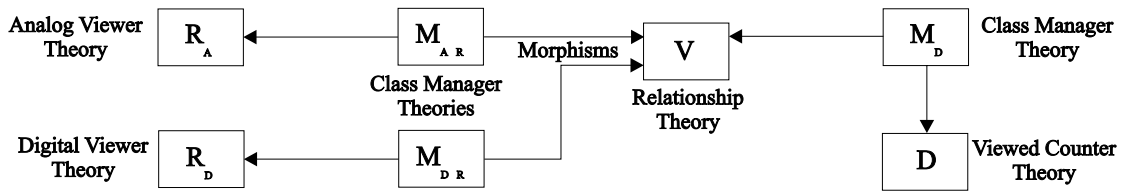


Figure 3.12: Interconnection between clock system theories

is an analog time display (Analog Clock Viewer) which has two hands moving on a dial that is divided in twelve sections. This interface does not differentiate the period of the day – i.e. AM or PM, – and with a double mouse click it resets the application counter to 12:00 AM. The second interface is a digital time display that shows the time and period of the day according to the values in the *Counter* object.

Besides the three object class theories, the system formalization involves a few other object theories and morphisms. The system specification, which is presented in the following paragraphs, starts with the viewed object description, and then introduces the two viewer (interface) theories. The relationship and class manager theories, which formalize how objects are put together, are described in sequence. Finally, a few morphisms interconnecting theories are specified. Figure 3.12 depicts all the theories in the system, which includes the viewer and viewed objects, the class managers, the relationship V, as well as some morphisms. The colimit of this diagram returns a new composite object description, which we call CLOCK-SYSTEM. Note that this diagram is an instance of the general diagram shown in Figure 3.7, which illustrated the colimit of object and relationship theories.

The specification language structure we will be using in the case study formalization is based on schemas and temporal logic, as described earlier. The first schema is shown in Figure 3.13. Such schema shows the signatures and axioms of the theory description D of the viewed counter object. Note that the state of

Object **D**

Sorts/Functions

TIME : [1..12] $\times$ [0..59];

AM-PM : { AM, PM };

Attributes

time : TIME;

period : AM-PM;

Actions

set-time : TIME;

set-period : AM-PM;

reset;

Axioms

$\boxed{\text{BEG}} \Rightarrow$ reset;

set-time(t) $\Rightarrow \bigcirc$ time = t;

set-period(p) $\Rightarrow \bigcirc$ period = p;

reset $\Rightarrow \bigcirc$ time = (12, 0) $\wedge \bigcirc$ period = AM;

$\neg$ (set-time(t) $\wedge$ reset);

$\neg$ (set-period(p) $\wedge$ reset);

End

Figure 3.13: Specification of the Viewed Counter object

D is represented by attributes *time* (which keeps number of hours and minutes in the day) and *period* (which determines whether the time is AM or PM). These attributes are only modified by the actions *set-time*(t), *set-period*(p) and *reset*. The effects of the execution of these actions on the attribute values are shown by the axioms of the theory.

The axioms of a theory may be used to specify constraints, or pre- and post-conditions on the execution of the actions. The first axiom in the specification of theory D specifies an initialization condition on the object, while the following three

Object $\mathbf{R_A}$

Sorts/Functions

TIME_{AR} : [1..12] $\times$ [0..59];

ANGLE : [0..359];

CHECK-HOURS : $\text{TIME}_{AR} \rightarrow$ ANGLE;

CHECK-MINUTES : $\text{TIME}_{AR} \rightarrow$ ANGLE;

Attributes

time_{AR} : TIME_{AR};

angle_h : ANGLE;

angle_m : ANGLE;

Actions

set-time_{AR} : TIME_{AR};

reset_{AR};

change-angle : TIME_{AR};

double-mouseclick;

Axioms

$\boxed{\text{BEG}} \Rightarrow \text{reset}_{AR}$;

$\text{set-time}_{AR}(t) \Rightarrow \bigcirc\ \text{time}_{AR} = \text{t}$;

$\text{set-time}_{AR}(t) \Rightarrow$ change-angle(t);

$\text{reset}_{AR} \Rightarrow \bigcirc\ \text{time}_{AR} = (12, 0)$;

$\text{reset}_{AR} \Rightarrow$ change-angle((12,0));

change-angle(t) $\Rightarrow \bigcirc\ \text{angle}_h =$ CHECK-HOURS(t);

change-angle(t) $\Rightarrow \bigcirc\ \text{angle}_m =$ CHECK-MINUTES(t);

double-mouseclick $\Rightarrow \text{reset}_{AR}$;

$\neg\ (\text{set-time}_{AR}(\text{t}) \wedge \text{reset}_{AR})$;

End

Figure 3.14: Specification of the Analog Clock Viewer object

define post-conditions on the occurrence of the actions, $set\text{-}time(t)$, $set\text{-}period(p)$, and $reset$. The last two axioms in D, however, deserve particular attention. For instance, axiom $\neg(set\text{-}time(t) \wedge reset)$ implies that the actions $set\text{-}time(t)$ and $reset$ cannot be executed simultaneously. This mutual exclusion axiom guarantees that these two actions within the two distinct viewers of D must not be executed concurrently if they lead to any kind of inconsistent behavior. Note that within D, the execution of $set\text{-}time(t)$ modifies the value of the attribute $time$ to a value, while the execution of $reset$ will modify $time$ to a different value. This potential inconsistent behavior is eliminated with the addition of the mutual exclusion axioms.

A second object specification is shown in Figure 3.14. In this schema, the analog viewer theory R_A is an interface for the application. Note that part of the structure of R_A, more specifically the signature elements with subscript AR (e.g., time_{AR}), is responsible for maintaining the consistency between the states of R_A and D. Such signature elements will be mapped by morphisms which allow the viewer to "observe" the viewed counter object. In addition, the axioms of D involving signatures of the morphism are also preserved in R_A, as this is a requirement of the morphism definition. Some of the system morphisms will be specified later in this section.

The other signature elements of R_A are responsible for user interface activities. The *change-angle* action is called to update the angles of the hands in the analog clock display every time an action changes the attribute time_{AR}, which is the only attribute of D being "observed" by the viewer object R_A. This action uses functions CHECK-HOURS(t) and CHECK-MINUTES(t) to calculate the new values of the angle-related attributes. The other action of the interface R_A, which is named *double-mouseclick*, triggers the $reset_{AR}$ action, whenever it is called. As a consequence,

$reset_{AR}$ will not only modify the time values in R_A, but it will also trigger the action *reset* in the object D by means of morphisms. Then, *reset* will modify the time values of D, thus keeping both viewer and viewed object states consistent.

The third object in the system is the digital viewer object, for which a specification is given in Figure 3.15. This viewer object is responsible only for monitoring and displaying the time, even though user input events in other interface objects could trigger actions such as $reset_{DR}$ to modify the counter object attribute values. Every time $set\text{-}time_{DR}(t)$, $set\text{-}period_{DR}(p)$, or $reset_{DR}$ is triggered, the display update changes the time values in accordance with the axioms. The other axioms in the specification are intended to preserve the properties of the viewed object D. Note that R_D, in contrast to R_A, monitors and displays not only the *time* attribute value in D, but also the attribute value of *period*.

In the same way as before (for the Analog Clock Viewer - see Figure 3.14), part of the structure of R_D, the signature elements with subscript DR (e.g., $time_{DR}$), is responsible for maintaining the consistency between the states of R_D and D. The other signature elements of R_D are responsible for user interface activities. For example, in the previous case (Figure 3.14) the attribute $time_{AR}$ was used for monitoring and the attributes $angle_h$ and $angle_m$ were used for user interfaces purposes. Now, for the second viewer, we use the attributes $time_{DR}$ and $period_{DR}$ for monitoring and the attributes $time_d$ and $period_d$ for displaying purposes (Figure 3.15). The actions $update\text{-}display\text{-}time(t)$ and $update\text{-}display\text{-}period(p)$ are called to update the time and period values of the digital clock displays $time_d$ and $period_d$.

Having described all the viewed and viewer objects, we now determine the pattern of interaction between these objects. Such pattern, as specified during the software modeling process, should conform to the properties of the *views* relation-

Object $\mathbf{R_D}$

Sorts/Functions

TIME_{DR} : [1..12] $\times$ [0..59];

AM-PM_{DR} : { AM, PM };

Attributes

time_{DR} : TIME_{DR};

period_{DR} : AM-PM_{DR};

time_d : TIME_{DR};

period_d : AM-PM_{DR};

Actions

set-time_{DR} : TIME_{DR};

set-period_{DR} : AM-PM_{DR};

reset_{DR};

update-display-time : TIME_{DR};

update-display-period : AM-PM_{DR};

Axioms

$\boxed{\text{BEG}} \Rightarrow \text{reset}_{DR}$;

$\text{set-time}_{DR}(\text{t}) \Rightarrow \bigcirc\ \text{time}_{DR} = \text{t}$;

$\text{set-time}_{DR}(\text{t}) \Rightarrow \text{update-display-time}(\text{t})$;

$\text{set-period}_{DR}(p) \Rightarrow \bigcirc\ \text{period}_{DR} = \text{p}$;

$\text{set-period}_{DR}(p) \Rightarrow \text{update-display-period}(\text{p})$;

$\text{reset}_{DR} \Rightarrow \bigcirc\ \text{time}_{DR} = (12, 0) \wedge \bigcirc\ \text{period}_{DR} = \text{AM}$;

$\text{reset}_{DR} \Rightarrow \text{update-display-time}((12,0)) \wedge \text{update-display-period}(\text{AM})$;

$\text{update-display-time}(\text{t}) \Rightarrow \bigcirc\ \text{time}_d = \text{t}$;

$\text{update-display-period}(\text{p}) \Rightarrow \bigcirc\ \text{period}_d = \text{p}$;

$\neg\ (\text{set-time}_{DR}(\text{t}) \wedge \text{reset}_{DR})$;

$\neg\ (\text{set-period}_{DR}(\text{p}) \wedge \text{reset}_{DR})$;

End

Figure 3.15: Specification of the Digital Clock Viewer object

ship. This means that all the axioms characterizing a general *views* relationship should hold together with additional properties to be specified for this particular case of the clock system.

Figure 3.16 shows the relationship theory V for our clock system. Parts of the theory defined in Section 3.7 for the general *views* relationship, such as the *link* and *unlink* actions, are now omitted for simplicity. Nevertheless, they are still part of the relationship theory, and so are the *views* properties introduced by their related axioms. The purpose of the signatures and axioms shown in the schema V is to synchronize the system objects.

The relationship theory V has two sets of actions: one indexed as $V1$ and the other as $V2$. Both sets act as a cable that connects the viewer objects with the relationship theory. The first cable is connected to the analog viewer theory, while the second one is connected to the digital viewer theory. The distinction between both cables allows the identification of the origin of the triggering of an action and, consequently, the specification of constraints about their execution.

The last axiom in the specification schema V represents a constraint established for the concurrent execution of viewer actions. Such an axiom states that whenever $set\text{-}time_{V1}$, which is connected to $set\text{-}time_{AR}$ by morphisms, and $set\text{-}time_{V2}$, which is connected to $set\text{-}time_{DR}$, occur simultaneously, their parameters must have equal values. This constraint guarantees that no two distinct viewers will concurrently try to set the same counter object to different times, thus generating inconsistent behavior. Note also that there is no concurrency constraint established for $set\text{-}period(p)$ as the viewer object R_A does not "monitor" the *period* attribute of D. For a different reason, no concurrency constraint was established for $reset_{V1}$ and $reset_{V2}$, as these actions have no parameters and their concurrent execution generates a consistent modification of the attribute values.

Relationship **V**

Sorts/Functions

TIME_V : [1..12] $\times$ [0..59];

AM-PM_V : { AM, PM };

Attributes

time_V : TIME_V;

period_V : AM-PM_V;

Actions

set-time_{V1} : TIME_V; set-time_{V2} : TIME_V;

reset_{V1}; set-period_{V2} : AM-PM_V;

reset_{V2};

Axioms

$\boxed{\text{BEG}} \Rightarrow \text{reset}_V$;

$\text{set-time}_{V1}(\text{t}) \Rightarrow \bigcirc\ \text{time}_V = \text{t}$;

$\text{set-time}_{V2}(\text{t}) \Rightarrow \bigcirc\ \text{time}_V = \text{t}$;

$\text{set-period}_{V2}(\text{p}) \Rightarrow \bigcirc\ \text{period}_V = \text{p}$;

$\text{reset}_{V1} \Rightarrow \bigcirc\ \text{time}_V = (12, 0) \wedge \bigcirc\ \text{period}_V = \text{AM}$;

$\text{reset}_{V2} \Rightarrow \bigcirc\ \text{time}_V = (12, 0) \wedge \bigcirc\ \text{period}_V = \text{AM}$;

$\text{set-time}_{V1}(\text{t}_1) \wedge \text{set-time}_{V2}(\text{t}_2) \Rightarrow \text{t}_1 = \text{t}_2$;

End

Figure 3.16: Specification of the *Views* relationship

Object Signature $\mathbf{M_{AR}}$

Sorts/Functions

$@\mathrm{R}_A$;

TIM : [1..12] $\times$ [0..59];

Attributes

$\overline{R}_A : \mathbb{F}\ @R_A$;

tim : TIM;

Actions

create : $@\mathrm{R}_A$;

kill : $@\mathrm{R}_A$;

sttm : TIM;

rst;

End

Figure 3.17: Specification of M_{AR} class manager signature

The class manager theories have two distinct purposes. One first part contains all the signatures and axioms which controls creation and destruction of all object instances of a class theory. These signatures and axioms were described in Section 3.3. A second part of these theories work as synchronization channels between the class theories and the relationship theory V by using signatures and axioms to maintain consistency between viewers and viewed objects. For example, in the class manager theory M_{AR} which is illustrated in Figure 3.17, action *rst* acts as a port that interconnects actions $reset_{AR}$ and $reset_{V1}$ by means of morphisms. Consequently, all actions $reset_{AR}$ of R_A class instances[4] are synchronized with both $reset_{V1}$ and the *reset* action of D.

Note that only the signature of the class manager specification is presented in Figure 3.17. The axioms for this class manager may be obtained by translations

[4]There is only one instance of R_A in this particular example

Morphism $M_{AR} \longrightarrow R_A$

Sorts/Functions

TIM $\rightarrow$ TIME$_{AR}$;

Attributes

tim $\rightarrow$ time$_{AR}$;

Actions

sttm(t) $\rightarrow$ set-time$_{AR}$(t);

rst $\rightarrow$ reset$_{AR}$;

End

Figure 3.18: A morphism between the Analog Viewer and a class manager theory

(according to the morphism property preservation requirement) from other object theories in the system, and from axioms defined for class manager theories in Section 3.3.

The dual interface clock specification also has a few morphisms interconnecting its different theories, as Figure 3.12 shows. One first morphism interconnects a class manager theory M_D to the viewed object D. Two other morphisms interconnect the viewer objects R_A and R_D to class manager theories. For instance, the morphism specified in Figure 3.18 connects the class manager theory M_{AR} to the viewer object R_A. There are also three other morphisms interconnecting class manager theories to the relationship theory V. The first of these three morphisms is illustrated in Figure 3.19 and connects V with the class manager M_{AR}. This morphism synchronizes the actions of "cable" $V1$ in theory V with actions in M_{AR}. Note that attributes do not need distinct "cables" to be connected, as no additional constraint is required. The second morphism, $M_{DR} \longrightarrow V$, connects "cable" $V2$ with theory M_{DR}. The third morphism is $M_D \longrightarrow V$. It synchronizes both cables $V1$ and $V2$ to the same actions of D (e.g., $reset_{V1}$ and $reset_{V2}$ are both synchronized to $reset$).

Morphism $M_{AR} \longrightarrow V$

Sorts/Functions

$\mathrm{TIM} \rightarrow \mathrm{TIME}_V$;

Attributes

$\mathrm{tim} \rightarrow \mathrm{time}_V$;

Actions

$\mathrm{sttm(t)} \rightarrow \mathrm{set\text{-}time}_{V1}\mathrm{(t)}$;

$\mathrm{rst} \rightarrow \mathrm{reset}_{V1}$;

End

Figure 3.19: A morphism between *Views* and a class manager theory

Chapter 4

Verification

In this chapter we focus on a logic-based specification using the formal concepts previously introduced. This process is illustrated with the definition of a number of object and relationship properties derived from a small UML model. The specification and verification of the model are developed in a formal specification environment which provides typechecking tools for a higher-order logic specification language and a powerful proof checker that allows the verification of properties of the formal specifications.

A verification process supports the proof of correctness of domain-specific properties defined within object theories, as well as properties of the relationships interconnecting these object theories. Our particular interest is on the definition and verification of the relationship properties characterizing a software model. These relationships are defined by separate theories which map elements of the objects being related, thus specifying a pattern of interaction among these objects. In this chapter we investigate some of the most popular types of object-oriented relationship, but we invest particular attention in the specification of the *views* relationship properties and its validation.

4.1 Formal Specification of Object Models

Despite extensive development over many years, formal methods remain poorly accepted by industrial practitioners [KDGN97]. There are researchers who attempt to alleviate this problem by introducing a graphical semi-formal notation together with a formal textual notation [BC95, WRC97, LB98a]. While the purpose of the semi-formal notation is the correct communication of concepts to users with little mathematical background, the formal notation supports precise specifications which may be used as an instrument to represent information unambiguously.

4.1.1 Semi-Formal Specifications

In general, semi-formal graphical representations are helpful in portraying properties and relationships of object-oriented models. However, it is not among the major objectives in this dissertation to improve or extend these graphical notations. Rather, we use a simple extension of a structural modeling notation for UML [Rat97] as described in Section 2.4. This notation is intended to convey clearly and informally some of the static object-oriented concepts we will be formally describing in this chapter.

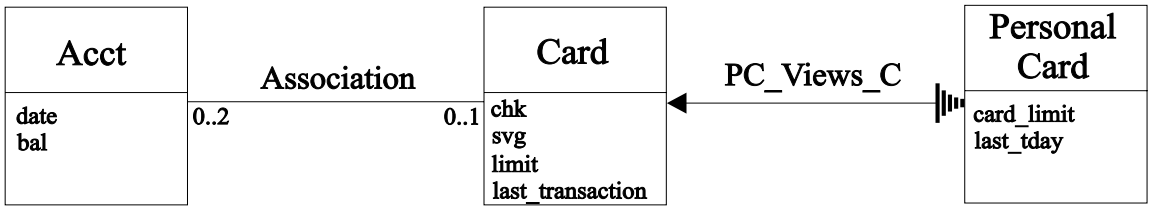


Figure 4.1: Model of a banking application

Figure 4.1 depicts an object model of a simple banking system and it is a good illustration of the limited use we make of semi-formal notations. This graphic

informally represents the object classes and relationships of a banking application that we will be using throughout this chapter.

The application is a simplified model of the convenience card issued to a bank's client for simple transactions using an automatic teller machine (ATM). In our model, we consider a logical card to be uniquely identified by a card number, and personal cards as physical objects that enable access to a logical card. Each convenience card number may correspond to several personal cards. A personal card may be issued to each member of a joint banking account, and all the account tenants will share the same card number (i.e. a logical card) and operate over the banking accounts as a single entity.

Each account is associated with at most one logical card. On the other hand, a logical card may be associated with at most two accounts. In case exactly two accounts are associated with a card number they will be of different types, i.e. one of the accounts will be of *checking* type, while the other account will be of *savings* type. Transactions executed from a logical card will trigger corresponding events in either a checking or a savings account.

As shown in Figure 4.1, we represent accounts as objects of *Acct* class, logical cards as objects of *Card* class, and personal cards as objects of *Personal Card* class. *Acct* is related to the *Card* class by means of a two-to-one association relationship, and the *Personal Card* class is related to *Card* by means of a *views* relationship.

Our banking model uses only two kinds of object-oriented relationships. However, in later sections we will formally approach some of the other popular types of relationship supported by the UML notations, including aggregation and inheritance.

4.1.2 Formal Specification of the *Views*-Based Application

The object-oriented concepts of our banking application will be translated and refined into a theory-based formal specification. Consistent with the previous chapter, when the *views* concepts were formally introduced, the formal system adopted here consists of an object calculus theory based on logic and a categorical framework that combines the smaller theories into a composite system.

Our formal specifications will be developed and analyzed using the integrated verification environment called PVS.[1] The formal language supported by PVS is based on a higher-order logic (HOL) with a rich type system. Thus, we are using HOL to model the behavior of the objects in the banking application, instead of the temporal operators as used in the previous chapter. HOL formulae are also used to express the properties to be validated.

Another differentiation we use from the formal framework previously used is the way in which theories are connected. In the previous chapter, category theory provided the mathematical basis to interconnect the elements in distinct object specifications. The PVS language, however, does not directly support the categorical constructs. Nevertheless, the PVS language provides an *import* and *export* mechanism which allows names declared in one theory to be made available to another theory importing the first one. Thus, while PVS does not use morphisms to make names of one theory known to another, it does use an import/export mechanism that is as effective for representing access to elements in other theories.

[1] In particular, PVS Version 2.2, September 1998 - http://www.csl.sri.com/pvs.html

4.1.3 Related Work

Several publications incorporate object-oriented concepts into formal specification languages. Bicarregui *et al.* [BLM97a] use an object calculus based on temporal logic to describe formally the elements of Syntropy [CD94], which is an object-oriented analysis and design methodology similar to OMT [RBP+91]. The object theories in this approach are connected using category theory. DeLoach, Bailor and Hartrum [DBH95] use an algebraic formalism to specify theories for the classes defined in OMT. This approach also uses category theory to connect the object theories in the formal specification. In addition, there are Z extensions [MC92, AG94, LH94, C+90] which provide enhanced structuring techniques for object specification.

4.2 Using a Verification System

For the current banking application being investigated, formal specification and reasoning is done in the context of the PVS integrated environment [SORSC98c]. Thus, the semantics of the objects composing the application are initially specified in HOL, and later analyzed using the tools available in the PVS system.

The analysis process consists of parsing the theories for syntactic consistency, typechecking the formal specification with the PVS typechecker, and reasoning about the model with the theorem prover tool. While the first two steps check the specification for syntactic and semantic consistency, reasoning supports proof of semantic correctness.

As mentioned by Lamport, the correctness of an algorithm means that the program satisfies the desired property [Lam91]. In our framework, logic is the

formal language used to define programs or properties. Thus, the correctness of our specification means that the formulae defining an object theory in the system specification logically implies the formula specifying the property to be verified.

The PVS environment provides several tools to support the formal development of models. However, we will limit ourselves to the discussion of the elements essential to our application. In this section we introduce a few characteristics of the PVS language [SORSC98a] and the prover facility [SORSC98b] and describe their use in our framework system.

4.2.1 The PVS Specification Language

PVS specifications are built from theories. Each of these theories is identified by a name that can be used to reference declarations inside the theory. However, as in our formal framework system each class definition is represented by a distinct theory presentation, we will need not only to reference the elements inside the theory specification, but we should also be able to say that, for example, an object instance is of a given class sort. As the theory identifiers do not represent a sort, we will define a class sort declaration with the same name as the theory for every theory presentation describing a class. For example, together with the *Card* theory we will also declare a sort named *Card*, so that we are able to refer to the elements of the theory, as well as define objects of a class sort *Card*.

As mentioned in Section 3.2, an object theory consists of a signature and set of axioms that specify the semantics of the signature operations. The theory signature is composed of a class sort (as defined in the previous paragraph), other sorts for the types referenced in the theory, functions, attributes, methods, and events.

Most of the types used in our theory specification are declared as nonempty types. We require the existence of at least one element of a type to avoid the burden

of proving additional conditions on the specifications. Empty types are tolerated as long as only variables range over them. However, declaring a constant of an empty type leads to an inconsistent specification. Thus, if we do not constrain a given type to be nonempty and, in addition, declare a constant of such type, PVS will generate an existence *type-correctness condition* (TCC) [SORSC98a] to guarantee that the type is nonempty. All TCCs should be proved correct before reasoning about any other theory property. PVS, nevertheless, is able to prove automatically many of the TCCs generated.

In each of the theory specifications, an attribute is represented as a function that returns the value of the data held by an object. These functions cannot modify those values in any way. Methods are also defined as functions that may modify the values of an attribute. Events, together with the attributes, represent the interface of the class. They usually invoke internal methods of the class or represent some condition to an external object. In most of our class specifications, we do not differentiate the concepts of methods and events. They are both comprehended by the *action* concept.

We use axioms inside a class sort specification to define the semantics of operations. However, the axiomatic style of specification can be rather dangerous, as inconsistent axioms may allow us to prove that, for instance, $true = false$. A different style of specifying semantics is to state the values of an operation as a definition rather than using axioms. The advantage of this style is that PVS is able to check for possible inconsistencies in the function definitions. Nevertheless, axioms will be needed whenever it is necessary to constrain function values. Since in our case study we frequently need to specify constraints on the values of a function, we opted to use the axiomatic style of specification uniformly.

Another feature of the PVS language often employed is the use of free variables

in axioms. PVS automatically interprets free variables that have been previously defined with the "VAR" construct as universally quantified at the outermost level. Thus, when using a name previously defined as a variable of a given type, we do not need to use the "FORALL" quantifier.

While the axioms of an object specification allow us to reason about the internal structure and of an object class, it does not provide the elements to reason about the relationship between the object classes. In order to specify and reason about properties of a relationship between distinct objects, the elements of these objects should be visible to the relationship theory. While in the framework used in the previous chapter category theory provided the formal basis to access elements in the object specifications, in our PVS specifications the *import* and *export* mechanisms will provide the access to names specified outside the relationship theory.

An "EXPORTING" clause specifies all the names in the theory presentation that are to be made visible to the outside. This clause is optional, which means that, if omitted, every element inside the theory aside from the variables will be made visible to other theories importing it. An "IMPORTING" clause imports all the visible names of another theory. In our case study, only a few of the theory elements will have to be referenced from outside the theory. But, for simplicity, we omit the exporting clause from all theory presentations. Thus, every element of every theory, variables excepted, will be visible from the importing theories.

4.2.2 The PVS Prover

Among the major advantages of the use of formal specifications is that we can analyze and reason about them. Verification systems, such as PVS, provide mechanical support to the formal analysis of design specifications which are still in the early

stages of development. In such early stages, attempted proofs of desirable properties on the specification may reveal subtle errors in design that are often very costly to detect and fix in later stages of development.

The desirable properties of the specification are introduced in the theory specification as conjectures or theorems. In a theory presentation, conjectures are defined by formulae with syntax similar to the axiom formulae. In PVS, conjecture formulae may be introduced with a number of keywords. In our case study we will use the "CONJECTURE" keyword to identify a property that needs to be proved. The free variables in conjecture formulae, identically as with axiom formulae, are universally quantified.

The use of a verification system in the reasoning process has a number of advantages if compared to manual proofs. These systems provide more readable proofs by means of commands that have well defined syntax and semantics. Such proofs can be easily saved or partially saved for future replications of the proving process. The automated system is also able to perform mechanical tasks quickly that are usually tedious and prone to errors if performed manually. Such reliability is particularly important for long proofs, where small errors may be very hard to find.

The PVS verification environment provides a large collection of proof commands that simplifies the whole proving process. These commands are classified in several groups of rules, with some of them being propositional rules, induction rules, equality rules, quantifier rules, flow control rules, help, and many others [SORSC98b]. These commands are also combined by PVS to form a large variety of proof strategies. Such strategies aim an effective automation of common sequences of proof steps. In our proofs we only use a small subset of all the proof commands and strategies supported by PVS.

4.3 Object Theories

In this section we begin the formalization of our small banking application by describing object theories. As already mentioned, some of the specification mechanisms used in the formal framework described in the previous chapter are adapted to a better use of the PVS environment. These changes were adopted because of the different characteristics of the PVS specification language and also as an effort to keep the specification from escalating. As we will see throughout this chapter, the changes in the formal framework are not substantial, and the formalization process will remain consistent with the tools and concepts introduced in the previous chapter.

In the previous chapter, a class theory was specified as a composition of all the class instance theories and a class manager theory, as shown in Figure 3.4. While the former specify theories for the objects of the class, the latter creates and destroys instances of the class and it also represents the synchronization mechanism with other object theories. In our case study, all of these theories are combined and shown as a single class theory. The reason for this combination is to avoid an increase in the number of theories with little contribution to our purposes. This procedure is better shown in Figure 4.2, which is an adaptation of Figure 3.7.

A class theory presentation contains type declarations, attributes, actions, and axioms. Some of these elements are domain specific, while others are characteristic of every class theory. For instance, every class theory should have an action that creates an object instance of that class. Thus, to avoid repeated declarations of semantically identical elements, we define a generic class theory with elements that should be contained in every class specification. In other words, the generic class theory will be imported by every class theory presentation.

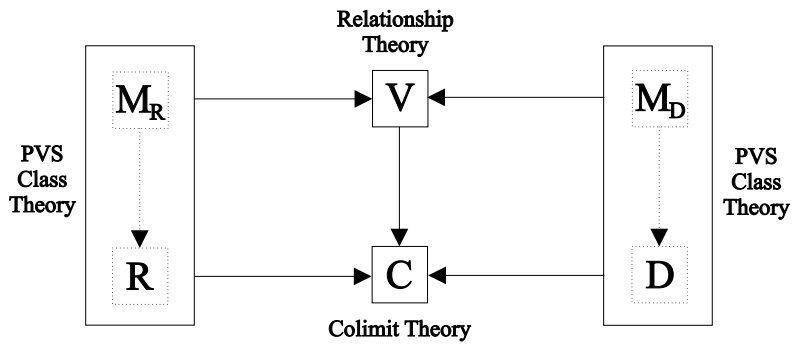


Figure 4.2: A model for the composition of theories

At this point, it is important to mention that the generic class theory should not be confused with the class manager theory concept, which was presented in the previous chapter. The generic class theory is just an artifice to avoid repeated declarations, while the class manager theory creates and destroys object instances, and also synchronizes the class theory with other class or relationship theories. The generic class theory is presented next.

4.3.1 A Generic Class Theory

The intent of the generic class theory is to specify the basic elements to support creation and destruction of object instances of a given class. As mentioned in Section 3.3, if X is a class type, then the $@X$ sort represents the set of all identifiers for instances of X, and $\overline{X}$ represents the set containing the identifiers of all currently existing (i.e. alive) instances of X. Thus, whenever an object instance of X is *killed*, it also means that its identifier has been removed from the set $\overline{X}$. Similarly, *creating* an object also means that one of the identifiers in $@X$ is now also a member of the $\overline{X}$ set. Such properties are stated in axioms 3.1 and 3.2.

In our PVS class specifications, an attribute named *status* identifies the current

condition of every object. An instance of a class X has the value of attribute *status* equal to *alive* only if the object identifier belongs to $\overline{X}$. Alternatively, if such attribute value is *dead*, it means that the object identifier is not in $\overline{X}$. *Alive* and *dead* are the only possible values of the *status* attribute, as defined in the specification of the enumerated type *Obj_Status*, shown in the theory of Schema 1 that follows.

The axioms specifying the semantics of the *create_object* and *kill_object* actions of a class reflect the attribute properties mentioned in the previous paragraph. Thus, if an object o is the result obtained from the action $create_object(o1)$, it means that the status of o is *alive* and the status of $o1$ is *dead*. Note that while axioms 3.1 and 3.2 use temporal logic operators to represent object creation and destruction properties, they are semantically consistent to the axioms defined in the generic class theory in Schema 1. As already mentioned earlier in this chapter, some adjustments in the formal framework will be made to adapt our theories to the PVS specification style.

Schema 1

```
Object[Obj : NONEMPTY_TYPE] : THEORY
  BEGIN

    Obj_Status : TYPE  =  {alive, dead}

    %% Theory Attributes
    status : [Obj → Obj_Status]

    %% Theory Actions
    create_object : [Obj → Obj]
    kill_object : [Obj → Obj]

    %% Theory Axioms
    o, o1 : VAR Obj

    OAx1 : AXIOM
       o = create_object(o1) ⇒ status(o1) = dead ∧ status(o) = alive

    OAx2 : AXIOM o = kill_object(o1) ⇒ status(o) = dead ∧ status(o1) = alive

  END Object
```

Schema 1

Another characteristic of the Schema 1 specification is the use of theory parameters. These parameters may be types, subtypes, or constants, and they provide support for universal polymorphism [SORSC98a]. In this particular case, the parameter is used to support polymorphism for the attribute and action specifications that will be imported by distinct class theories. In each of these theories the value returned by the imported action (e.g. *create_object*) will be of a different type. As seen on the theory presented on Schema 1, the returned value is the polymorphic object being created or killed.

Note also that commenting in PVS specifications is done by placing a "%" character in front of a comment string. The comment is terminated by the end of the line. In our schemas, a comment such as "% ..." indicates that the specification is complemented in the next (or previous) page.

4.3.2 Specification of Object-Oriented Theories

Before we introduce some of the theories of our banking system specification, we define a few types that will be used in most of the class and relationship specifications henceforth. These types are specified within the PVS theory called *BasicTypes*, which is shown in Schema 2.

Schema 2

```
BasicTypes : THEORY
  BEGIN

    Amnt : TYPE  = int
    Date : NONEMPTY_TYPE
    Life_Status : TYPE  = {alive, dead}

    %% Class Types
    Acct : NONEMPTY_TYPE
    Card : NONEMPTY_TYPE
    Personal_Card : NONEMPTY_TYPE

  END BasicTypes
```

Schema 2

Amnt is declared as an alternative name for the type of integers. Because PVS uses structural equivalence, rather than name equivalence, the *int* type is actually equivalent to the *Amnt* type. *Date* is an uninterpreted type declaration, as no assumptions other than being non-empty and disjoint from any other type are made on this type. The abstraction power provided by uninterpreted types is very important in the verification of the type consistency of a system. In addition, by allowing the specifier to omit details on the types, this technique avoids early commitments on implementation strategies. *Life_Status* is the enumerated type used to represent the current status of a relationship instance. This type is actually equivalent to the *Obj_Status* type which was previously defined.

Acct, *Card*, and *Personal_Card* are names of classes that are used whenever we need to declare the type of an object. These three class names declared in

the *BasicTypes* theory will be used again to name the theories in which the class properties are actually specified. By allowing theories to have the same name as types, PVS avoids naming confusion for the object-oriented definitions in our system specification.

The specification of the *Acct* class is the first of three classes in our banking system. Objects of this class represent bank accounts with only some of the elementary characteristics, such as a balance information and deposit or withdrawal operations. The theory presentation of the *Acct* class is shown in Schema 3.

An account object has four attributes. The attribute *bal* records the current balance of a client, while *date* stores the date of creation of an account. An attribute called *acct_type* specifies whether the account mode is checking, savings, or other. Other objects interacting with an account object may use this attribute to allow or disallow certain types of banking operations. *Acct_status* is an attribute derived from *bal*. The constraints on the values of *acct_status* are defined in the axioms of the theory.

Earlier in this chapter (Section 4.2.1) we mentioned that the *action* concept would comprehend both methods and events. However, for the *Acct* class specification only, we distinguish events from methods. Incoming events are the elements of an object triggered from the outside to invoke a method. However, only methods are capable of modifying attribute values. *Deposit* and *withdrawal* are the events of an account object, while *credit* and *debit* are the methods. Among the reasons for separating the two concepts is that the triggering of an *withdrawal* request will not always incur in the invocation of *debit*. For instance, *debit* will not be triggered whenever the account balance is negative.

With the higher-order logic specification style of PVS, the representation of state transitions is not as easy and evident as with the temporal logic style. An action

specification is a function that takes the "old" identity of an object as a parameter and, as the result, returns the object with a "new" identity and new attribute values. Therefore, we use the *equivalent?* predicate in the *Acct* class specification to represent the invocation of a method by an event. This predicate guarantees that, whenever appropriate, the attribute values of the new object returned by the event (e.g. *deposit*) are equivalent to the attribute values of the object returned by the method (e.g. *credit*). Axiom *AAx1* in Schema 3 defines the properties of the *equivalent?* predicate.

Axioms *AAx2* to *AAx4* define additional constraints on the creation of *Acct* objects. These constraints are specified over the method *create_object* that was imported from the *Object* theory. Axioms *AAx5* to *AAx8* define how the methods of the theory modify (or not) the values of attributes *bal* and *date*. Axioms *AAx9* and *AAx10* specify how the value of the derived attribute *acct_status* is obtained from the attribute *bal*. Finally, Axioms *AAx11* to *AAx16* define the pattern of invocation between events and methods of the theory.

While the set of axioms in the *Acct* theory presentation is not complete[2], it does represent a meaningful subset of a simple banking account class. As will be seen later in this chapter, these axioms allow us to prove a few important properties about the theory.

[2] A design specification does not have to be complete. The specification can be incomplete and meaningful, as opposed to implementation.

Schema 3

```
Acct : THEORY
  BEGIN

    IMPORTING BasicTypes, Object[Acct]

    Acct_Type : TYPE  =  {checking, savings}
    Acct_Status : TYPE  =  {ok, overdrawn}

    %% Theory Functions and Attributes
    equivalent? : [Acct, Acct → bool]
    date : [Acct → Date]
    bal : [Acct → Amnt]
    acct_type : [Acct → Acct_Type]
    acct_status : [Acct → Acct_Status]

    %% Theory Actions
    credit : [Acct, Amnt → Acct]
    debit : [Acct, Amnt → Acct]

    deposit : [Acct, Amnt → Acct]
    withdrawal : [Acct, Amnt → Acct]

    %% Theory Axioms
    a, a1 : VAR Acct
    x : VAR Amnt

    AAx1 : AXIOM
       equivalent?(a, a1) = (date(a) = date(a1)) ∧ (bal(a) = bal(a1)) ∧
         (acct_type(a) = acct_type(a1)) ∧ (acct_status(a) = acct_status(a1))

    AAx2 : AXIOM ∃ (d : Date) : date(create_object(a)) = d

    AAx3 : AXIOM bal(create_object(a)) = 0

    AAx4 : AXIOM acct_status(create_object(a)) = ok

    AAx5 : AXIOM bal(credit(a, x)) = bal(a) + x

    AAx6 : AXIOM bal(debit(a, x)) = bal(a) − x

  % ...
```

Schema 3

Schema 3

% ...

AAx7 : AXIOM $\text{date}(\text{credit}(a, x)) = \text{date}(a)$

AAx8 : AXIOM $\text{date}(\text{debit}(a, x)) = \text{date}(a)$

AAx9 : AXIOM $(\text{acct_status}(a) = \text{ok}) \Leftrightarrow \text{bal}(a) \geq 0$

AAx10 : AXIOM $(\text{acct_status}(a) = \text{overdrawn}) \Leftrightarrow \text{bal}(a) < 0$

AAx11 : AXIOM
$\text{acct_status}(a) = \text{ok} \Rightarrow$
$\text{acct_status}(\text{deposit}(a, x)) = \text{ok} \wedge$
$\text{equivalent?}(\text{deposit}(a, x), \text{credit}(a, x))$

AAx12 : AXIOM
$\text{acct_status}(a) = \text{overdrawn} \wedge \text{bal}(a) + x \geq 0 \Rightarrow$
$\text{acct_status}(\text{deposit}(a, x)) = \text{ok} \wedge$
$\text{equivalent?}(\text{deposit}(a, x), \text{credit}(a, x))$

AAx13 : AXIOM
$\text{acct_status}(a) = \text{overdrawn} \wedge (\text{bal}(a) + x) < 0 \Rightarrow$
$\text{acct_status}(\text{deposit}(a, x)) = \text{overdrawn} \wedge$
$\text{equivalent?}(\text{deposit}(a, x), \text{credit}(a, x))$

AAx14 : AXIOM
$\text{acct_status}(a) = \text{ok} \wedge \text{bal}(a) \geq x \Rightarrow$
$\text{acct_status}(\text{withdrawal}(a, x)) = \text{ok} \wedge$
$\text{equivalent?}(\text{withdrawal}(a, x), \text{debit}(a, x))$

AAx15 : AXIOM
$\text{acct_status}(a) = \text{ok} \wedge \text{bal}(a) < x \Rightarrow$
$\text{acct_status}(\text{withdrawal}(a, x)) = \text{overdrawn} \wedge$
$\text{equivalent?}(\text{withdrawal}(a, x), \text{debit}(a, x))$

AAx16 : AXIOM
$\text{acct_status}(a) = \text{overdrawn} \Rightarrow$
$\text{acct_status}(\text{withdrawal}(a, x)) = \text{overdrawn} \wedge$
$\text{equivalent?}(\text{withdrawal}(a, x), a)$

END Acct

Schema 3

A second class theory of our banking system is called *Card*. This theory formalizes a few properties available in a typical convenience card used to perform banking transactions in automated teller machines (ATMs). Each *Card* object may perform transactions over up to two distinct banking accounts. These two accounts are identified inside the card object by the attributes *chk* and *svg*.

Four different transaction types are supported by the *Card* class. These types range from Withdraw From Checking (WFC) to Deposit In Savings (DIS) as specified in the enumerated type *TransactionType*. For simplicity, in the *Card* theory presentation shown in Schema 4, we only specify the semantics of transaction types WFC and DIS. The semantics of the other transaction types may be easily inferred from those two defined types.

Card holders are allowed to withdraw up to a certain amount from all of their banking accounts daily. This amount is identified by the constant *DAILY_MAX*. The transaction will fail (i.e. no attributes are changed) whenever this daily amount is exceeded. The date of the last transaction is used to verify whether previous withdrawals were performed within the same day or the transaction is the first of the day. In case the former is true, the current withdrawal limit will be updated from the *current_limit* attribute. In the latter case, the current limit will be updated from *DAILY_MAX*. These withdrawal properties are defined by Axioms *CAx2* and *CAx3*.

The semantics of the DIS transactions are specified in Axiom *CAx4*. In this type of transactions, the *put* action is triggered to store an amount x in the savings account. It will be shown later, in a relationship theory, how actions *put* (*Card* class) and *deposit* (*Acct* class) interact to modify the *balance* attribute of an account as a result of a card transaction.

Schema 4

```
Card : THEORY
  BEGIN

    IMPORTING BasicTypes, Object[Card]

    TransactionType : TYPE  =  {WFC, WFS, DIC, DIS}
    DAILY_MAX : Amnt

    %% Theory Functions and Attributes
    equivalent? : [Card, Card → bool]
    chk : [Card → Acct]
    svg : [Card → Acct]
    current_limit : [Card → Amnt]
    last_transaction : [Card → Date]

    %% Theory Actions
    transaction : [TransactionType, Date, Amnt, Card → Card]
    put : [Acct, Amnt → Acct]
    get : [Acct, Amnt → Acct]
    notify_balance : [Acct → Amnt]

    %% Theory Axioms
    c, c1 : VAR Card
    d : VAR Date
    x : VAR Amnt

    CAx1 : AXIOM
       equivalent?(c, c1) = (chk(c) = chk(c1)) ∧
         (svg(c) = svg(c1)) ∧ (current_limit(c) = current_limit(c1)) ∧
           (last_transaction(c) = last_transaction(c1))

    CAx2 : AXIOM
       c1 = transaction(WFC, d, x, c) ∧ d = last_transaction(c) ⇒
         IF x ≤ current_limit(c) THEN
           chk(c1) = get(chk(c), x) ∧
             svg(c1) = svg(c) ∧
               current_limit(c1) = current_limit(c) − x ∧
                 last_transaction(c1) = last_transaction(c)
         ELSE equivalent?(c, c1)
         ENDIF

% ...
```

Schema 4

```
% ...                                                                  Schema 4

    CAx3 : AXIOM
      c1 = transaction(WFC, d, x, c) ∧ ¬(d = last_transaction(c)) ⇒
       IF x ≤ DAILY_MAX THEN
        chk(c1) = get(chk(c), x) ∧
         svg(c1) = svg(c) ∧
          current_limit(c1) = DAILY_MAX − x ∧
           last_transaction(c1) = d
       ELSE equivalent?(c, c1)
       ENDIF

    CAx4 : AXIOM
      c1 = transaction(DIS, d, x, c) ⇒
       chk(c1) = chk(c) ∧
        svg(c1) = put(svg(c), x) ∧
         last_transaction(c1) = d ∧
          IF d = last_transaction(c) THEN current_limit(c1) = current_limit(c)
           ELSE current_limit(c1) = DAILY_MAX
           ENDIF

  END Card
                                                                       Schema 4
```

Note that the action *notify_balance* could be used in the Axioms *CAx2* and *CAx3* to produce a different behavior for withdrawal transactions over accounts with balance lower than zero. Also, we could decrease the withdrawal limit by the difference in balance between accounts *chk(c)* and *chk(c1)* instead of decreasing it by x. These modifications would provide unchanged withdrawal limits for unsuccessful withdrawal transactions. However, we opted for the current specification to avoid additional complexity in the theory presentation.

4.4 Relationship Theories

Relationships between class theories are represented as middle theories that synchronize the behavior of the objects of those classes. In this section we use PVS

to describe formally some of the properties associated with the different kinds of object-oriented relationship. While different methods and notations do not converge on a standard set of relationship types, we use the UML semi-formal relationship types as the basis to our formalization effort. To this small UML set of relationship types we add the *views* relationship, the formalization of which was one of the foci of the previous chapter.

UML defines semantics for both structural and behavioral object models. While the structural model represents the static aspects of a system, the behavioral model defines the dynamic properties of such system. Behavioral properties in UML are mostly represented by means of collaboration diagrams and state machines. The translation of these two behavioral notations into the PVS-based theories should not be a complicated task. In fact, Lano and Bicarregui [LB98b, LB98a] formally describe part of dynamic notations of UML using temporal theories. In addition, DeLoach and Hartrum [DH99] describe some transformations of statecharts into an algebraic formalism. However, our focus will be on the specification of the structural properties of a model, even though some of the behavioral properties of the system are also specified in the theories.

4.4.1 Formalization of an Association

The semantics of an association relationship varies among objet-oriented methods. Each of these methods use a number of properties to characterize different types of association. In this section we present a simple association relationship theory which connects the *Acct* and *Card* object theories. The name of the theory is *Association* and it is presented in the Schema 5 shown next. In addition, we also mechanize the formal use of UML association attributes which were formally presented in the previous chapter.

Schema 5

```
Association : THEORY
  BEGIN

    IMPORTING BasicTypes, Acct, Card

    Association : NONEMPTY_TYPE

    %% Theory Attributes
    status : [Association → Life_Status]
    image : [Association, Card → Acct]
    image : [Association, Acct → Card]

    %% Theory Actions
    create_assoc : [Card, Acct → Association]
    kill_assoc : [Association → Association]

    %% Theory Axioms
    s, s1 : VAR Association
    a, a1 : VAR Acct
    c, c1 : VAR Card
    x : VAR Amnt

    SAx1 : AXIOM a = image(create_assoc(c, a), c)

    SAx2 : AXIOM a = image(s, c) ⇔ image(s, a) = c

    SAx3 : AXIOM
       a = image(s, c) ∧ status(s) = alive ⇒
         status(a) = alive ∧ status(c) = alive

    SAx4 : AXIOM s = create_assoc(c, a) ⇒ status(s) = alive

    SAx5 : AXIOM s1 = kill_assoc(s) ⇒ status(s) = alive ∧ status(s1) = dead

    SAx6 : AXIOM
       chk(c) = a ∨ svg(c) = a ⇒
         (∃ (s : Association) : image(s, a) = c)

    SAx7 : AXIOM chk(c) = a ⇒ acct_type(a) = checking

    SAx8 : AXIOM svg(c) = a ⇒ acct_type(a) = savings
% ...
```

Schema 5

Schema 5

% ...

SAx9 : AXIOM $\text{chk}(c) = a \vee \text{svg}(c) = a \Rightarrow \text{bal}(a) = \text{notify_balance}(a)$

SAx10 : AXIOM $\text{chk}(c) = a \vee \text{svg}(c) = a \Rightarrow \text{deposit}(a, x) = \text{put}(a, x)$

SAx11 : AXIOM $\text{chk}(c) = a \vee \text{svg}(c) = a \Rightarrow \text{withdrawal}(a, x) = \text{get}(a, x)$

SAx12 : AXIOM
$a = \text{image}(s, c) \wedge a_1 = \text{image}(s_1, c) \Rightarrow$
$a = a_1 \vee \text{acct_type}(a) \neq \text{acct_type}(a_1)$

SAx13 : AXIOM
$a = \text{image}(s, c) \wedge a = \text{image}(s_1, c_1) \Rightarrow s = s_1 \wedge c = c_1$

END Card

Schema 5

The *Association* theory defines additional constraints over the attributes and actions of the *Acct* and *Card* theories. Each instance of the association (i.e. a link) connects a single instance of *Acct* to another instance of *Card*. Within every link, each object instance is said to be the image of the other object being connected. The *image* attributes identify the image of every object participating in an association.

The *status* attribute assists in the definition of semantics for the creation and destruction of each association link. Like a typical object instance, the status of a link may be either *alive* or *dead*. The value of this attribute may be changed only by the *create_assoc* and *kill_assoc* actions.

A number of axioms defines the semantics for the *Association* theory. Axioms *SAx1*, *SAx2*, and *SAx3* specify constraints over the *image* attributes. One of the constraints, defined by Axiom *SAx3*, is that the objects interconnected by an active association instance should be alive too. Axiom *SAx2* states that the *image* attributes represent a symmetric relation between objects of *Acct* and *Card*. Ad-

ditionally, *SAx4* and *SAx5* are axioms defining pre- and post-conditions for the occurrence of actions *create_assoc* and *kill_assoc*.

The last eight axioms defined in this interpretation theory represent domain-specific constraints. Most of these axioms also use elements of the object theories being interconnected to specify constraints over the association. *SAx6* defines conditions on two attributes of the *Card* class. *SAx7* and *SAx8* associate values of attributes in *Card* with attributes in *Acct*. Axioms *SAx9*, *SAx10* and *SAx11* interconnect actions of instances of both classes. In other words, the triggering of an action in one object will imply the triggering of another action in the associated object. Axiom *SAx12* specify conditions on the types of accounts associated with the same *Card* object. Finally, *SAx13* defines that an account object will be connected to at most one card object.

There are several properties which may be inferred from the set of *Association* axioms in Schema 5. Such properties are useful to check whether the specification represents the association correctly. For instance, the system requires that the *savings* account in a given card be different from the *checking* account of that same card. Formally, this is stated as:

$$chk(c) = a \wedge svg(c) = a1 \Rightarrow a \neq a1$$

The above property can be easily verified from axioms *SAx7* and *SAx8*. However, details on the proof of the banking system properties will be considered only later in this chapter. Now, we turn our attention to the mechanization in PVS of the properties represented by UML associations attributes. These properties were formally introduced in Section 3.6.1 by means of temporal logic axioms.

Schema 6

$\text{AssociationEnd} : \text{TYPE} = \{\text{AcctEnd}, \text{CardEnd}\}$
$\text{AggregationKind} : \text{TYPE} = \{\text{none}, \text{aggregate}, \text{composite}\}$

$\text{aggregation} : [\text{AssociationEnd} \rightarrow \text{AggregationKind}]$

$s, s_1 : \text{VAR Association}$
$a : \text{VAR Acct}$
$c, c_1 : \text{VAR Card}$

$\text{ESAx1} : \text{AXIOM aggregation(CardEnd)} \neq \text{none} \Rightarrow \text{aggregation(AcctEnd)} = \text{none}$

$\text{ESAx2} : \text{AXIOM}$
$\quad \text{aggregation(CardEnd)} \neq \text{none} \Rightarrow \text{status}(a) = \text{alive} \Leftrightarrow$
$\quad\quad (\exists\ (c : \text{Card}) : a = \text{image}(s, c) \wedge \text{status}(s) = \text{alive})$

$\text{ESAx3} : \text{AXIOM}$
$\quad \text{aggregation(CardEnd)} = \text{composite} \wedge$
$\quad\quad a = \text{image}(s, c) \wedge a = \text{image}(s_1, c_1) \Rightarrow c = c_1 \wedge s = s_1$

Schema 6

Aggregation

Note that the *Association* theory shown in Schema 5 formalizes a corresponding UML binary association where the *aggregation* attribute values of both ends of the association are equal to *none*. Consequently, that relationship theory represents a regular association with no additional constraints needed to specify. However, in this section we want to illustrate the specification of an aggregation form of association. Hence, we extend Schema 5 with additional constraints that characterize the aggregation relationship.

According to the UML meta-model illustrated in Figure 2.4, a binary association is composed of two association ends. In our mechanization, we define the attributes and properties of both of these association ends as an extension to the *Association* theory presented in Schema 5. The association theory extensions pertinent to the *aggregation* attribute are presented in Schema 6. Theory extensions related to the

other UML association end attributes follow.

Aggregation is defined as an attribute with three possible values: *none*, *aggregate*, and *composite*. In Schema 6, aggregation is specified as an *AggregationKind* type of attribute, where *AggregationKind* is an enumerated type containing those three values. Note that the *aggregation* attribute declaration has no parameter to identify a relationship instance. Alternatively, all of the previous attribute specifications of the theory (see Schema 5) have a parameter of type *Association* which identifies different attribute values for different relationship instances. The reason for omitting an *Association* parameter in the current case is that the value of the *aggregation* attribute is the same for every instance of the association theory. This will also be the case for every other UML-related attribute defined in this chapter.

The three PVS-based Axioms *ESAx1*, *ESAx2*, and *ESAx3* specify additional constraints which are effective whenever the aggregation value of the *Card* end of the association is different than *none*. These axioms represent properties which are equivalent to the ones defined in Chapter 3 by means of Axioms 3.10, 3.11, and 3.12, respectively. To avoid the repetition of properties, we do not consider the case where the *Acct* end of the association is an aggregation.

Changeable

The attribute *changeable* specifies rules for the creation and removal of association instances related to an object. Like the UML *aggregation* attribute, *changeable* also had its semantics formally defined in the previous chapter. The enumerated type *ChangeableKind* defines the possible values for such attribute, and Schema 7 shows the PVS axioms constraining its values.

Axiom *ESAx4* defines the extended relationship semantics of our system for cases where the *CardEnd* side of the association has value *frozen*. This axiom is the

Schema 7

$\text{AssociationEnd} : \text{TYPE} = \{\text{AcctEnd}, \text{CardEnd}\}$
$\text{ChangeableKind} : \text{TYPE} = \{\text{none}, \text{frozen}, \text{addOnly}\}$

$\text{changeable} : [\text{AssociationEnd} \rightarrow \text{ChangeableKind}]$

$s, s_1 : \text{VAR Association}$
$a, a_1 : \text{VAR Acct}$
$c, c_1 : \text{VAR Card}$

$\text{ESAx4} : \text{AXIOM}$
$\quad \text{changeable}(\text{CardEnd}) = \text{frozen} \wedge s = \text{create_assoc}(c, a) \Rightarrow$
$\quad\quad \exists (c_1 : \text{Card}) : c = \text{create_object}(c_1)$

$\text{ESAx5} : \text{AXIOM}$
$\quad \text{changeable}(\text{CardEnd}) = \text{addOnly} \wedge a = \text{image}(s, c) \Rightarrow$
$\quad\quad (\text{status}(s) = \text{dead} \Rightarrow \text{status}(a) = \text{dead} \vee \text{status}(c) = \text{dead})$

Schema 7

PVS correspondent for Axiom 3.13, defined back in Chapter 3. Note, however, that Axiom 3.13 uses temporal logic concepts to specify that every association instance related to one particular object has to be created together with that same object. In PVS, we use the less intuitive artifice in which every change of the object's state is represented by a change in the name used to reference the object. Thus, if an association instance $s : Association$ is created between objects $c : Card$ and $a : Acct$, we require the status of this instance to be the same as the status of c.

Axiom *ESAx5* is the other rule constraining the *changeable* attribute, and it is effective when the *CardEnd* side of the association has value *addOnly*. Axiom 3.14 uses temporal operators to state the semantics of the *addOnly* associations. It formally says that a link will not be removed until one of the objects being connected is killed. Axiom *ESAx5* defines equivalent semantics using PVS logic.

Schema 8

IMPORTING orders[Association]

$\leq$: VAR (partial_order?)

s, s_1 : VAR Association
a, a_1 : VAR Acct
c : VAR Card

ESAx6 : AXIOM
$a = \text{image}(s, c) \wedge a_1 = \text{image}(s_1, c) \wedge \text{bal}(a) \leq \text{bal}(a_1) \Rightarrow s \leq s_1$

Schema 8

IsOrdered

The *isOrdered* attribute of UML works as a flag which indicates the existence or not of an ordering relation for the instances of an association. PVS libraries define a partial order over a type T as being a set of pairs of elements of type T. Such a set should respect the reflexivity, antisymmetry, and transitivity conditions defined in Table 3.1. In our example, which is illustrated in Schema 8, type T is substituted by the *Association* type, and the partial order conditions are imported from theory *orders*[*Association*]. The partial ordering relation is represented by the symbol "$\leq$".

Axiom *ESAx6* illustrates some ordering constraints for an association example with attribute $isOrdered(CardEnd) = true$. According to this axiom, if an object $c : Card$ is connected to two different accounts $a, a_1 : Acct$ by means of *Association* instances s and s_1, and the balance attribute value of account a is less or equal to the balance of account a_1 (i.e. $bal(a) \leq bal(a_1)$), then we say that s precedes or equals $s1$ (i.e. $s \leq s_1$). The precedence of one link over another defines an ordering element over the instances of an association. The set of these elements form an ordering relation.

Multiplicity

This attribute constrains the number of instances of an object theory that may be attached by an association link to one single object at the other end of the relationship. In UML, *multiplicity* is defined as a range of integers inside the interval $[0,\infty)$. In PVS, this range is represented by means of logic axioms defined inside the association theory.

In the *Association* theory of Schema 5, Axioms *SAx12* and *SAx13* define a one-to-two multiplicity constraint for the association theory. Axiom *SAx12* specifies that each card object will be associated with at most two account objects. Note that the axiom states that a card object cannot be associated with two or more accounts of the same type (i.e. savings). Consequently, as only two types of account exist, a single card object will not be connected to more than two accounts at the same time. Alternatively, Axiom *SAx13* states that one account object will not be connected to more than one card object at a time.

IsNavigable and TargetScope

In Section 3.6.1, no additional constraints were introduced in the association theory for both the *IsNavigable* and *TargetScope* attributes. For reasons similar to those indicated in that section, we do not extend the *Association* theory in Schema 5 with new axioms. Thus, the information introduced by these concepts may be represented as theory attributes, which values will be regarded during later development stages.

4.4.2 Inheritance and Subtyping

Subtyping is the mechanism we use to describe inheritance. A class theory B is said to inherit from A, if the class type of B is a subtype of the class type of A, and all the elements of A are also elements of B. A is then called a superclass, while B is called a subclass. In this section, we illustrate the mechanization of the subtyping mechanism in a PVS theory by means of a *SavingsAcct* subclass theory. Such theory is shown in Schema 9.

Schema 9

```
SavingsAcct : THEORY
  BEGIN
   IMPORTING Acct

   SavingsAcct : TYPE FROM Acct
   Percentage : TYPE  =  real

   creditInterest : [Acct, Percentage → Acct]

   a : VAR Acct
   p : VAR Percentage

   CAAx1 : AXIOM bal(creditInterest(a, p)) = bal(a) + (p × bal(a))
   CAAx2 : AXIOM date(creditInterest(a, p)) = date(a)

 END SavingsAcct
```

Schema 9

PVS supports subtyping by means of the syntactic construct "TYPE FROM". In our example, the *SavingsAcct* class type is declared as a subtype of *Acct*. In addition, while morphisms were used in the previous chapter to embed the elements of the superclass inside the subclass, we now use the *importing* mechanism to obtain equivalent results inside PVS. Consequently, all the attributes and actions defined in theory *Acct* are also part of the *SavingsAcct* theory.

As shown in Schema 9, *SavingsAcct* also extends the properties of its superclass

by means of the *creditInterest* action. Such action allows a periodic addition of savings interest to the balance of an account. All other properties inherited from *Acct* remain unchanged.

4.4.3 *Views*

In this section we use the higher-order logic language of PVS to describe the *views* concepts. These concepts are shown in Schema 10, and they define a theory containing some of the properties that should be common to every different specification of a *views* relationship. In the following section, an application of a *views* theory is created and it imports all the basic *views* properties defined in the PVS-based *Views* theory.

Properties 1 and 2 establish that *views* is a relationship between different class types. PVS, however, does not allow the specification of type comparison expressions. Therefore, the constraints defined in those properties should be enforced by each distinct application of a *views* relationship. Such application, which is later illustrated by the *PC_Views_C* theory, will be responsible for identifying which two classes are connected by the *views* relationship. We need to assure that those two classes are not the same.

Property 3 precludes any *views*-cycles in the system. As this property concerns the system as a whole, it should be guaranteed by the colimit of all the theories in the system, and not only within a *views* theory. Our PVS formalism, however, does not support the specification of a general axiom that guarantees the absence of *views*-cycles. Thus, we will only indicate the absence of cycles in our case study.

Another rule to be verified in the colimit of the theories is stated by Property 9. This rule is an extension of Property 4 which states that a viewer cannot participate

Schema 10

Views $[R:$ NONEMPTY_TYPE$, D:$ NONEMPTY_TYPE$, V:$ NONEMPTY_TYPE$]:$ THEORY
BEGIN

IMPORTING BasicTypes, Object$[D]$, Object$[R]$

%% *Theory Attributes*
status : $[V \rightarrow$ Life_Status$]$
viewer : $[V, D \rightarrow R]$
viewed : $[V, R \rightarrow D]$

%% *Theory Actions*
create_view : $[D, R \rightarrow V]$
remove_view : $[V \rightarrow V]$

%% *Theory Axioms*
$r, r_1 :$ VAR R
$d, d_1 :$ VAR D
$v, v_1 :$ VAR V

VAx1 : AXIOM $d = \text{viewed}(v, r) \Leftrightarrow r = \text{viewer}(v, d)$

VAx2 : AXIOM
$d = \text{viewed}(v, r) \wedge \text{status}(v) = \text{alive} \Rightarrow$
$\text{status}(r) = \text{alive} \wedge \text{status}(d) = \text{alive}$

VAx3 : AXIOM
$v = \text{create_view}(d, r) \Rightarrow r = \text{viewer}(v, d) \wedge \text{status}(v) = \text{alive}$

VAx4 : AXIOM
$v = \text{remove_view}(v_1) \Rightarrow \text{status}(v_1) = \text{alive} \wedge \text{status}(v) = \text{dead}$

Property4 : AXIOM
$d = \text{viewed}(v, r) \wedge d_1 = \text{viewed}(v_1, r) \Rightarrow d = d_1 \wedge v = v_1$

Property5 : AXIOM
$\text{status}(r) = \text{alive} \Rightarrow$
$\exists (v : V, d : D) : (d = \text{viewed}(v, r) \wedge \text{status}(v) = \text{alive})$

Property6a : AXIOM
$d = \text{viewed}(v, r) \wedge d_1 = \text{viewed}(v, r_1) \Rightarrow d = d_1 \wedge r = r_1$
% ...

Schema 10

% ... **Schema 10**

Property6b : AXIOM $\text{status}(v) = \text{alive} \Rightarrow \exists\ (r:\ R, d:\ D):\ d = \text{viewed}(v, r)$

Property7 : THEOREM
$d = \text{viewed}(v, r) \Rightarrow (\text{status}(r) = \text{dead} \Leftrightarrow \text{status}(v) = \text{dead})$

Property8 : THEOREM
$d = \text{kill_object}(d_1) \wedge d = \text{viewed}(v, r) \Rightarrow$
$\text{status}(r) = \text{dead} \wedge \text{status}(v) = \text{dead}$

END Views **Schema 10**

in more than one instance of a relationship theory. Alternatively, Property 9 states that a viewer object cannot *view* more than one viewed object in the whole system. Later in this chapter, this property will be formally stated in one of the colimit theories which contain more than one *views* relationship.

The first four axioms of the *Views* theory define some basic semantics which are similar to the semantics introduced by the first five axioms of the *Association* theory, which was previously specified in Schema 5. Axiom *VAx1* defines a symmetry between the *viewer* and *viewed* attributes. For instance, if r is the viewer of an object d within a relationship, then d is viewed by the object r. Axiom *VAx2* defines that the objects interconnected by an active *views* relationship instance should be alive too. *VAx3* and *VAx4* are the axioms defining pre- and post-conditions for the occurrence of actions *create_view* and *remove_view*.

The other axioms in the theory presentation specify the *views* relationship properties previously described in Section 3.7. Axiom *Property4* uses the PVS formalism to specify a cardinality constraint which was previously defined in Axiom 3.18. According to such rule, a viewer object will be related to at most one viewed object. Axiom *Property5* complements such rule by stating that a viewer object will be

interconnected to at least one viewed object. These two axioms together define the cardinality of the *viewed* end of the relationship as being exactly one. The *viewer* end of the *views* relationship has no inherent constraint.

The sixth *views* property states that each relationship instance interconnects exactly one viewer to one viewed object. Axiom *Property6b* guarantees that for every *views* instance there is a pair of viewer and viewed objects which the relationship interconnects. Using Axiom *VAx2*, we also have that both objects in the pair are alive. Axiom *Property6a* establishes that a single *views* instance will not involve more than one viewer and one viewed object. A similar property could also be stated for the *Association* relationship.

Property7 is defined as a theorem, as it may be derived from other axioms of the *Views* theory. The actual proof of the theorem, however, will only be described later in this chapter. The semantics associated with the *Property7* theorem is that a relationship instance v will be dead if and only if the viewer object r it interconnects is also dead. In other words, the life time of v and r should be identical.

Property8 is another theorem of the *Views* theory. The semantics of such a theorem is a constraint over the life time of the interconnected objects. More specifically, the theorem states that the destruction of a viewed object implies the destruction of all the interconnected viewer objects and, consequently, its relationship instances. In the previous chapter, Property 8 was formally described and proved by Theorem 3. Later in this chapter, we develop a similar proof for the corresponding theorem, but this time using the PVS environment resources.

4.4.4 Using *Views* to Relate Object Theories

In Section 4.4.1, an association theory interconnecting classes *Acct* and *Card* was specified. In this section, we define the other relationship theory of the system,

which is called *PC_Views_C*. This relationship illustrates the application of the *views* concepts between classes *Card*, which was defined in Schema 4, and *Personal_Card*, which is here defined in Schema 11. While the former represents the viewed class, the latter is a viewer class representing an interface of the system to the bank customer.

Schema 11

```
Personal_Card : THEORY
  BEGIN

    IMPORTING BasicTypes, Object[Personal_Card]

    %% Theory Attributes
    personal_limit : [Personal_Card → Amnt]
    personal_LTday : [Personal_Card → Date]

    %% Theory Actions
    instant_cash : [Personal_Card, Date, Amnt → Personal_Card]
    transfer2savings : [Personal_Card, Date, Amnt → Personal_Card]

  END Personal_Card
```

Schema 11

Schema 12 introduces the simple *Personal_Card* theory. This class has an attribute *personal_limit*, which defines a withdrawal limit to the card, and an attribute *personal_LTday*, which stores the date of the last transaction performed by the card object being viewed. In addition, two actions characterize the interface of the class. Action *instant_cash* allows the customer to withdraw money from the checking account, while action *transfer2savings* allows this customer to move funds between accounts. All of the attribute values and actions of the *Personal_Card* objects will be constrained by the attributes and actions of the viewed objects. These constraints will be specified in the axioms of the relationship theory.

An application of the *views* concepts is illustrated by the *PC_Views_C* relationship theory described in Schema 12. As stated in the theory, *PC_Views_C* imports

Schema 12

```
PC_Views_C : THEORY
  BEGIN

   PC_Views_C : NONEMPTY_TYPE

   IMPORTING BasicTypes, Card, Personal_Card,
             Views[Personal_Card, Card, PC_Views_C]

   %% Theory Axioms
   c : VAR Card
   x : VAR Amnt
   d : VAR Date
   t : VAR TransactionType
   pc, pc1 : VAR Personal_Card
   v : VAR PC_Views_C

   DAx1 : AXIOM c = viewed(v, pc) ⇒ current_limit(c) = personal_limit(pc)

   DAx2 : AXIOM c = viewed(v, pc) ⇒ last_transaction(c) = personal_LTday(pc)

   DAx3 : AXIOM
      c = viewed(v, pc) ∧ pc1 = instant_cash(pc, d, x) ⇒
       ∃(v1 : PC_Views_C) : viewed(v1, pc1) = transaction(WFC, d, x, c)

   DAx4 : AXIOM
      c = viewed(v, pc) ∧ pc1 = transfer2savings(pc, d, x) ⇒
       ∃(v1 : PC_Views_C) :
        viewed(v1, pc1) = transaction(DIS, d, x, transaction(WFC, d, x, c))

 END PC_Views_C
```

Schema 12

all the attributes, actions, and axioms specified in the *Views* schema. Note that the IMPORTING clause uses *Personal_Card* as the viewer class type, *Card* as the viewed class type, and *PC_Views_C* as the relationship type. This means that the *Views* theory parameters *R*, *D*, and *V* will be replaced during the import by *Card*, *Personal_Card*, and *PC_Views_C*, respectively.

Note also that *views* Property 2 is respected during the import. As in PVS, types which do not have a supertype in common are assumed to be disjoint, *Card* and *Personal_Card* are, consequently, different classes.

Four axioms represent the interaction between viewer and viewed objects. Axioms *DAx1* and *DAx2* guarantee the consistency between the attributes of the related objects. These two axioms constrain the values of the viewer attributes and allow the viewer object to maintain a state consistent with the corresponding viewed object. Axioms *DAx3* and *DAx4* provide semantics for the *Personal_Card* actions. *DAx3* establishes that an *instant_cash* operation at the viewer end of the relationship will trigger a withdrawal transaction from the checking account associated with the viewed object. Finally, *DAx4* defines that a *transfer2savings* operation in a *Personal_Card* object is equivalent to a pair of transactions at the viewed end of the relationship. This pair of transactions moves a specified amount of funds from a checking account to a savings account.

4.5 Colimit Theories

In Section 3.1, a few concepts of category theory were introduced. Among those concepts, the colimit of a number of theories was described as the *amalgamated sum* of those theories. In this section, we use the notion of a colimit, which is the disjoint union of all specifications, to represent a whole system as a single theory.

In other words, the colimit theory will contain the attributes, actions, and axioms of all the object classes specified in the system.

4.5.1 A General Colimit Theory

The PVS-based formalism used in the definition of object theories does not allow the specification of some of the *views* concepts, as already mentioned in Section 4.4.3. The reason is that, in our mechanization formalism, each class is represented by a different type. For instance, the *Acct* class type is disjoint from the *Card* type. As a consequence, we cannot refer to the objects of these types in general terms. In other words, if a is an object of a certain class and c is an object of another class with no superclass in common, then a and c cannot be used together in expressions that require compatibility, e.g. the typechecking of an expression $(a \neq c)$ would fail.

While specifying a *Class* supertype for every definition of a class seems to be the reasonable choice, this solution would add complexity to all the previously defined theories. Just as an example of the complexity introduced, several of the theory axioms would require additional coercion statements to indicate explicitly the *Class* subtype expected in the axiomatic expressions. Therefore, we opted not to sacrifice the readability of the theories, and, instead, illustrate the mechanization of some properties using a general colimit theory that refers to all the objects of the system as instances of a *Class* type.

Schema 13 shows a general colimit theory which specifies some of the *views* properties that were not defined in the *Views* theory of Schema 10. Note that, for example, the *Acct* class in this alternative mechanization formalism would be declared as a subtype of the *Class* supertype.[3] This is inconsistent with what was

[3]The declaration syntax would be: *Acct: TYPE FROM Class*

Schema 13

```
GeneralColimit : THEORY
  BEGIN

    Class : NONEMPTY_TYPE
    VRelationship : NONEMPTY_TYPE

    IMPORTING Views[Class, Class, VRelationship]

    viewerset : [Class → setof[Class]]

    o1, o2, o3 : VAR Class
    v1, v2 : VAR VRelationship

    VSetAxiom : AXIOM
      ∃ (v : VRelationship) :
        o1 = viewed(v, o2) ⇒
          o2 ∈ viewerset(o1) ∧ (o3 ∈ viewerset(o2) ⇒ o3 ∈ viewerset(o1))

    Property1 : AXIOM o1 = viewed(v1, o2) ⇒ o1 ≠ o2

    Property3 : AXIOM ¬ (o1 ∈ viewerset(o1))

    Property9 : AXIOM
      o1 = viewed(v1, o3) ∧ o2 = viewed(v2, o3) ⇒
        o1 = o2 ∧ v1 = v2

  END GeneralColimit
```

Schema 13

actually stated in the class type declarations defined in the *BasicTypes* theory. However, the general colimit theory is being described here independently from the banking system case study.

In this general theory, *VRelationship* represents all the different types of *views* relationship in the system. The IMPORTING statement associates all the properties of the *Views* theory with the *VRelationship* type. Note also, from this statement, that all *views* relationships of the theory are between two *Class* types, even though each particular application of this type of relationship relates two different classes.

The *viewerset* attribute in this general theory represents the set of all the objects which are direct or indirect viewers of a certain object. The semantics associated with this attribute is recursively defined by Axiom *VSetAxiom*. According to such a specification, an object belongs to the set $viewerset(o_1)$ if either it directly *views* the object o_1 or it belongs to the *viewerset* of another object that *views* o_1.

The three other axioms of the theory represent *views* properties. Axiom *Property1* defines that an object cannot view itself. Axiom *Property3* specifies that an object cannot belong to its own *viewerset*. In other words, this property assures that there are no *views*-cycles inside the system. Finally, Axiom *Property9* indicates that each viewer object cannot be related to more than one viewed object.

4.5.2 The Colimit of the Whole

We now define the colimit of the whole application as the composite of all the object and relationship theories of the system. Alternatively, we could have defined the composite of the system as the colimit of a subsystem, which is another colimit theory, and the other objects and relationships of the system. For example, a subsystem of the banking application could be described as the colimit of

the *Acct*, *Card*, and *Association*[4] theories. We could use this smaller subsystem theory to prove properties about some specific part of the whole. *Personal_Card* and *PC_Views_C* theories would later be added to that subsystem to represent the complete application.

Colimit : THEORY	**Schema 14**
BEGIN	
IMPORTING BasicTypes, Acct, Card, Personal_Card, Association, PC_Views_C	
END Colimit	**Schema 14**

Schema 14 shows the composite theory for our case study. While we do not define any additional axioms in such theory, in the following section we use this *Colimit* theory to prove several properties about the complete system specification. Some of these properties are useful in the verification of correctness of the *views* properties specified in Schema 10.

4.6 Proving System Properties

Validation represents one major step in a software lifecycle. Currently, testing is commonly used as the validation technique, as formal techniques remain complex and expensive for the typical developer [KDGN97]. However, many critics argue that the only acceptable way of validating a software is to prove mathematical properties of the system. The reason is that testing covers only a limited number of cases. According to E. W. Dijkstra, *Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence* [Dij72].

[4]The three *ExtendedAssoc* theories could also be added, if necessary.

In this section, we validate the specification of the banking system by proving the correctness of its properties. Such validation is achieved by mathematically proving that the formulae defined in the object theories of the banking system specification logically imply the formulae specifying the validation properties. These validation properties represent the desired characteristics of a system. They are here referred as *conjectures*. In PVS, the keywords CONJECTURE, THEOREM, COROLLARY, CHALLENGE, PROPOSITION, CLAIM, and a few others have the same associated semantics.

We start the verification of our specification with a simple conjecture on the *Acct* theory. The semantics associated with this conjecture is that the balance of an account should remain unchanged after the credit of zero units of currency. The PVS construct for this conjecture is:

$$\text{Conjecture1}: \ \textsc{conjecture} \ \forall\ (a:\ \text{Acct}):\ \text{bal}(\text{credit}(a,0)) = \text{bal}(a)$$

Schema 15 describes in detail all the steps required in the proof of this first conjecture. Note that each proof step is graphically represented by a *sequent*. The formulae above sequent line are called *antecedents*, while the ones below the line are called *consequents*. In the PVS prover environment, each antecedent formula is numbered with a negative integer. Alternatively, each consequent is uniquely identified by a positive integer. Schema 15 shows five sequents with exactly one consequent each. The last three sequents also have one antecedent each. The logical meaning of a sequent is that the conjunction of antecedents implies the disjunction of consequents. Therefore, a sequent is true if any antecedent is *false*, any consequent is *true*, or any antecedent is equivalent to any consequent.

The evolution in the sequents of Schema 15 reflect the application of PVS prover commands. Each of these commands performs one or more verification tasks. The

Verbose proof for `Conjecture1`.

Schema 15

`Conjecture1`:

$$\{1\} \quad (\forall\ (a:\ \text{Acct}):\ \text{bal}(\text{credit}(a,0)) = \text{bal}(a))$$

Skolemizing,

$$\{1\} \quad \text{bal}(\text{credit}(a',0)) = \text{bal}(a')$$

Applying Axiom AAx5

$$\begin{array}{ll} \{\text{-1}\} & (\forall\ (a:\ \text{Acct}, x:\ \text{Amnt}):\ \text{bal}(\text{credit}(a,x)) = \text{bal}(a)\ +\ x) \\ \hline \{1\} & \text{bal}(\text{credit}(a',0)) = \text{bal}(a') \end{array}$$

Instantiating the top quantifier in formula -1 with the terms: a!1, and 0

$$\begin{array}{ll} \{\text{-1}\} & \text{bal}(\text{credit}(a',0)) = \text{bal}(a')\ +\ 0 \\ \hline \{1\} & \text{bal}(\text{credit}(a',0)) = \text{bal}(a') \end{array}$$

Simplifying with decision procedures,

$$\begin{array}{ll} \{\text{-1}\} & \text{bal}(\text{credit}(a',0)) = \text{bal}(a') \\ \hline \{1\} & \text{bal}(\text{credit}(a',0)) = \text{bal}(a') \end{array}$$

which is trivially true.
This completes the proof of `Conjecture1`.

Q.E.D.

Schema 15

combination of a number of commands is called a proof strategy. The semantics of some of the PVS prover commands are described in Appendix A.

Another useful representation of the proof of a conjecture is shown in a Lisp-like representation. This abbreviated form of proof representation is semantically equivalent to the extended sequent-based representation. While compact, this representation requires some understanding of the PVS prover commands. For example, the abbreviated representation for the sequent transformations of Schema 15 is described by the following sequence of PVS commands:

```
("" (SKOLEM!) (LEMMA "AAx5") (INST -1 "a!1" "0") (SIMPLIFY) (PROPAX))
```

In this section, most of our proofs are presented in this abbreviated form. We found it unnecessary to present extended representations for each of our proofs, as the extended form can be automatically generated from the abbreviated form. Nevertheless, the sequent-based form is used a few times to illustrate the correspondence between both forms.

We also categorized the verification formulae according to the type of specification properties it validates. These properties were divided into four categories: framework, *views*, UML, and domain specific. Property proofs on each of those groups are described next.

4.6.1 Framework Properties

We call framework properties the formulae specifying characteristics of the supporting formalisms, such as the object calculus. These properties were specified as axioms and theorems of the general interpretation theories for objects or relationships. Such a general object and relationship theories were previously described in Sections 3.3 and 3.5, respectively.

Our first PVS proof of a framework property refers to Theorem 1, which was also proved in Section 3.3 using temporal logic axioms. The theorem states that an object cannot be created and killed at the same time. This is now stated as follows:

$$\begin{array}{l}\text{Theorem1}: \textsc{theorem} \\ \quad \forall\ (o:\ \text{Obj},\ o_1:\ \text{Obj},\ o_2:\ \text{Obj}): \\ \quad\ \neg(o = \text{create_object}(o_1) \wedge o = \text{kill_object}(o_2))\end{array}$$

The proof starts with the skolemization of the above formula. Next, each of the Axioms *OAx1* and *OAx2*, defined in the *Object* theory, is applied and later instantiated with the skolem constants (i.e. "oh!1", "o1!1", and "o2!1"). Finally, the BASH[5] command performs some propositional simplifications to end the proof. Note that the basis for this simple proof is specified within the *Object* theory. The PVS representation of this sequence of commands is:

```
(""
 (SKOLEM!)
 (LEMMA "OAx1") (INST -1 "oh!1" "o1!1")
 (LEMMA "OAx2") (INST -1 "oh!1" "o2!1")
 (BASH))
```

Another framework property was previously specified by Theorem 2. Similarly to *Theorem1*, this property constrains the creation and destruction of relationship instances. More specifically, *Theorem2* specifies that the association methods *create_assoc* and *kill_assoc* cannot return the same association instance identifier. In PVS, this property is stated by:

$$\begin{array}{l}\text{Theorem2}: \textsc{theorem} \\ \quad \forall\ (a:\ \text{Acct}, c:\ \text{Card}, s:\ \text{Association}, s_1:\ \text{Association}): \\ \quad\ \neg(s = \text{create_assoc}(c, a) \wedge s = \text{kill_assoc}(s_1))\end{array}$$

[5] This composite command executes a number of simpler commands, as shown in Appendix A.

The axioms used in the proof of *Theorem2* are defined in the *Association* theory. The proof of the property is very similar to the *Theorem1* proof that was just described. However, instead of using Axioms *OAx1* and *OAx2*, we now use their equivalents in the association theory, which are Axioms *SAx4* and *SAx5*.

```
(""
 (SKOLEM!)
 (LEMMA "SAx4") (INST -1 "a!1" "c!1" "s!1")
 (LEMMA "SAx5") (INST -1 "s1!1" "s!1")
 (BDDSIMP)
 (PROPAX))
```

Note that while *Theorem1* proof was completed with the BASH command, the current proof was finalized with the BDDSIMP and PROPAX commands. BDDSIMP is a relatively simple prover command that performs propositional simplifications to the sequent. Alternatively, BASH is a more complex command which tries to apply a number of different commands, BDDSIMP included. The proof strategies here described are not unique, and may not always be the shortest. In fact, any of the two proof strategies are suitable for both of the theorem proofs. Our change of strategy was just for illustrative purposes.

4.6.2 *Views*-Related Properties

We call *views* validation properties the theorems and conjectures derived from the axioms characterizing the relationship. Theory *Views*, described in Schema 10, shows some of the PVS axioms characterizing the relationship, as well as two other theorems which are referred to as properties of the relationship. The proof of these properties is shown in the next subsection. The other subsection is used to illustrate derived properties.

Proof of Properties

In Section 4.4.3, it was mentioned that the *views* Properties 7 and 8 could be inferred from the other axioms of the relationship. In this section we confirm that statement by actually showing the theorem proofs.

As already specified in the Theory *Views*, Property 7 is stated by:

$$\begin{array}{l} \text{Property7}: \ \textsc{theorem} \\ \quad \forall\,(r:\ R, d:\ D, v:\ V): \\ \quad\quad d = \text{viewed}(v, r) \Rightarrow (\text{status}(r) = \text{dead} \Leftrightarrow \text{status}(v) = \text{dead}) \end{array}$$

The proof of the *Property7* theorem is more complex than the proofs presented so far. Three axioms of the *Views* theory – "VAx2", "Property4", and "Property5" – are used as proof lemmas. In addition, this proof is split in two subgoals after the skolemization and flattening of the initial formula. More specifically, the proof of the IFF clause – also represented by "⇔" – is subdivided in the proof of "⇒" and "⇐" clauses by the SPLIT command.

```
(""
 (SKOLEM!)
 (FLATTEN)
 (SPLIT 1)
 (("1"
   (LEMMA "VAx2") (INST -1 "d!1" "r!1" "v!1")
   (BDDSIMP) (PROPAX))
  ("2"
   (FLATTEN)
   (LEMMA "Property5") (INST -1 "r!1")
   (GRIND)
   (LEMMA "Property4") (INST -1 "d!1" "d!2" "r!1" "v!1" "v!2")
   (ASSERT))))
```

The verbose description corresponding to the above Lisp-like representation of the *Property7* theorem proof is shown next.

Schema 16

`Property7`:

$$\{1\} \quad (\forall\ (d:\ D, r:\ R, v:\ V):\ d = \text{viewed}(v, r) \Rightarrow (\text{status}(r) = \text{dead} \Leftrightarrow \text{status}(v) = \text{dead}))$$

Skolemization of the above consequent leads to:

$$\{1\} \quad d' = \text{viewed}(v', r') \Rightarrow (\text{status}(r') = \text{dead} \Leftrightarrow \text{status}(v') = \text{dead})$$

After applying disjunctive simplification to flatten the sequent, we have:

$$\begin{array}{ll} \{\text{-1}\} & d' = \text{viewed}(v', r') \\ \hline \{1\} & (\text{status}(r') = \text{dead} \Leftrightarrow \text{status}(v') = \text{dead}) \end{array}$$

Splitting conjunctions (i.e. the "$\Leftrightarrow$" clause) in the previous sequent leads to 2 subgoals. The sequent describing the `Property7.1` subgoal is represented by:

$$\begin{array}{ll} \{\text{-1}\} & d' = \text{viewed}(v', r') \\ \hline \{1\} & \text{status}(r') = \text{dead} \Rightarrow \text{status}(v') = \text{dead} \end{array}$$

Applying Axiom *VAx2*:

$$\begin{array}{ll} \{\text{-1}\} & (\forall\ (d:\ D, r:\ R, v:\ V):\ d = \text{viewed}(v, r) \wedge \text{status}(v) = \text{alive} \Rightarrow \\ & \quad \text{status}(r) = \text{alive} \wedge \text{status}(d) = \text{alive}) \\ \{\text{-2}\} & d' = \text{viewed}(v', r') \\ \hline \{1\} & \text{status}(r') = \text{dead} \Rightarrow \text{status}(v') = \text{dead} \end{array}$$

Instantiating the quantifier in antecedent formula {-1} with the term d', r', v':

$$\begin{array}{ll} \{\text{-1}\} & d' = \text{viewed}(v', r') \wedge \text{status}(v') = \text{alive} \Rightarrow \\ & \quad \text{status}(r') = \text{alive} \wedge \text{status}(d') = \text{alive} \\ \{\text{-2}\} & d' = \text{viewed}(v', r') \\ \hline \{1\} & \text{status}(r') = \text{dead} \Rightarrow \text{status}(v') = \text{dead} \end{array}$$

Schema 16

Schema 16

Applying the BDDSIMP command, leads to the trivially *true* sequent:

{-1}	FALSE

This completes the proof of the subgoal called `Property7.1`.

The other sequent resulting from the SPLIT command is given by the `Property7.2` subgoal:

{-1}	$d' = \text{viewed}(v', r')$
{1}	$\text{status}(v') = \text{dead} \Rightarrow \text{status}(r') = \text{dead}$

Applying disjunctive simplification to flatten sequent:

{-1}	$\text{status}(v') = \text{dead}$
{-2}	$d' = \text{viewed}(v', r')$
{1}	$\text{status}(r') = \text{dead}$

Applying Axiom *Property5*:

{-1}	$(\forall\ (r:\ R) : \text{status}(r) = \text{alive} \Rightarrow$ $\exists(v:\ V, d:\ D) : (d = \text{viewed}(v, r) \wedge \text{status}(v) = \text{alive}))$
{-2}	$\text{status}(v') = \text{dead}$
{-3}	$d' = \text{viewed}(v', r')$
{1}	$\text{status}(r') = \text{dead}$

Instantiating the quantifier in formula {-1} with the term r':

{-1}	$\text{status}(r') = \text{alive} \Rightarrow$ $\exists(v:\ V, d:\ D) : (d = \text{viewed}(v, r') \wedge \text{status}(v) = \text{alive})$
{-2}	$\text{status}(v') = \text{dead}$
{-3}	$d' = \text{viewed}(v', r')$
{1}	$\text{status}(r') = \text{dead}$

Schema 16

Trying repeated skolemization, instantiation, and if-lifting (i.e. GRIND): **Schema 16**

{-1}	alive?(status(r'))
{-2}	$d'' = \text{viewed}(v'', r')$
{-3}	alive?(status(v''))
{-4}	dead?(status(v'))
{-5}	$d' = \text{viewed}(v', r')$

Applying Axiom *Property4*:

{-1}	$(\forall\ (d:\ D, d_1:\ D, r:\ R, v:\ V, v_1:\ V):$ $d = \text{viewed}(v, r) \wedge d_1 = \text{viewed}(v_1, r) \Rightarrow d = d_1 \wedge v = v_1)$
{-2}	alive?(status(r'))
{-3}	$d'' = \text{viewed}(v'', r')$
{-4}	alive?(status(v''))
{-5}	dead?(status(v'))
{-6}	$d' = \text{viewed}(v', r')$

Instantiating the quantifier in formula {-1} with the terms d', d'', r', v', v'':

{-1}	$d' = \text{viewed}(v', r') \wedge d'' = \text{viewed}(v'', r') \Rightarrow d' = d'' \wedge v' = v''$
{-2}	alive?(status(r'))
{-3}	$d'' = \text{viewed}(v'', r')$
{-4}	alive?(status(v''))
{-5}	dead?(status(v'))
{-6}	$d' = \text{viewed}(v', r')$

Simplifying, rewriting, and recording with decision procedures (i.e. ASSERT) completes the proof of the subgoal `Property7.2`. Q.E.D. **Schema 16**

Property 8 is the other *views* concept to be proved in this section. It involves elements of distinct theories of the system. This property was previously proved in Section 3.7.4 using temporal logic axioms. In that section, the equivalent theorem was referred to as Theorem 3. The PVS specification of the property is represented by:

$$
\begin{array}{l}
\text{Property8}: \ \textsc{theorem} \\
\quad \forall\ (r:\ R, d:\ D, d_1:\ D, v:\ V): \\
\quad\quad d = \text{kill_object}(d_1) \wedge d = \text{viewed}(v, r) \Rightarrow \\
\quad\quad\quad \text{status}(r) = \text{dead} \wedge \text{status}(v) = \text{dead}
\end{array}
$$

The proof of the *Property8* theorem is also split in two subgoals. Each of the subgoals correspond to one of the terms in the conjunction sentence at the right side of the "$\Rightarrow$" symbol in the above formula. Note that the proof of the subgoal corresponding to the "$status(v) = dead$" term of the conjunction is identified by a "1" label in the abbreviated proof description shown next. Alternatively, the proof corresponding to the "$status(r) = dead$" term is labeled as "2".

```
(""
 (SKOLEM! 1)
 (LEMMA "OAx2[D]") (INST -1 "d!1" "d1!1")
 (BASH)
 (("1"
   (LEMMA "VAx2") (INST -1 "d!1" "r!1" "v!1")
   (GROUND))
  ("2"
   (LEMMA "Property7") (INST -1 "d!1" "r!1" "v!1")
   (LEMMA "VAx2") (INST -1 "d!1" "r!1" "v!1")
   (GROUND))))
```

An interesting remark from the above proof description is that the Lemma *OAx2* is used with a parameter D. In fact, such parameter is used to identify which of the two *OAx2* axioms imported by the *Views* theory from the *Object* theory is being

referenced. Those could be either $OAx2[D]$ or $OAx2[R]$. Note from the IMPORTING definition of the *Views* theory in Schema 10, that the elements of the *Object* theory are imported twice. First, the importing statement associates object properties with the viewed class (i.e. IMPORTING Object$[D]$). Second, this same statement associates object properties with the viewer class R. Therefore, one copy of all the attributes, actions and axioms of the *Object* theory exist for each of the D and R classes defined in the *Views* theory.

Verification of Properties

Verification properties are used as a different form to express concepts of the *Views* theory. These alternative specifications represent the confirmation that the original properties hold the desired semantics. In this section, we illustrate these type of properties with a derived concept that was verified within the scope of the colimit theory.

The *DerivedP4* conjecture defines an alternative formula to state that a viewer instance – i.e. a Personal_Card object – cannot be related to more than one viewed object. Such conjecture is defined as:

$$
\begin{array}{l}
\text{DerivedP4}: \textsc{conjecture} \\
\quad \forall\ (c:\ \text{Card}, \text{pc}:\ \text{Personal_Card}, v:\ \text{PC_Views_C}): \\
\quad\ c = \text{viewed}(v, \text{pc}) \Rightarrow \\
\quad\quad \neg\exists\ (c_1:\ \text{Card}, v_1:\ \text{PC_Views_C}): (c_1 = \text{viewed}(v_1, \text{pc}) \wedge c \neq c_1)
\end{array}
$$

As expected, the proof of *DerivedP4* needs only one lemma, which is the *Property4* axiom. Such proof is performed as follows.

```
(""
 (SKOLEM!)
 (LEMMA "Property4")
 (BASH)
 (INST -1 "c1!1" "v1!1")
 (BDDSIMP) (PROPAX))
```

4.6.3 UML-Related Properties

In Section 4.4.1, a few axioms are used to specify the semantics of the different types of association supported by UML. The relevance of each of those axioms to the specification of an association depends on the value of the attributes defining what type of relationship is being modeled. For example, if the *aggregation* attribute of the relationship theory has the value *aggregate*, axioms in the theory – in this particular case *ESAx1* and *ESAx2* [6] – will define specific semantics for this type of relationship.

We now use the formula *UMLConjecture1* to verify the properties of a *composite* type of association. The semantics associated with this formula is that if the "whole" object in a *composite* association is dead, then all the "part" objects related to such "whole" should also be dead. This conjecture is formally stated as:

$$
\begin{array}{l}
\text{UMLConjecture1} : \textsc{conjecture} \\
\quad \forall\, (a : \text{Acct}, c : \text{Card}, s : \text{Association}) : \\
\quad\quad \text{aggregation(CardEnd)} = \text{composite} \wedge \\
\quad\quad\quad a = \text{image}(s, c) \wedge \text{status}(c) = \text{dead} \Rightarrow \text{status}(a) = \text{dead}
\end{array}
$$

The proof of the above formula is based on two axioms related to the *aggregation* attribute of UML relationships (i.e. *ESAx2* and *ESAx3*), and another axiom from the *Association* theory (i.e. *SAx3*). The sequence of prover commands is shown next.

```
("
 (SKOLEM!)
 (LEMMA "ESAx2") (INST -1 "a!1" "s!1")
 (LEMMA "ESAx3") (INST -1 "a!1" "c!1" "_" "s!1" "_")
 (LEMMA "SAx3") (INST -1 "a!1" "c!1" "s!1")
 (GRIND))
```

[6] These axioms are defined in Schema 6.

Another property of the UML theory is defined by formula *UMLConjecture2*. Such validation property describes the semantics for relationships with *changeable* attribute with an *addOnly* value. In addition, the current formula represents an alternative form to specify the constraints previously introduced by Axiom *ESAx5*. The PVS syntax for the conjecture is stated by:

$$
\begin{array}{l}
\text{UMLConjecture2 : CONJECTURE} \\
\quad \forall\ (a:\ \text{Acct}, c:\ \text{Card}, s:\ \text{Association}) : \\
\quad\ \text{changeable(CardEnd)} = \text{addOnly} \wedge \\
\qquad \text{status}(c) = \text{alive} \wedge \text{status}(a) = \text{alive} \Rightarrow \\
\quad\ (a = \text{image}(s, c) \Rightarrow \text{status}(s) = \text{alive})
\end{array}
$$

The proof for the above conjecture is simple. It only uses Axiom *ESAx5* as a lemma. Such axiom was described in theory *ExtendedAssoc2* of Schema 7.

```
(""
 (SKOLEM!)
 (LEMMA "ESAx5") (INST -1 "a!1" "c!1" "s!1")
 (BASH))
```

4.6.4 Domain-Specific Properties

Another group of validation properties is called domain specific. In our case study, the domain is represented by the properties modeling concepts specific to a banking application. These concepts may be presented inside a class theory, a relationship theory, or as part of the colimit of all theories of the system. We illustrate all of these cases.

The Acct Theory

The *Acct* theory represent the typical elements of a banking account, such as balance information and deposit or withdrawal operations. In this case, validation

properties provides a confirmation that both methods and events behave as expected.

An initial conjecture, called *AcctConj1*, defines an expected behavior for the two methods of the class. More specifically, the formula states that the balance of an account should remain unchanged if you perform sequentially debit and credit operations of the same value. This behavior is formally stated as:

$$\begin{array}{l}\text{AcctConj1} : \textsc{conjecture} \\ \quad \forall\ (a:\ \text{Acct}, x:\ \text{Amnt}):\ \text{bal}(\text{debit}(\text{credit}(a,x),x)) = \text{bal}(a)\end{array}$$

The proof of this first *Acct* conjecture is based on the axioms defining the semantics of the two methods of the theory, which are Axioms *AAx5* and *AAx6*.

```
("" (LEMMA "AAx5") (LEMMA "AAx6") (REDUCE))
```

The following conjecture relates the behavior of the *debit* method with the *withdrawal* event. Note that the formula is effective whenever the balance of the account object is greater or equal to zero. Otherwise, a *withdrawal* event does not trigger any method, as stated in Axiom *AAx16* in Schema 3. The conjecture is defined as:

$$\begin{array}{l}\text{AcctConj2} : \textsc{conjecture} \\ \quad \forall\ (a:\ \text{Acct}, x:\ \text{Amnt}) : \text{bal}(a) \geq 0 \Rightarrow \text{bal}(\text{withdrawal}(a,x)) = \text{bal}(\text{debit}(a,x))\end{array}$$

The proof of the *AcctConj2* conjecture uses four *Acct* axioms to be completed. The proof process is described next.

```
(""
 (SKOLEM!)
 (FLATTEN)
 (LEMMA "AAx9") (INST -1 "a!1")
 (LEMMA "AAx14") (INST -1 "a!1" "x!1")
 (LEMMA "AAx15") (INST -1 "a!1" "x!1")
 (LEMMA "AAx1")
 (GRIND))
```

Finally, the *AcctConj3* conjecture associates the behavior of the two events of the *Acct* theory. Note that, similar to the previous conjecture, the *withdrawal* event will be effective only if the account status is not *overdrawn*. This explains the precedent expression in the conjecture formula, which is given by:

$$\begin{array}{l}\text{AcctConj3 : CONJECTURE}\\ \quad \forall\,(a:\ \text{Acct}, x:\ \text{Amnt}):\\ \quad\quad (\text{acct_status}(a) = \text{ok}) \Rightarrow \text{bal}(\text{withdrawal}(\text{deposit}(a,x),x)) = \text{bal}(a)\end{array}$$

Several axioms of the *Acct* theory are used in the proof of the above formula. While no complex proving strategy was used in the process, the selection of lemmas and the values used to instantiate its quantification variables should be carefully considered. The adopted PVS proof strategy follows.

```
("")
 (SKOLEM!)
 (FLATTEN)
 (LEMMA "AAx14") (INST -1 "deposit(a!1, x!1)" "x!1")
 (LEMMA "AAx15") (INST -1 "deposit(a!1, x!1)" "x!1")
 (LEMMA "AAx11") (INST -1 "a!1" "x!1")
 (LEMMA "AAx1") (INST -1 "withdrawal(deposit(a!1, x!1), x!1)"
                         "debit(deposit(a!1, x!1), x!1)")
 (LEMMA "AAx1") (INST -1 "deposit(a!1, x!1)" "credit(a!1, x!1)")
 (LEMMA "AAx5")
 (LEMMA "AAx6")
 (REDUCE))
```

Other Properties

So far, all of the domain-specific properties described were based on the axioms of the *Acct* theory. We now introduce validation properties based on concepts defined in the association and colimit theories of the banking system specification.

First, note that each *Card* object has two attributes, called *chk* and *svg*, that identify the account objects with which such object is associated. According to the *Association* theory interconnecting the *Acct* and *Card* objects, the *chk* attribute has to indicate the value of an account object of type *checking*. Alternatively, the *svg* attribute indicates the existence of a link to an account object of type *savings*. The *AssocConj* conjecture uses these constraints to state that the *chk* and *svg* attributes hold different *Acct* object identities. Formally, this is stated as:

$$\begin{array}{l}\text{AssocConj}: \textsc{conjecture} \\ \quad \forall\ (a:\ \text{Acct}, a_1:\ \text{Acct}, c:\ \text{Card}) : \text{chk}(c) = a \wedge \text{svg}(c) = a_1 \Rightarrow a \neq a_1\end{array}$$

The proof of the above conjecture is based on Axioms *SAx7* and *SAx8*, which use the PVS language to formalize the association properties described in the previous paragraph. The used sequence of prover commands is:

```
("
 (SKOLEM!)
 (LEMMA "SAx7") (INST -1 "a!1" "c!1")
 (LEMMA "SAx8") (INST -1 "a!1" "c!1")
 (REDUCE))
```

Conjecture *ColimitConj* is a property based on a few different specification theories of the banking system (i.e. *Acct*, *Card*, and *Association*). This conjecture defines the expected semantics for a withdrawal operation started from a banking card under certain circumstances. More specifically, the formula states the post-condition for a "Withdraw From Checking" (WFC) type of transaction whenever no other transaction was performed within the same day, the amount requested is less then the daily withdrawal limit, and the balance of the associated account is greater or equal to zero. These conditions are all stated in the following formula:

$$
\begin{array}{l}
\text{ColimitConj}: \textsc{conjecture} \\
\quad \forall\ (a:\ \text{Acct}, c:\ \text{Card}, c_1:\ \text{Card}, d:\ \text{Date}, x:\ \text{Amnt}): \\
\quad\ a = \text{chk}(c)\ \wedge\ c_1 = \text{transaction}(\text{WFC}, d, x, c)\ \wedge\ d \neq \text{last_transaction}(c)\ \wedge \\
\quad\quad x \leq \text{DAILY_MAX}\ \wedge\ \text{notify_balance}(a) \geq 0 \Rightarrow \\
\quad\quad\quad \text{bal}(\text{chk}(c_1)) = \text{bal}(a)\ -\ x
\end{array}
$$

Note that the proof of the *ColimitConj* formula uses axioms defined in three theories. *CAx3* is an axiom of the *Card* theory, *SAx9* and *SAx11* are properties of the *Association* theory, *AAx6* is an axiom of *Acct*, and *AcctConj2* is a conjecture defined within the *Acct* theory that was proved true earlier in this section. These properties were introduced in the proof process which is shown next in an order that was relevant to the adopted strategy.

```
(""
 (SKOLEM!)
 (FLATTEN)
 (LEMMA "CAx3") (INST -1 "c!1" "c1!1" "d!1" "x!1")
 (ASSERT)
 (LEMMA "SAx9") (INST -1 "a!1" "c!1")
 (LEMMA "SAx11") (INST -1 "a!1" "c!1" "x!1")
 (BASH)
 (LEMMA "AcctConj2")
 (LEMMA "AAx6")
 (GRIND))
```

4.7 Other *Views*-Based Systems

The banking system specified in this chapter illustrates the application of some of the most important characteristics of the *views* relationship. While larger and more realistic systems could be used as case studies, we believe the additional complexity would only hinder the focus of the underlying chapter, which is the mechanization of the verification of formal relationship properties.

One of the drawbacks of the adopted example is that it contains only a single specification of a *views* relationship. Therefore, properties such as horizontal consistency, defined in Section 3.7.7, were not formally stated or proved. However, these kind of properties may be easily verified in systems with multiple *views* relationships by means of the consistency axioms defined within each particular relationship theory. For instance, Axioms *DAx1* and *DAx2*, defined in the *PC_Views_C* theory, represent what is called vertical consistency between a viewer and a viewed object. As mentioned in an earlier chapter, this type of consistency is sufficient to infer the horizontal consistency among attribute values of distinct viewer objects.

Chapter 5

Conclusion

5.1 Summary

In this thesis we developed a model called *views*. This model defines a mechanism that disciplines the separation of an object-oriented design into a basic concern, representing the application domain, and special concerns, representing other software issues such as user interfaces. In this model objects can be designed so that they are independent of their environment, because adaptation to the environment is the responsibility of the interface or view.

The basic construct of the model is the *views* relationship, which defines the pattern of interaction among objects representing distinct concerns. This *views* relationship provides a framework for interface modeling which is not supported by other modeling languages. It is characterized by a number of properties which aims to support a disciplined separation of concerns. These properties were described by means of a formal framework that consisted on object calculus theories based on logic and a categorical framework to interconnect those theories. This formal

framework provides the mathematical foundation to support the verification of the properties of both the components and the composite software system.

Finally, we worked on a logic-based specification and reasoning about the object theories of a small banking system. As argued by Rushby, the mechanization of the reasoning process creates opportunities for using formal methods as an exploratory tool [Rus95b]. In our particular case, the main focus of exploration was related to the formal constraints defined for the relationship theories. The mechanization process was based on the PVS formal verification environment, which supported the specification and verification of properties characterizing the object-oriented constructs.

5.2 Future Work

There are different ways to extend the research described in this thesis. These extensions fall in two general categories. First, the modeling approach could be extended and compared to existing implementation mechanisms that support the development of interfaces. Second, larger applications could be developed to assess the importance of each of the specified properties in different situations.

The only stage of development addressed in this thesis was specification. However, it would be valuable to know how the properties identified in the approach would translate into other phases of the software lifecycle. For instance, there are some design patterns [GHJV95] which are currently very popular to the definition of interfacing mechanisms. However, no complete analysis was performed to verify which of their properties are fulfilled by the modeling constructs here presented. Note that such an analysis would first require a formal representation of the patterns.

Other valuable contributions would be based on the assessment of the importance of the properties in solutions. Similarly, to the association attributes in the UML, *views* could have a set of core properties that identifies the semantics of the approach and another set of attributes to identify properties which may be applicable in certain situations. This kind of framework preserves the expressiveness of the relationship construct, while sustaining the flexibility to adapt certain characteristics of the approach according to particular needs.

Another result attainable from the specification of more complex systems is the identification of new relationships and properties. The specification framework described in this thesis together with the mechanization based on the PVS logic is expected to provide the building blocks to the definition of new object-oriented constructs.

Appendix A

The PVS Environment

The PVS environment[1] consists of several tools that are integrated to support the specification of systems or subsystems. A PVS specification, which is based on a higher order logic formalism, can be parsed and typechecked by environment tools. Theorems raised by the typechecker or by the user can be interactively proved with the assistance of the proof checker. Such a tool partially automates the proving process with a number of built-in strategies.

A.1 The PVS Language

The specification language supported by PVS is based on a classical higher-order logic. This logic allows quantification over functions, sets and properties. The PVS language is also based on a rich type system.

In this section, we summarize the PVS language used in this thesis. In the following definitions, we use P, Q and R to denote predicates, S to denote sets,

[1]PVS is a *Prototype Verification System* developed by SRI International.

and X and Y to denote arbitrary types. A complete specification of the language can be found in [SORSC98a].

Declarations

The syntax of a theory declaration is given by the following rule:

name [*theory_formals*] : THEORY
[*exportings*]
BEGIN
[*assuming_part*]
[*theory_part*]
END *name*

where *theory_formals* represents a list of formal parameters, *exportings* defines the list of elements made available by the theory, and *assuming_part* allows the definition of constraints on the use of the theory by means of assumptions. The *theory_part* section typically contains the main body of the theory.

A few other declarations usually defined within the *theory_part* section of a theory are listed next.

IMPORTING *theorynames*	Importation of elements from other theories
$X,\ Y$: TYPE	Uninterpreted types
$X,\ Y$: NONEMPTY_TYPE	Uninterpreted nonempty types
$X,\ Y$: TYPE = *type_expression*	Type declarations
$x,\ y$: VAR *type_expression*	Variable declarations
$x,\ y$: *type_expression* [= *expression*]	Constant declarations
names : AXIOM *expression*	Specification of theory axioms
names : CONJECTURE *expression*	Specification of conjectures

Logic

$\neg P$	Negation
$P \wedge Q$	Conjunction
$P \vee Q$	Disjunction
$P \Rightarrow Q$	Implication
$P \Leftrightarrow Q$	Equivalence (if and only if)
$if\ P\ then\ Q\ else\ R$	Conditional statement
$\exists(x : S) : P$	There exists at least one element of S that satisfies P
$\forall(x : S) : P$	All elements of S satisfy P

Set Theory

$\{x_1, x_2, ..., x_n\}$	Set ennumeration
$x \in S$	Membership
$x \notin S$	Non-membership
$\emptyset$	Empty set

Functions

$X \rightarrow Y$	The set of all total functions from X to Y
$f(a)$	Function application

A.2 PVS Prover Commands

The PVS documentation organizes the large number of available commands in categories. PVS supports a collection of proof commands to carry out *propositional*, *equality*, and *arithmetic* reasoning. There are also *structural* rules which allow, for example, to copy, delete, or hide selected formulae, quantifier rules which allow

to generalize, instantiate, or skolemize formulae, and *control* rules which allow to postpone, quit, undo, or skip partial or complete proof attempts. There is also support to the use of *definitions and lemmas*, *induction*, *simplification procedures*, and other types. The environment also supports the combination of a number of proof commands into a strategy.

This section describes some of the commands available in the PVS proof checker, which is one of the tools available in the PVS verification environment. It is not our goal to make a complete or detailed description of the commands, which can be found in the PVS prover guide [SORSC98b]. Rather, we give a brief description of the applicability of some of the commands used during the research associated with this thesis. To this small set we add a few other commands which we consider relevant.

A.2.1 Verification Commands

ASSERT - This command represents a combination of rules to perform simplification using decision procedures. These procedures are invoked to prove trivial theorems, to simplify complex expressions, and to perform matching.

BASH - This command consists in the ordered execution of a number of simplification and instantiation commands.

BDDSIMP - Performs propositional simplifications by means of an external package based on binary decision diagrams (BDDs).

CASE (use: case "*expression1*" "*expression2*") - The CASE command allows the splitting of the current sequent in a number of subgoals. These subgoals are derived from the parameters specified in the proof command. If n parameters are given, $n + 1$ subgoals are generated.

FLATTEN - Disjunctively simplifies sequent formulae containing disjuncts. A disjunct is an antecedent formula of the form $\neg A$ or $A \wedge B$, or a consequent formula of the form $\neg A$, $A \rightarrow B$ or $A \vee B$.

GRIND - This strategy is commonly used to automatically complete a proof branch or to apply the obvious simplifications. PVS calls it a "catch-all" strategy.

GROUND - This command invokes propositional simplifications followed by an ASSERT command.

LEMMA (use: lemma "*lemma_name*") - This rule introduces in the sequent an instance of the lemma called *lemma_name*. All of the lemmas used in our proofs were defined as axioms of a theory.

INDUCT (use: induct "*variable*") - This command automatically employs an induction scheme. The variable name *variable* must be quantified at the outermost level of a universally quantified consequent formula.

INST (use: inst *formula_number* "*term1*" "*term2*") - The universally quantified formulae in the antecedent and the existentially quantified formulae in the consequent are reduced by instantiating the quantified variables.

PROPAX - This command proves trivial sequents such as "TRUE $\rightarrow \Delta$". It is automatically applied by the prover to conclude the proving process.

REDUCE - This command is the main workhorse of the GRIND command. It repeatedly uses the BASH command to perform simplification with decision procedures.

SKOLEM! - The bound variables of the sequent are replaced with Skolem constants.

SPLIT - The conjuntive formulae in the current goal sequent are split.

A.2.2 Control and Structural Commands

DELETE (use: delete *formula_number*) - This command yields the subgoal where the formulae identified by *formula_number* in the current goal sequent have been deleted.

POSTPONE - This is used to mark the current goal as pending to be proved and to shift the focus to the next pending proof.

QUIT - It terminates the current proof attempt.

UNDO - This command is used to undo the proof until a certain previous step.

Bibliography

[ACLN95] P.S.C. Alencar, D. Cowan, C.J.P. Lucena, and L.C.M. Nova. Formal Specification of Reusable Interface Objects. *ACM SIGSOFT Software Engineering Notes*, 20(SI):88–96, August 1995.

[ACLN98a] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and L.C.M. Nova. Gluing components together. In *Proceedings of ECOOP'98 Workshop Reader*, Brussels, Belgium, July 1998.

[ACLN98b] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena, and L.C.M. Nova. A model for gluing components. In *Proceedings of the Third International Workshop on Component-Oriented Programming*, TUCS General Publication No 10, pages 101–108, Brussels, Belgium, October 1998.

[ACN98] P.S.C. Alencar, D.D. Cowan, and L.C.M. Nova. A formal theory for the views-a relationship. In *Proceedings of the Third Northern Formal Methods Workshop*, Ilkley, UK, September 1998.

[AG94] Antonio Alencar and Joseph Goguen. Specification in OOZE with Examples. In K. Lano and H. Houghton, editors, *Object-Oriented Specification Case Studies*, pages 158–183. Prentice-Hall, 1994.

[Age96] S. Agerholm. Translating Specifications in VDM-SL to PVS. In *Proceedings of the Ninth International Conference on Theorem Proving in Higher Order Logics (TPHOL'96)*, 1996.

[Aks96] Mehmet Aksit. Separation and Composition of Concerns. In *ACM Workshop on Strategic Direction in Computing Research*, USA, June 1996.

[AM94] J. Armstrong and R. Mitchell. Uses and Abuses of Inheritance. *Software Engineering Journal*, pages 19–26, January 1994.

[Bar87] H. Barringer. The Use of Temporal Logic in the Compositional Specification of Concurrent Systems. In A. Galton, editor, *Temporal Logic and Their Applications*. Academic Press, 1987.

[BC91] Len Bass and Joëlle Coutaz. *Developing Software for the User Interface*. The SEI Series in Software Engineering. Addison-Wesley, 1991.

[BC95] R.H. Bourdeau and B.H.C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10), October 1995.

[BG77] R. Burstall and J. Goguen. Putting Theories Together to Make Specifications. In R. Reddy, editor, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, 1977.

[BH94] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths on Formal Methods. Technical Report PRG-TR-7-94, Oxford University Computing Laboratory, 1994.

[BLM97a] J.C. Bicarregui, K.C. Lano, and T.S.E. Maibaum. Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation. In *Proceedings of ECOOP*, Finland, 1997.

[BLM97b] J.C. Bicarregui, K.C. Lano, and T.S.E. Maibaum. Towards a Compositional Interpretation of Object Diagrams. In *Proceedings of IFIP TC2 Working Conference on Algorthmic Languages and Calculi*, Strasbourg, February 1997.

[Boe87] Barry W. Boehm. Improving Software Productivity. *Computer*, 20(9):43–57, September 1987.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.

[BR87] Ted J. Biggerstaff and Charles Richter. Reusability Framework, Assessment, and Directions. *IEEE Software*, 4(2), March 1987.

[BR94] P. Brassi and R. Rousseau. Irec: An Object Oriented Abstract Representation to Handle Software Components. In *Proceedings of ACFAS*, Montreal, Canada, May 1994.

[BW90] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[C+90] D. Carrington et al. Object-Z: An Object-Oriented Extension to Z. In *Proceedings of Formal Description Techniques, II (FORTE'89)*, pages 281–296, North-Holland, December 1990.

[CBI+92] D.D. Cowan, L.F. Barbosa, R. Ierusalimschy, C.J.P. Lucena, and S.B. Oliveira. Program Design Using Abstract Data Views–An Illustrative Example. Technical Report 92–54, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada, December 1992.

[CD94] S. Cook and J. Daniels. *Designing Object-Oriented Systems: Object-Oriented Modeling with Syntropy.* Prentice-Hall, 1994.

[CES97] Krzysztof Czarnecki, Ulrich Eisenecker, and Patrick Steyaert. Beyond Objects: Generative Programming. In *Proceedings of Aspect-Oriented Programming Workshop at ECOOP*, Finland, June 1997. Springer-Verlag.

[CILS93a] D.D. Cowan, R. Ierusalimschy, C.J.P. Lucena, and T.M. Stepien. Abstract Data Views. *Structured Programming*, 14(1):1–13, January 1993.

[CILS93b] D.D. Cowan, R. Ierusalimschy, C.J.P. Lucena, and T.M. Stepien. Application Integration: Constructing Composite Applications from Interactive Components. *Software Practice and Experience*, 23(3):255–276, March 1993.

[Civ93] Franco Civello. Roles for Composite Objects in Object-Oriented Analysisand Design. In *Proceedings of OOPSLA*, pages 376–393, 1993.

[CL95] D.D. Cowan and C.J.P. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design. *IEEE Transactions on Software Engineering*, 21(3):229–243, March 1995.

[CLV93] D.D. Cowan, C.J.P. Lucena, and R.G. Veitch. Towards CAAI: Computer Assisted Application Integration. Technical Report 93–17,

Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, January 1993.

[Cor97] Rational Software Corporation. *UML Semantics - version 1.0*. January 1997. Available at http://www.rational.com/uml/.

[CY91] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press/Prentice-Hall, 1991.

[DBH95] Scott DeLoach, Paul Bailor, and Thomas Hartrum. Representing Object Models as Theories. In *Proceedings of the Tenth Knowledge-Based Software Engineering Conference*, pages 28–35, Boston, MA, November 1995.

[DH99] Scott DeLoach and Thomas Hartrum. A Theory-Based Representation for Object-Oriented Domain Models. *accepted for publication in IEEE Transactions on Software Engineering*, 1999.

[Dig91] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc., and SunSoft, Inc. *The Common Object Reqest Broker: Architecture and Specification*, OMG document number 91.12.1, revision 1.1 edition, December 1991.

[Dig92] Digitalk. *PARTS Workbench User's Guide*. Digitalk, 1992.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, October 1972.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dil90] Antoni Diller. *Z An Introduction to Formal Methods.* John Wiley and Sons, 1990.

[DK76] F. DeRemer and H. Kron. Programming-in-the-large Versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

[DN70] M.B. Dahl and K. Nygaard. SIMULA Common Base Language. Technical Report S-22, Norvegian Computing Center, 1970.

[EFLR98] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the UML as a formal modelling notation. In Pierre-Alain Muller and Jean Bézivin, editors, *Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998*, pages 297–307. ESSAIM, Mulhouse, France, 1998.

[FM91] J. Fiadeiro and T. Maibaum. *Describing, Structuring, and Implementing Objects*, volume 489 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[FM92] J. Fiadeiro and T. Maibaum. Temporal Theories as Modularisation Units for Concurrent System Specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[FSJ99] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks.* John Wiley and Sons, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, Massachusetts, 1995.

[Gog86] Joseph A. Goguen. Reusing and Interconnecting Software Components. *IEEE Computer*, 19(2), February 1986.

[Gog89] J. Goguen. A Categorical Manifesto. Technical Report PRG-72, Programming Research Groupo, University of Oxford, March 1989.

[Hal90] J.A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[Hil92] Ralph D. Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In *CHI '92*, pages 335–342. ACM, May 1992.

[HL95] W. Hursch and C. Lopes. Separation of Concerns. Technical report, Northeasten University, February 1995.

[HSL91] R.C. Holt, T. Stanhope, and G. Lausman. Object-Oriented Computing: Looking Ahead to the Year 2000. Technical Report ITRC TR-9101, Information Technology Research Center, Univ. of Toronto, April 1991.

[IBM94] IBM. *Visual Age: Concepts & Features*. IBM, 1994.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: a Use Case Driven Approach*. ACM Press, 1992.

[KDGN97] John Knight, Colleen DeJong, Matthew Gibble, and Luís Nakano. Why Are Formal Methods Not Used More Widely? In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM' 97:*

Fourth NASA Langley Formal Methods Workshop, NASA Conference Publication 3356, pages 1–12, Hampton, VA, September 1997. NASA Langley Research Center. Available at http://atb-www.larc.nasa.gov/Lfm97/proceedings/.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997.

[KP88] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 3(1):26–49, August September 1988.

[Lam91] Leslie Lamport. The Temporal Logic of Actions. Technical Report 79, Digital Equipment Corporation, Systems Research Center, December 1991.

[Lan95] Kevin Lano. *Formal Object-Oriented Development*. Springer-Verlag, 1995.

[LB98a] Kevin Lano and Juan Bicarregui. Formalising the UML in structured temporal theories. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings of the Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 105–121. Technische Universität München, TUM-I9813, 1998.

[LB98b] Kevin Lano and Juan Bicarregui. Semantics and transformations for UML models. In Pierre-Alain Muller and Jean Bézivin, editors,

Proceedings of UML'98 International Workshop, Mulhouse, France, June 3 - 4, 1998, pages 97–106. ESSAIM, Mulhouse, France, 1998.

[LCP92] C.J.P. Lucena, D.D. Cowan, and A.B. Potengy. A Programming Model for User Interface Compositions. In *Anais do V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, SIBGRAPI'92*, Aguas de Lindóia, SP, Brazil, November 1992.

[LH94] K. Lano and H. Houghton. Specifiying a Concept-Recognition System in Z++. In K. Lano and H. Houghton, editors, *Object-Oriented Specification Case Studies*, pages 137–157. Prentice-Hall, 1994.

[Lim94] Wayne C. Lim. Effects of Reuse on Quality, Productividy, and Economics. *IEEE Software*, September 1994.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5):17–34, May 1988.

[MB97] Savi Maharaj and Juan Bicarregui. On Verification of VDM Specification and Refinement with PVS. In *Proceedings of the Twelfth IEEE International Conference in Automated Software Engineering (ASE'97)*, 1997.

[MC92] Silvio Meira and Ana Cavalcanti. The MooZ Specification Language. Technical report, Departamento de Informatica, Universidade Federal de Pernambuco, Recife, PE, Brazil, 1992.

[McC97] Carma McClure. *Software Reuse Techniques*. Prentice Hall, 1997.

[Mey85] Bertrand Meyer. On Formalisms in Specifications. *IEEE Software*, pages 6–26, January 1985.

[Mey99] Bertrand Meyer. Every Little Bit Counts: Toward More Reliable Software. *IEEE Computer*, pages 131–135, November 1999.

[Mit65] Barry Mitchell. *Theory of Categories*. Academic Press, 1965.

[MMM95] H. Mili, F. Mili, and A. Mili. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[Mye91] Brad A. Myers. Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In *UIST–Fourth Annual Symposium on User Interface Software Technology*, pages 211–220, 1991.

[Par72a] D. Parnas. A Technique for Software Module Specification with Examples. *CACM*, 15(5), 1972.

[Par72b] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *CACM*, 15(12), December 1972.

[Par90] H.A. Partsch. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

[Par97] Rational Partners. *UML Notation Guide*. Object Management Group (OMG), September 1997. Available at http://www.rational.com/uml/.

[Pen93] John J. Penix. *Automated Component Retrieval and Adaptation Using Formal Specifications*. PhD thesis, Department of Electrical and

Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio, 1993.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, 1991.

[PLC93] A.B. Potengy, C.J.P. Lucena, and D.D. Cowan. A Programming Approach for Parallel Rendering Applications. Technical Report 93–62, Computer Science Department and Computer Systems Group, University of Waterloo, Waterloo, Ontario, Canada, March 1993.

[Pre92] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 1992.

[Rat97] Rational Partners (Rational, HP, IBM, MCI, Microsoft, ObjecTime, Oracle, Unisys, etc.). *UML Semantics.* Object Management Group (OMG), September 1997. Available at http://www.rational.com/uml/.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Rei97] D.J. Reifer. *Practical Software Reuse.* John Wiley and Sons, 1997.

[Rum88] James Rumbaugh. Relational Database Design Using an Object-Oriented Methodology. *Communications of the ACM*, 31(4):417, April 1988.

[Rus95a] John Rushby. Formal Methods and Their Role in Certification of Critical Systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1995.

[Rus95b] John Rushby. Mechanizing Formal Methods: Opportunities and Challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.

[Sah81] Sartaj Sahni. *Concepts in Discrete Mathematics.* The Camelot Publishing Company, 1981.

[SD98] Monique Snoeck and Guido Dedene. Existence Dependency: The Key to Semantic Integrty Between Structural and Behavioral Aspects of Object Types. *IEEE Transaction on Software Engineering*, pages 233–251, April 1998.

[SLMD96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proceedings of OOPSLA*, San Jose, California, 1996.

[SORSC98a] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[SORSC98b] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[SORSC98c] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[Sri90] Y.V. Srinivas. Category Theory Definitions and Examples. Technical Report TR-90-14, Department of Information and Computer Science, University of California, Irvine, CA, February 1990.

[Syb96] Sybase. *Optima++*. Sybase, 1996.

[Was94] Michael Wasmund. Reuse Facts and Myths. In *Proceedings of IEEE International Conference on Software Engineering*, pages 273–274, 1994.

[Wat93] Watcom International Corporation, Waterloo, Ontario, Canada. *WATCOM VX·REXX for OS/2 Programmer's Guide and Reference*, 1993.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[WdJS95] Roel Wieringa, Wiebren de Jonge, and Paul Spruit. Using Dynamic Classes and Role Classes to Model Object Migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995.

[Wen94] Kevin D. Wentzel. Software Reuse - Facts and Myths. In *Proceedings of IEEE International Conference on Software Engineering*, pages 267–268, 1994.

[WRC97] Enoch Y. Wang, Heather A. Richter, and Betty C. Cheng. Formalizing and Integrating the Dynamic Model within OMT. In *Proceedings*

of IEEE International Conference on Software Engineering, pages 45–55, Boston, Massachusetts, May 1997.

[WWC92] G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming. *CACM*, 35(11), November 1992.