

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

Heterogeneous View Integration and its Automation

by

Alexander Franz Egyed

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Science)

August 2000

Copyright 2000

Alexander Franz Egyed

UMI Number: 3018075

Copyright 2000 by
Egyed, Alexander Franz

All rights reserved.

UMI[®]

UMI Microform 3018075

Copyright 2001 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

To my parents Franz and Berta

Acknowledgements

So many people contributed to the making of this work that it seems impossible to thank all of them. I am very grateful towards my advisor Barry Boehm for all the dedication and support he has given along these many joyful years. I am also most grateful for all the invaluable suggestions I got from Nenad Medvidovic.

I would like to thank the reviewers of my thesis including my other dissertation committee members Lewis Johnson, Bert Steece, and David Wile as well as other reviewers like Nicolas Rouquette. Also my thanks to all the people who gave suggestions and insights along the way, including the anonymous reviewers of journal and conference papers I have written. Moreover, I would like to thank Philippe Kruchten.

My dissertation would have been impossible if other researchers would not have provided a foundation onto which to build upon. I greatly value the papers I have read for they opened my eyes onto the vast challenges that exist but also onto the unique solutions they provided. My work builds upon their contributions.

Finally yet importantly I would like to thank my family, friends, and colleagues for their support or just for being there. I value their friendship, encouragement, and companionship.

Table of Contents

DEDICATION	ii
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	viii
LIST OF FIGURES	ix
ABSTRACT.....	xiv
1 INTRODUCTION	1
1.1 OVERVIEW	1
1.2 MOTIVATION	3
1.3 CONTRIBUTIONS	5
1.4 BACKGROUND INFORMATION	7
1.5 OUTLINE.....	9
1.6 SUMMARY	11
2 MODEL-BASED SOFTWARE DEVELOPMENT.....	12
2.1 SOFTWARE MODELING	12
2.2 MODELS AND VIEWS	13
2.3 COMMON MODELS AND VIEWS	14
2.4 THE UNIFIED MODELING LANGUAGE (UML).....	16
2.5 ARCHITECTURE DESCRIPTION LANGUAGES (ADL)	18
2.6 STAKEHOLDERS AND MODEL LIFE CYCLES	20
2.7 INFORMATION GAP AND INFORMATION DISCONTINUITY	23
2.8 INFORMATION DEGRADATION	25
2.9 SUMMARY	27
3 MODEL INTEGRATION.....	28
3.1 WHAT ARE MODEL REDUNDANCIES?	28
3.2 MISSING INTEGRATION IN MODELS AND VIEWS	31
3.3 WHAT IS INTEGRATION?	32
3.4 THE VIEW INTEGRATION PROBLEM	34
3.4.1 Why Integrate Views?.....	34
3.4.2 Why Integrate Heterogeneous Views?	35
3.4.3 Why Automate Heterogeneous View Integration?	36
3.5 BRIDGING THE INFORMATION GAP: SYNTHESIS AND ANALYSIS	36
3.6 POTENTIAL INTEGRATION COMPLEXITY	39
3.7 WHAT IS NOT INTEGRATION?.....	39
3.8 SUMMARY	41
4 SCOPE AND LIMITATIONS	42

5	MODEL ELEMENTS AND VIEWS	45
5.1	WHAT ARE MODEL ELEMENTS, VIEWS, AND MODELS?	45
5.2	MODEL ELEMENTS, MODEL INSTANCES, AND USER OBJECTS.....	47
5.3	UML MODEL, MODEL ELEMENTS, AND VIEWS	48
5.4	VIEW DIMENSIONS	49
5.4.1	Level of Generality	49
5.4.2	Level of Abstraction.....	50
5.4.3	Level of Behaviorism.....	51
5.5	VIEW SPACE AND ITS RELATION TO VIEWS	51
5.6	INTERDEPENDENCIES OF MODEL ELEMENTS	54
5.7	SUMMARY	55
6	MODEL INCONSISTENCIES.....	56
6.1	EXAMPLES OF INCONSISTENCIES	56
6.1.1	Inconsistency between Class Layers	56
6.1.2	Inconsistency between Class and Sequence Diagram	57
6.1.3	Cardinality Inconsistency	58
6.1.4	Inconsistency between State and Sequence Diagrams	59
6.2	LIST OF INCONSISTENCIES	61
6.2.1	Inconsistencies in the Abstract Dimension	61
6.2.2	Inconsistencies in the Generic Dimension	70
6.2.3	Inconsistencies in the Behavioral Dimension	75
6.3	CLASSIFICATION OF INCONSISTENCIES	78
6.4	ACT IN THE PRESENCE OF INCONSISTENCIES	78
6.5	SUMMARY	79
7	OUR VIEW INTEGRATION FRAMEWORK	80
7.1	OVERVIEW.....	80
7.2	VIEW INTEGRATION FRAMEWORK.....	81
7.3	SIMPLE MODEL TRANSFORMATION	85
7.3.1	Abstraction.....	89
7.3.1.1	Classifier Abstraction.....	89
7.3.1.2	Relation Abstraction	91
7.3.1.3	Semantic Rules for Abstraction	91
7.3.1.4	Complex Abstraction	97
7.3.1.5	Abstraction Algorithm	99
7.3.1.6	Specialized Abstraction	100
7.3.1.7	Example	101
7.3.2	Generalization	103
7.3.2.1	Sequence to Statechart Generalization.....	104
7.3.2.2	Object to Class Generalization.....	108
7.3.2.3	Generalization Rules and Automation	110
7.3.3	Structuralization	110
7.3.3.1	Sequence to Object Structuralization.....	111
7.3.3.2	Statecharts to Class Structuralization.....	113
7.3.3.3	Structuralization Rules and Automation	114
7.3.4	Translation	115
7.4	COMPLEX TRANSFORMATION	116
7.4.1	Deferred Issues.....	122
7.5	AUTOMATING MODEL DIFFERENTIATION	123

7.5.1	Comparing User-Defined and Derived Elements.....	124
7.5.1.1	Comparison Modes.....	124
7.5.1.2	Multiple Interpretations and Realizations.....	125
7.5.1.3	Ambiguous Interpretations.....	128
7.5.2	Consistency Rules.....	130
7.5.2.1	List of Inconsistencies.....	130
7.5.2.2	Simple Consistency Checking Example.....	133
7.5.2.3	Consistency Rules Defined and Applied.....	136
7.5.3	Triggering Transformation.....	144
7.5.4	User Interaction.....	146
7.5.5	Deferred Issues.....	148
7.6	MODEL SYNTHESIS AND MAPPING.....	148
7.6.1	Model Synthesis.....	148
7.6.2	Model Mapping.....	149
7.6.2.1	Traceability Types.....	149
7.6.2.2	Mapping Support.....	150
7.6.3	Deferred Issues.....	150
7.7	MODEL REPOSITORY.....	150
7.7.1	Implementing View Integration Elements.....	151
7.7.2	Evolutionary Scalability Problem.....	152
7.7.3	Reduced Redundancy Model.....	153
7.7.3.1	Reduced Redundancy Model for Class Diagrams.....	156
7.7.3.2	Reduced Model Redundancy and UML.....	163
7.7.4	Purging.....	165
7.8	SUMMARY.....	166
8	CASE STUDY.....	168
8.1	ARCHITECTURE LEVEL.....	168
8.2	REFINEMENT TO HIGHER-LEVEL DESIGN.....	169
8.2.1	Overview.....	169
8.2.2	Transformations.....	176
8.2.3	Consistency Checking.....	177
8.3	REFINEMENT TO LOWER-LEVEL DESIGN.....	179
8.3.1	Overview.....	179
8.3.2	Transformations.....	184
8.3.3	Consistency Checking.....	185
8.4	SCALABILITY.....	191
8.5	SUMMARY.....	192
9	UML/ANALYZER-A TOOL.....	193
10	RELATED WORK.....	198
10.1	OVERVIEW.....	198
10.2	COMPARISON OF VIEW INTEGRATION APPROACHES.....	200
10.3	INTEGRATION CRITERIA.....	203
10.4	MODES CRITERIA.....	205
10.5	MEDIA CRITERIA.....	206
10.6	VIEW DIMENSIONS CRITERIA.....	206
10.7	LIFE-CYCLE CRITERIA.....	208
10.8	FLOW CRITERIA.....	210

10.9	SCALABILITY CRITERIA	210
10.10	OTHER MODELS CRITERIA	211
11	EVALUATION, FUTURE WORK, AND SUMMARY	212
11.1	EVALUATION	212
11.1.1	Evaluating Transformation Techniques	212
11.1.2	Evaluating Comparison Methods.....	214
11.1.3	Evaluating Effectiveness, Efficiency, and Reliability.....	215
11.1.4	Evaluating Scalability	217
11.1.5	Evaluating Applicability outside UML Domain	217
11.1.6	Evaluating UML's ability to support analysis	218
11.1.6.1	Reduced Redundancy Model	218
11.1.6.2	Explicit and Implicit Treatment of Traces	219
11.1.6.3	Ambiguous and Partial Interpretations.....	219
11.1.7	Evaluating in the Context of Other Approaches	220
11.1.8	Breadth over Depth.....	222
11.1.9	Technology and Research Transfer	224
11.2	FUTURE WORK	224
11.3	CONCLUSION	226
12	REFERENCES.....	228

List of Tables

Table 1. Stakeholder Concerns as Architecture Evaluation Criteria from [Gacek et al. 1995]	21
Table 2. Discontinuity of Project Information over Time	24
Table 3. UML Views and Diagrams adapted from [Rumbaugh et al. 1999]	48
Table 4. Eight Regions of the View Space	53
Table 5. List of Inconsistencies on the Abstract/Concrete Dimension	130
Table 6. List of Inconsistencies on the Generic/Specific Dimension	131
Table 7. List of Inconsistencies on the Structural/Behavioral Dimension	132
Table 8. Comparison of View Integration Approaches	199

List of Figures

Figure 1: Mathematical Systems Theory.....	12
Figure 2: Software Engineering Theory	13
Figure 3: Some of Diagrammatic Views support by UML	16
Figure 4: Architectural Views in UML	22
Figure 5: Information Degradation over Time	27
Figure 6. Different Views for Hospital System.....	28
Figure 7. View Redundancy.....	29
Figure 8: Two Problem Solving Approaches	31
Figure 9. Integration to Enable Automated Synthesis and Analysis	37
Figure 10. Information Discontinuity, Degradation, and Restoration	38
Figure 11: Complexity in Integrating Views.....	39
Figure 12: Views and ADLs represented in UML	40
Figure 13. UML Core Elements as defined in [OMG 1999].....	45
Figure 14. The Four-Layer Meta-Modeling Architecture of UML [Medvidovic et al. 1999b].....	47
Figure 15. Views Dimensions	50
Figure 16. Views and the View Space	52
Figure 17: Potential Mismatch between two Layers (Completeness)	57
Figure 18: Potential Mismatch between Class Diagram and Sequence Diagram.....	58
Figure 19: Potential Mismatch Between a Structural View and two Behavioral Views	59
Figure 20: Potential Mismatch between State-, Sequence- and Collaboration Diagrams	60
Figure 21. Concrete Relation has no Corresponding Abstraction	61
Figure 22. Concrete Classifier has no Corresponding Abstraction	62
Figure 23. Abstract Classifier has not been Refined	63
Figure 24. Concrete Classifier is of Different Type than its Corresponding Abstraction	64
Figure 25. Concrete Relation uses Abstract Classifier Instead of its Refinement	64

Figure 26. Abstract Classifier is Replicated at the Concrete Level Although Refinement Exists.....	65
Figure 27. Concrete Classifier is Assigned to Multiple Abstract Classifiers	66
Figure 28. Cardinality of Refinement does not Match its Abstraction.....	66
Figure 29. Direction of Concrete Relation does not Match its Abstraction	67
Figure 30. Concrete Classifier does not Replicate a Method of its Abstraction.....	68
Figure 31. Concrete Method is of Different Type than its Corresponding Abstraction	68
Figure 32. Specific Relation has no Corresponding Generalization.....	70
Figure 33. Cardinality of Generic Classifiers does not Match Specific Scenarios.....	71
Figure 34. Direction of Specific Relation does not Match its Generalization.....	72
Figure 35. Specific View uses a Method that is not Defined in Generic Classifier	73
Figure 36. Specific Classifier has not been Assigned to Generic Classifier.....	73
Figure 37. Generic Pre-Condition is Violated in Specific View	74
Figure 38. Structural View does not Support all Behavioral Needs.....	75
Figure 39. Structural Declaration does not Match its Usage.....	76
Figure 40: Categories of Mismatches.....	78
Figure 41: Model-based Development—a view independent representation.....	80
Figure 42. View Integration Framework.....	81
Figure 43. View Transformation and Mapping to Complement View Comparison	83
Figure 44. Transforming Model Elements between Regions in the View Space	86
Figure 45. View Dimension and View Transformation Axes	87
Figure 46. Transformations Currently Supported	88
Figure 47: Classifier (left) and Relation (right) Abstraction—Two Approaches.....	90
Figure 48. Class Patterns.....	92
Figure 49: Simple Input/Output Structure Patterns for Abstractions	94
Figure 50: Abstraction Rules for Class/Object Diagrams	96
Figure 51. Serial Abstraction	97
Figure 52: Abstraction Rules for State Diagrams.....	98

Figure 53: Abstraction Rules for Package Diagrams	99
Figure 54. Abstraction Algorithm	99
Figure 55. Cardinality Examples between Classes.....	100
Figure 56. Cardinality Examples and their Abstractions.....	101
Figure 57. Simple Example of Generated Abstractions from three Input Diagrams.....	102
Figure 58: Generating transitive relationship from Flight to Person	103
Figure 59. Sequence to State Generalization–Basics	105
Figure 60. Sequence to State Generalization–Extended.....	107
Figure 61. Minimal Statechart Diagram.....	107
Figure 62. Object Diagram.....	108
Figure 63. Generalized Object Diagram Represented as Class Diagram	109
Figure 64. Generalization Patterns	110
Figure 65. Sequence Diagram	112
Figure 66. Sequence Diagram Structuralized into on Object Diagram	113
Figure 67. Structuralizing Statechart views into Class Views.....	114
Figure 68. Structuralization Patterns	115
Figure 69. Transformation Methods and Paths	117
Figure 70. Complex Transformation Paths	118
Figure 71. Complex Transformation Algorithm	119
Figure 72. Lack of Intermediate Views in Covering Full Transformation.....	120
Figure 73. Examples of Equivalence Comparison	124
Figure 74. Examples of Part-of Comparison.....	124
Figure 75. One-to-many Comparison.....	126
Figure 76. Many-to-One Comparison	126
Figure 77. Zero-to-one Comparison.....	127
Figure 78. Variations in View Comparison.....	127
Figure 79. Ambiguous Comparison	128

Figure 80. Variations (Ambiguities) in Transformation Results	129
Figure 81. Refinement Inconsistency	133
Figure 82. Abstraction Example.....	135
Figure 83. Example of Consistency Checking between Abstract and Concrete Elements.....	136
Figure 84. Example of Consistency Checking between Generic and Specific Elements	141
Figure 85. Origin Traces and Interpretations Traces.....	151
Figure 86. Evolutionary Complexity.....	153
Figure 87. Reducing Model Redundancy using UML	154
Figure 88. Reducing Model Redundancy outside UML	155
Figure 89: Generic Example of Classifier Abstraction	157
Figure 90: UML-A Model satisfying Classifiers for Multiple Abstractions	158
Figure 91: Ambiguity in Accessing Composite Classifiers via Relational Names	159
Figure 92: Deriving correct Projection from Relation Identifier.....	160
Figure 93. Generic Example of Relation Abstraction	160
Figure 94: UML-A Model satisfying Relations for Multiple Abstractions.....	161
Figure 95: Ambiguity in Accessing Composite Relations via Classifier Names	162
Figure 96: Special Case of Relations using Associations.....	163
Figure 97: Linear Integration Work using an Integrated Repository	165
Figure 98. Architecture Overview of HMS	168
Figure 99. HMS Data Types	169
Figure 100. Employees Interacting with Applications using Services	170
Figure 101. Services, their Dialogs, and the Database (DB).....	171
Figure 102. Containers used by Dialogs	172
Figure 103. Data Types used by Services and Dialogs	172
Figure 104. Mapping from Design Classes to Architecture Components	173
Figure 105. Object Diagram Depicting the Relationships between Guests and Hotels.....	174
Figure 106. Statechart for <i>EditDlg</i>	174

Figure 107. Statechart for <i>CaptureContainer</i>	174
Figure 108. Statechart for <i>ReservationService</i>	175
Figure 109. Sequence Diagram depicting a Search for a Reservation	176
Figure 110. Transformations to support Consistency Checking of Architecture and Design	176
Figure 111. Abstracted Design-Level Class Diagram.....	177
Figure 112. Structuralized and Generalized Sequence Diagram.....	178
Figure 113. Generalized Object Diagram.....	178
Figure 114. Structuralized Statechart Diagrams.....	179
Figure 115. Low-Level Design of Basic HMS Data Types	180
Figure 116. ReservationServices, ReservationDlg, and its Containers	181
Figure 117. Statechart diagram for ReservationCaptureContainer.....	181
Figure 118. Startchart Diagram for ReservationEditDlg.....	182
Figure 119. Sequence Diagram Capturing the Modification of a Reservation.....	183
Figure 120. Transformations to support Consistency Checking between Designs	184
Figure 121. Mapping from Low-Level Design Classes to High-Level Design Classes.....	185
Figure 122. Abstracted Low-Level Design Class Diagram.....	186
Figure 123. Abstracted Statechart Diagram for EditDlg.....	187
Figure 124. Abstracted Sequence Diagram for <i>modify_reservation()</i>	188
Figure 125. Structuralization and Generalization of Sequence Diagram	189
Figure 126. Generalized Sequence Diagram to Statechart Diagram	190
Figure 127. Structuralized Statechart Diagrams into Class Diagrams	190
Figure 128. UML-Analyzer Tool Supporting View Integration	193
Figure 129. Complexity in Class Abstraction	194
Figure 130. Inconsistencies between HMS Architecture and High-Level Design.....	195
Figure 131. Inconsistencies between HMS High- and Low-Level Designs.....	196
Figure 132. Reuse and Duplication Elimination during Abstraction	197

Abstract

Software systems are characterized by unprecedented complexity. One effective means of dealing with that complexity is to consider a system from a particular perspective, or view (e.g., architecture or design diagram). Views enable software developers to reduce the amount of information they have to deal with at any given time. They enable this by utilizing a divide-and-conquer strategy that allows large-scale software development problems to be broken up into smaller, more comprehensible pieces. Individual development issues can then be evaluated without the need of access to the whole body of knowledge about a given software system. The major drawback of views is that development concerns cannot truly be investigated by themselves, since concerns tend to affect one another. Successful and precise product development supported via multiple views requires that common assumptions and definitions are recognized and maintained in a consistent fashion. In other words, having views with inconsistent assumptions about a system's expected environment reduces their usefulness and possibly renders invalid solutions based on them.

Developing software systems therefore requires more than what general-purpose software development models can provide today. Development is about modeling, solving, and interpreting, and in doing so a major emphasis is placed on mismatch identification and reconciliation within and among diagrammatic and textual views. Our work introduces a view integration framework and demonstrates how its activities enable view comparison in a more scalable and reliable fashion. Our framework extends the comparison activity with mapping and transformation to define the 'what' and the 'how' of view integration. We will demonstrate the use of our framework on the Unified Modeling Language (UML), which has become a de-facto standard for object-oriented software development. In this context we will describe causes of model inconsistencies among UML views, and show how integration techniques can be applied to identify and resolve them in a more automated fashion. Our framework is tool supported.

1 Introduction

1.1 Overview

Boehm and Ross said, “your project will succeed if and only if you make winners out of all critical stakeholders” [Boehm and Ross 1989]. This notion, coined in requirements engineering, implies that it is vital to identify critical stakeholders, capture their concerns and goals, and resolve conflicts between them to ensure that the software system under development meets everyone’s expectations. By stakeholder we mean an individual or a group that shares concerns or interests in the system (e.g., developers, users, customers, etc.). Software views, such as diagrammatic and textual views in architecture and design, assist the modeling of concerns. It is thus not surprising that recent standards, such as the *IEEE Recommended Practice for Architectural Description* [IEEE Architecture Working Group 1999] (P1471), advocate using architectural views to address stakeholder concerns. Concerns can be of different origins: (1) they can be goals that reflect wishes and expectations of stakeholders; or (2) they can be conflicts that reflect clashes between those goals. To address concerns, the P1471 standard suggests the use of views, following a widespread practice in software and systems modeling. Views deal with concerns in the following manners:

- Views separate concerns and reduce their overall complexity,
- Views describe and analyze concerns to evaluate the feasibility of stakeholder goals, and
- Views assist in the identification and resolution of conflicts among concerns.

The focus of this work is geared towards architecture, design, and implementation views since those views describe software systems within the boundary of their expected working environment. Architecture and design views need to be capable of describing functional and non-functional aspects of software systems. Among non-functional aspects we include software properties such as feasibility, security, maintainability, performance, reliability, cost, schedule, or interoperability.

Since architecture and design modeling goes hand in hand with requirements modeling, it follows that architecture and design views should be used to generally validate concerns early on (as compared to identifying problems in the coding/testing stage resulting in potentially higher costs in fixing

them [Boehm 1981]). Architectural and design views are thus *synthesized* in response to concerns and are *analyzed* to validate those concerns.

Currently, we are in the fortunate situation of having ample modeling support available. Researchers and practitioners alike have built strong and powerful foundations for modeling software development issues, covering all activities in the software life cycle from requirements engineering (e.g., [Carmel et al. 1983], [Conklin and Begeman 1988], [Dardenne et al. 1993], [Finkelstein et al. 1991], [Jackson 1995], [Mullery 1979], [Potts and Takahashi 1993], [Robertson and Robertson 1999], and [Sommerville and Sawyer 1997]) to architecture and design (see Sections 2.2 and 2.5), to coding and maintenance. Although most of those views do not cover development concerns in a comprehensive manner, they have nevertheless shown great promises in addressing individual software difficulties (and complexity). In particular, architectural models stand out in their innovative way of handling automated analysis and simulation capabilities, enabling the identification and evaluation of potential risks early on.

The major drawback of the concept of views is that development concerns cannot truly be investigated all by themselves. Instead development concerns tend to affect one another. If a set of issues about a modeled system is investigated, each one through its own views, then the underlying correctness requires that assumptions and definitions common to multiple views are recognized and maintained in a consistent fashion (consistency issue). It seems, however, that the perceived benefits of using software development views (models) are under-realized since the independent nature of views hinders the integration of their results. This is a serious dilemma since the independence of views (models) is a desirable property because it allows development concerns to be addressed separately and individually. Views thus enable closed-world environments that separate concerns. On the other hand, this independence manifests itself also in the model's inability to carry-over information from its first definition (specification) to its subsequent usages. Therefore, the disruption of the development flow caused by the gap between multiple views weakens their benefits. After all, regardless how pretty software development models look or how effective they are in modeling *individual* concerns, they do not add any value to the final product unless the information specified through them can somehow be transitioned into the final product.

Despite the downside, views are still the only major mechanism in simplifying software development by reducing its complexity [Brooks 1987] [Rumbaugh et al. 1999] [Nuseibeh 1994]. What makes software so complex and so difficult to grasp is that the amount of information loaded onto a single person is vastly exceeding the capabilities of the human mind. We are not able to handle thousands of pieces of information at any given time. Instead it seems that the human short-term memory is quite limited in that respect. The 7 ± 2 rule is a well-known example. This rule states that the human short-term memory can usually only handle 7 new items (plus or minus 2) at a time. The separation of concerns into views is a powerful tool in allowing software developers to reduce the amount of information they have to deal with at any given time [Tarr et al. 1999]. It has been recognized that “it is not the number of details, as such, that contributes to complexity, but the number of details of which we have to be aware at the same time” [Siegfried 1996].

Views handle software complexities by allowing development concerns to be addressed, solved, and interpreted individually. Today, unfortunately, the inclusion of modeling information (e.g., from domain, architecture, and design) into the final product frequently has to be done through manual interpretation and conversion of that information. For instance, a programmer has to read the design specifications and realize them through a programming language. We speak of an information gap causing a discontinuity of the natural flow of software development. For modeling this entails several challenges:

- Need for support of a broad set of concerns,
- Need for validation capabilities to ensure consistency between those views, and
- Need for an integrated toolset to support modeling via multiple views.

1.2 Motivation

The motivation of this work is in enabling view integration. View integration allows working with multiple views without having to live with their negative side effects caused by information discontinuity (e.g., manual, repetitive labor, and inconsistencies). View integration, which enables consistency and continuity between views, cannot easily be guaranteed since views embody information

redundancies (information overlap). Redundancy is a side effect of the closed-world environment that views create. It implies that information and assumptions common to multiple views must be replicated among all views that need them. For view integration this implies that model redundancy is the primary cause for inconsistencies among multiple views due to information replication. Model redundancy is also the primary cause for information discontinuity since replication among views is required but not automated. There are three basic choices on how to handle redundancy:

- (1) Create fully orthogonal views
- (2) Limit the domain
- (3) Bridge the information gap

The first option of creating fully orthogonal, non redundant views is likely the most effective form of dealing with redundancy since this approach circumvents the problem altogether. The drawback of this approach is that creating orthogonal views is both infeasible and impractical. Understanding the reasons for this returns us to the issue of stakeholder concerns. We argued that views are needed to address concerns (and not vice versa). It follows that views and concerns are inevitable intertwined which implies that views can only be as orthogonal as the concerns they are modeling. Views, therefore, inherit redundancies from the concerns they are addressing. Option one is not a viable alternative to view integration.

The second option (limit the domain) avoids redundancy by having all views implicitly share the same domain assumptions. The strength of this approach is in having domain-specific views that are powerful and concise. The drawback, however, is that if interoperability across domains is required, the view integration problem is back again, potentially worsened due to the lack of “implicit” domain knowledge or assumptions that were not explicitly modeled.

The third option handles redundancies by building communication links (view connectors) between views. Using that option implies accepting views with all their benefits, but, also with all their flaws. However, those flaws may be mitigated in the form of automated connectors that enable continuous information flow between views. Option three, therefore, bridges the information gap in an

explicit manner via view connectors. We have identified two major threads to support automated communication across multiple views:

- (1) Automated synthesis to enable view transformation
- (2) Automated analysis to identify view mismatches (inconsistencies)

Automated synthesis bridges the information gap between views by supporting the transformation of information that those views have in common. For instance, a common form of automated synthesis is code generation given some design specification. The advantage of automated synthesis is that same or similar information need not be captured multiple times (manually) but can instead be transitioned automatically between views (e.g., design information that can be transitioned automatically into code). Automated synthesis therefore replicates information needed in other views (redundant information) and provides information continuity across multiple views. The benefit of using automated synthesis is a reduction (or even elimination) of manual, error-prone, and repetitive activities in capturing recurring modeling information.

Since automated synthesis is often infeasible, automated analysis may be used to bridge the information gap by enabling information comparison between views. A common form of automated analysis is type and constraint checking between (formal) specifications. The advantage of automated analysis is that inconsistencies between views are identified and even pinpointed automatically (e.g., a type error in a formal specification). Automated analysis, therefore, compares replicated (redundant) information across multiple views and provides consistency feedback. The benefit of automated analysis is a reduction of manual, error-prone, and repetitive activities in validating the consistency and integrity of modeling information.

1.3 Contributions

The main contribution of this work is a framework for view integration. We describe this framework and its major activities in Section 7. Because of the abundant number of views currently in existence we decided to follow a breadth and depth approach in dealing with the integration problem. Our approach had to cover significant breadth in order to come up with a framework that was generic enough

so that it would scale to numerous situations and types of views. Our approach also had to provide significant depth in order to come up with a framework that was specific enough to be useful. Since the effort of integrating views rises exponentially (see Section 3.6) it was not possible to provide in-depth coverage for all selected views. We, therefore, chose one view category and studied its integration complexities and scalability issues in more detail.

To test our framework, we chose the Unified Modeling Language (UML) [Rumbaugh et al. 1999] [Booch et al. 1999], which, to date, has had little integration support. In particular, we chose the class, object, sequence, and statechart views of UML to ensure breadth coverage (Section 5) as well as the class and object views for depth coverage. We chose class, object, sequence, and statechart views for breadth coverage because we wanted to have at least one view species for each major view dimension (see Section 5.4). The rationale for the views we chose for depth coverage was that we wanted a situation where simple comparison would not suffice (e.g., as in the case of comparing class views of different levels of abstraction—see also Sections 7.3.1 and 7.5.2).

Another contribution of this work is an analysis of UML's suitability for dealing with view integration issues via automated analysis and synthesis. In particular, we wanted to investigate how well the UML meta-model would adapt to our integration needs and how well our framework could be fitted on top of UML. Although we built the framework with UML in mind, our goal was to remain as generic as possible and to allow other views to be integrated as well. For example, our work on C2-to-UML integration in [Egyed and Medvidovic 2000] talks about how to use our framework with other types of views where we show the integration between a design language (UML) and an architecture description language (C2) [Taylor et al. 1996].

We have analyzed techniques for both synthesis and analysis, and have found ways to automate parts of them. Although both are equally important for view integration, the emphasis of this work is more geared towards automated analysis. Section 7.3 will, however, also discuss that automated synthesis is often an enabling technology for analysis (this is one of the reasons why we chose class diagrams for an in-depth study) and as such we will revisit synthesis in Sections 7.3 and 7.5.

In order to validate our framework, we followed multiple paths (see Section 11). First, we validated its ability to handle inconsistencies among the four types of views mentioned above. Then we validated our framework's ability to include a development view for which it was not intended initially. To this end, we integrated the UML class and object views with the C2 architecture description language. Furthermore, we validated the in-depth solutions and associated tool on various real applications for which we either had models or were able to derive them (e.g., via reverse engineering). We also evaluated the complexity of our view integration approach by analyzing its scalability. Besides our framework, we also evaluated UML and its ability to deal with view integration in general and our approach in particular. It was our hypothesis that UML was not created with view integration in mind and this work indeed identified major deficiencies in UML's ability to support view integration (Section 11.1.6).

1.4 Background Information

The absence of view integration is not a new discovery. Quite the contrary, many software modeling approaches talk about the need of keeping model(s) consistent. Sometimes, process models provide additional guidelines on what activities one can do to improve the conceptual integrity of models. For instance, a case study in using the WinWin Spiral Model [Boehm et al. 1998] suggests using Architecture Review Boards [AT&T 1993] after the LCO (life-cycle objectives) and LCA (life-cycle architecture) stages [Boehm 1996] to verify and validate the integrity of analysis and design. Similar viewpoints are given by other researchers:

- Sage and Lynch [Sage and Lynch 1998] describe various aspects of integration (enterprise wide). They frequently stress "the important role that architecture plays in system integration." They present the need for three major views: enterprise view, systems engineering and management view, and technology implementation view, stressing the need to ensure consistency among these views.
- Rehtin [Rehtin 1991] emphasizes strongly the validity and consistency of requirements as well as the interface definitions. He further suggests the need for problem detection and diagnosis.
- Gacek, Abd-Allah, Clark, and Boehm [Gacek et al. 1995] present the results of a survey of people frequently involved in the software development process (developers, customers, maintainers,

acquisitioners, etc.). They found that, with respect to architects, the three major concerns were “1) requirements traceability; 2) support of tradeoff analyses; and 3) completeness, consistency of architecture.”

- The IEEE P1471 Committee [IEEE Architecture Working Group 1999] speaks of *Architecture Evaluation*: “The purpose of evaluation is to determine the quality of an architectural description, and through it assess the quality of the related architecture.” They further state the need evaluation criteria against which the architecture should be verified.
- [Kuhn 1996], [Humphrey 1995], and [Paulk et al. 1995] who defined the Software and Systems CMM (Capability Maturity Model) stress the need for integration and quality control as part of the software life cycle. Especially the SE-SMM (Systems CMM) identifies *Integration*, *Validation*, and *Architectural Evolution* as key process areas.
- Nuseibeh [Nuseibeh 1995] wrote that “inconsistency is an inevitable part of a complex, incremental software development process” and that “the incremental development of software systems involves the detection and handling of inconsistencies.”
- Wang and Cheng [Wang and Cheng 1998] propose a more rigorous object-oriented design process to deal with the shortcomings of the OMT [Rumbaugh et al. 1991] model. We share their view when they say that “the lack of a well-defined semantics for the individual [OMT] models and their integration hinders the overall development process.”
- Shaw and Garlan [Shaw and Garlan 1996] describe architecture very provocatively as being “a substantial folklore of system design, with little consistency or precision.” They further state that “software architecture found its roots in diagrams and informal prose. Unfortunately, diagrams and descriptions are highly ambiguous.”
- Perry and Wolf [Perry and Wolf 1992] realized the importance of software architectures early on and they state as one of the four major benefits of architectures that they are “the basis for dependency and consistency analysis.”

These references, and many more, talk about the need for (or lack of) view integration. Despite that, not many solutions exist on how to do automated view integration (Section 10 will discuss some of those that exist). In some cases, the details of how to enable integration are purposely omitted, such as in case of the CMM, since they do not wish to favor a particular integration approach. However, in most cases it seems that architects and designers are left ill equipped to ensure the integrity of their work.

Some of the techniques that are sometime suggested are often aimed at making people talk to each other. For instance, the *Architecture Review Board* [AT&T 1993] or the *Inspection Process* [NASA 1993] [Fagan 1986] are primarily tailored for getting the most capable people together so that they may share their findings. These techniques may follow a defined process (e.g., checklists) and may yield very effective results but the actual activities of identifying and correcting defects are still done manually without much automated (or automatable) assistance.

1.5 Outline

This work is divided up as follows:

- Section 2 discusses model-based software development in general. This section discusses relevant modeling terminologies as well as various modeling approaches.
- Section 3 focuses on the view integration problem itself. The problem of view redundancy is explained and (view) integration is defined. The section also discusses the goals and benefits of view connectors to automate synthesis and analysis. This section will also illustrate one scalability problem associated with view integration.
- Section 4 summarizes the main scope and limitations of our work. As it was indicated previously, the view integration problem is too vast to be solved at once. It is conceivable that the view integration problem is not even solvable entirely. This section will therefore re-iterate key contributions of our work but also emphasize areas that are out of the scope.
- Section 5 revisits models and views, discussing them in the context of their atomic elements as well as their types and instances. This section is foundational since it establishes view dimensions that will become very relevant later.

- Section 6 then discusses inconsistencies in the face of views and their elements. This section shows examples and lists all inconsistency types we identified.
- Section 7 introduces the basics of our view integration approach that consists of the three major components *Transformation*, *Differentiation*, and *Mapping* sitting on top of a *Repository*. This section is a general overview of some of the latter sections.
- Section 8 discusses transformation, the first of the three major components of our integration approach. Transformation converts model information to simplify validation. Transformation is subdivided into types following the view dimensions discussed in Section 5.
- Section 9 discusses differentiation, the actual consistency checking activity of our approach. This section introduces consistency rules for the inconsistency types of Section 6 and then illustrates how those rules must be applied. This section also discusses some ergonomic (e.g., human-computer interface) aspects of view integration.
- Section 10 briefly covers two other important areas of modeling; that of information capture and tracing. Both activities have important ramifications towards view integration but are considered out of the scope of this work.
- Section 11 covers the third and last important piece of our approach—the model repository. Repository design is fundamental for transformation, consistency checking, and their scalability.
- Section 12 discusses our tool UML/Analyzer that supports the in-depth part of our integration approach in the context of the object/class abstraction and consistency checking. The tool is discussed in the context of an example.
- Section 13 discusses other examples in the context of larger models onto which our approach was applied. One of these examples is discussed in detail.
- Section 14 shows related works in this area and discusses them with respect to eight criteria.
- Section 15 evaluates our work and its contributions in the context of eight evaluation criteria.
- Section 16 ,17, and 18 describe future work, conclusions, and bibliographical entries.

1.6 Summary

This chapter laid out the basic problem of view integration. We gave a brief overview on the current state of software modeling and pointed out its deficiencies. We also emphasized key contributions of this thesis and briefly described its outline. Additional background information was given to support our claim that the proposed problem is indeed important and the problem is severe enough that it needs attention. The motivation for our work is in addressing deficiencies of model-based software development that are caused by the lack of automated assistance in identifying and resolving inconsistencies.

2 Model-Based Software Development

A model of a large software system permits dealing with complexity that is too difficult to deal with directly. A model can abstract to a level that is comprehensible to humans, without getting lost in details. A computer can perform complicated analyses on a model in an effort to find possible trouble spots, such as timing errors and resource overruns. A model can determine the potential impact of a change before it is made, by exploring dependencies in the system. A model can also show how to restructure a system to reduce such effects. [Rumbaugh et al. 1999]

2.1 Software Modeling

In science and engineering, we (humans) have made use of abstraction to deal with complexities. *Software Engineering* is no exception and thus emphasizes the need for abstraction in the software development domain. Sommerville defined that software engineering is pre-occupied with “theories, methods, and tools which are needed to develop [...] software” [Sommerville 1996]. We would extend this statement to say that those theories, methods, and tools facilitate abstraction to separate concerns.

Model-based software development is all about abstraction but that alone does not solve problems. In order to solve complex problems, we need to solve problems in the abstracted model world and we then interpret model solutions in the real world. Figure 1 shows this process in the field of mathematical systems theory. There a problem solver uses some mathematical formula (function $f(x)$) to translate a real world problem into a (mathematical) model world problem. The model problem (if it is simple enough) is then solved to yield a model solution. Applying the translation backward will result in a solution that is applicable in the real world. Should the model problem still be too difficult to solve, the same technique can be applied recursively again (the previous model problem becoming the real problem). If the refined model problems are easier to solve than the real problems, we will eventually find a model problem that is simple enough to be solved directly. Figure 1 shows that, in mathematical

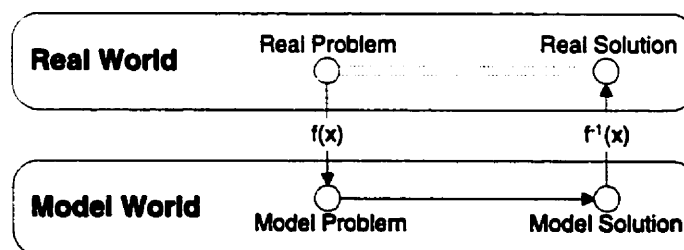


Figure 1: Mathematical Systems Theory

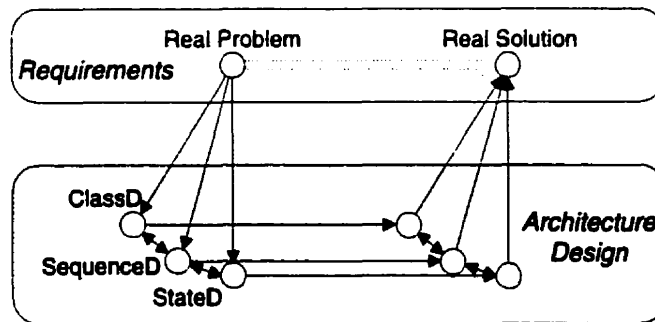


Figure 2: Software Engineering Theory

systems theory, finding a solution for the real problem is reduced to finding a solution for the model problem. The principles, which guide the mathematical systems theory in Figure 1, are also visible in software development (see Figure 2).

Figure 2 shows the task of going from a real software problem (e.g., requirements) to a real software solution (e.g., source code). Since solving real problems directly is often too difficult for large software projects [Brooks 1995] (although often attempted), models can provide enormous simplifications. In Figure 2, diagrams are used to represent the model problem and the model solution (analogous to Figure 1). Those diagrams could be class diagrams, sequence diagram, state diagrams, or other types of diagrams (to name just a few). The real picture of software modeling is of course more complex since it usually involves multiple levels of refinement. Nevertheless, the basic idea is still the same. The model-based problem solving process uses intermediate models to simplify a more complex solving task. The number of diagrams required be may increased as the complexity of the problem increases.

2.2 Models and Views

So far we have used the terms *model* and *views* in many ways and even interchangeably. We referred to them as being diagrams, languages, and even mathematical representations. This section defines these words in more detail and discusses what they imply.

The IEEE Draft Standard 1471 [IEEE Architecture Working Group 1999] refers to a view as something that “addresses one or more concerns of a system stakeholder.” By stakeholder we mean an

individual or a group that shares concerns or interests in the system (e.g., developers, users, customers, etc.). A model is the union (collection) of all views related to the same problem (e.g., related to the same software project) and views are partial descriptions of that model in the context of stakeholder concerns. A *view* is, therefore, a piece of the *model* that is still small enough for us to comprehend but that also contains all relevant information about a particular concern. As such, the diagrams depicted in Figure 2 really show views of the problem/solution model.

Ideally, there would only be a small set of views covering all development needs and concerns. In reality, there are many types of views. Recent developments in architectural modeling showed that even more types of views (e.g., architecture description languages) are needed to address other development needs and concerns that were not addressed well before. The current advances in architecture modeling are a strong indication for that. Given the many needs and concerns that can arise during a software life cycle, it is not surprising that a myriad of specific models and views are in existence, many of which have their respective advantages that also justify their continuing existence.

Besides modeling concerns, another reason for the diversity of views lies in their need of having to address different audiences. Possible audiences include architects, analysts, coders, maintainers, testers, users, customers, and many more. If the audience is, for instance, a customer or user, then the emphasis of a model is in having descriptions that are simple and easy to understand. Although developers would similarly benefit from simple models, reality shows that those simple models frequently lack the precision required to describe a problem and/or solution in detail. Likewise, development models are often not ideal for handling customer or maintainers needs, and so forth.

2.3 Common Models and Views

“Until relatively recently, the most commonly used software design strategy involved decomposing the design into functional components with system state information held in a shared data area” [Sommerville 1996]. Sommerville goes on further, stating that “it is only since the late 1980s that ... alternative, object-oriented design has been widely adopted.”

Numerous software modeling techniques and methodologies (collection of techniques) have been developed in the past decades. Among the most notable methodologies (both functional and object-oriented) are Booch's Object-Oriented Design Method (BOOD) [Booch 1994] [Booch 1996], Coad-Yourdon Method [Coad and Yourdon 1991a] [Coad and Yourdon 1991b], Controlled Requirements Expression (CORE) [Mullery 1979], Data Flow Models (DFD) [DeMarco 1978], Entity-Relationship Models (ERM) [Chen 1976], Jackson Design Method (JSD) [Jackson 1983], Object Modeling Technique (OMT) [Rumbaugh et al. 1991], OOSE Method [Jacobson et al. 1992], SADT [Ross 1977] [Schoman and Ross 1977], Shlaer-Mellor Method [Shlaer and Mellor 1989] [Shlaer and Mellor 1991], Structured Systems Analysis and Design Method (SSADM) [Cutts 1988] [Weaver 1993], SRD [Orr 1981], and Warnier-Orr Method [Warnier 1977]. It is out of the scope to discuss those development methodologies. There are numerous comparative studies about the features, strengths, and weaknesses of these techniques such as [Sommerville 1996], [Carmichael 1994], [Sheard and Lake 1998] and [Song and Osterweil 1992].

The types of models supporting software development can be very distinct in their characteristics. Many models are (at least partially) graphical in nature, yet other models are more textual, spanning the use of plain English and some types of formal or semi-formal language. Most of those modeling techniques have shown promise in at least some aspect of software development. It was only natural that people started to combine individual models into more comprehensive development methodologies. Those methodologies emphasize a (usually) small number of views covering the most important and interesting aspects of development. With time, the community was even able to standardize some of those development methodologies, providing more generality that, in turn, increased their applicability to even larger software development domains. The Unified Modeling Language (UML) [Rumbaugh et al. 1999] is the result of one such endeavor to unify object-oriented analysis and design techniques and their associated diagrams into a single methodology. UML supports a series of diagrams (views) and provides a common meta-model underneath them.

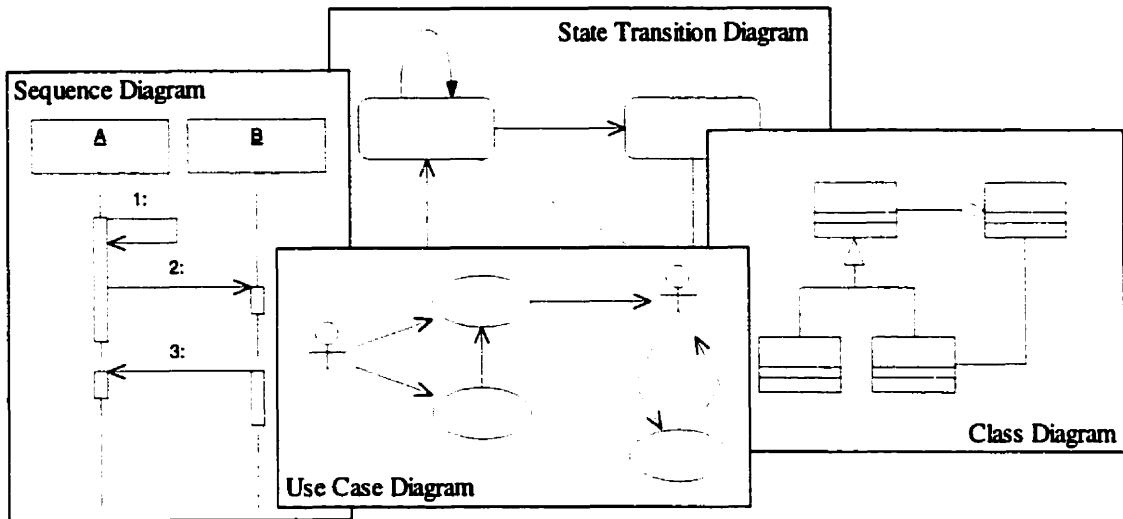


Figure 3: Some of Diagrammatic Views support by UML

2.4 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) “is the successor to the wave of object-oriented analysis and design (OOA & D) methods that appeared in the late ‘80s and early ‘90s. It most directly unifies the methods of Booch, Rumbaugh (OMT), and Jacobson...” [Fowler 1997]. UML is a generic modeling language that aims at supporting a broad range of development concerns.

In the remainder of this work, we will primarily use the Unified Modeling Language (UML) to illustrate our view integration approach as well as to describe examples. In this section we will briefly describe UML, which is currently the leading object-oriented analysis and design model. UML supports a variety of design views, some of which object-oriented in nature (e.g., class diagrams [Booch 1994] [Rumbaugh et al. 1991]) and others more functional (e.g., statecharts [Harel 1987]). For the most part, views in UML are graphical; however, there are also textual descriptions, mostly in the form of add-ons to the graphical notation (e.g., Object Constraint Language [Warmer and Kleppe 1999]). UML is the result of a collaboration between numerous companies and OO modeling experts and it borrows heavily from Booch [Booch 1994], OMT [Rumbaugh et al. 1991], and other OO models such as [Coad and Yourdon 1991a], [Coad and Yourdon 1991b], and [Jacobson et al. 1992].

“UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems” [Booch et al. 1999]. These different but overlapping uses of the model can only be achieved by supporting a variety of views. Some of those views (diagrams) are schematically depicted in Figure 3 showing a sequence diagram (left), a class diagram (right), a use-case diagram (middle-bottom), and statechart diagram (middle-top).

These and other views are briefly explained below. A more detailed discussion of UML diagrams is outside the scope of this thesis. We assume the reader to be familiar with the basic UML design concepts. Please refer to [Rumbaugh et al. 1999] or the “OMG Notation and Semantics Guide for UML” [OMG 1999] for more detailed descriptions (this work uses UML version 1.3). Additionally, [Fowler 1997] provides a brief overview of UML.

- **Use Case:** Depict the interaction between users and components or between components. In doing so, use cases provide a high-level view of the usage of a system and frequently shows the interaction of multiple functions of that system. For instance, the task of editing a document involves the functions *open document*, *edit document*, and *save document*.
- **Interaction** (e.g., Sequence and Collaboration diagrams): Sometimes also referred to as Mini-Uses. Interaction diagrams show concrete examples of how components communicate. They can often be seen as test cases and depict sequences of interactions (e.g., calls). A call can refer to user interface invocations (e.g., open file) or to component interactions.
- **Objects and Classes** (e.g., Class diagrams): Classes are the most central view in UML. Class diagrams depict the relationships between classes and objects, which are the smallest stand-alone components in OO. Class relationships further depict their generic interactions (e.g., aggregations, dependencies, etc.).
- **Packages** (e.g., Package diagram): Packages are used to group classes into layers and partitions. As such they show system decompositions.
- **State Transition** (e.g., State and Activity diagrams): Are used in UML to describe the states that classes can go through. In UML, state diagrams are bounded to individual classes. Activity Diagrams

are a generalization of state diagrams in that they can also be used to depict events or other 'transitional' elements across class boundaries.

- **Deployment** (e.g., Deployment diagrams): Shows the physical components of the system during deployment. It presents a physical view of the system and is, therefore, frequently used to depict the component dependency of the actual implementation.

The Object Constraint Language (OCL) [Warmer and Kleppe 1999] supports UML and provides some limited integration within and between UML diagrams. OCL is a formal language for expressing constraints on model elements in UML (see Section 5 for more information on model elements).

2.5 Architecture Description Languages (ADL)

The development of design methodologies such as the ones listed in Section 2.3 had stagnated in the late 80's. Until then, a strong development driver of newer design methodologies was comprehensiveness to increase their concern coverage and applicability. With the emergence of architecture description languages (ADLs) in the mid-nineties, a reverse trend started. ADLs, contrary to methodologies such as SSADM and SADT, are very specialized and often only address specific concerns (e.g., reliability, presence of deadlocks, dynamism, etc.). Although, ADLs are very restrictive in their scope, they are nevertheless extremely powerful in analyzing and simulating their respective niches [Medvidovic et al. 1999b]. More general-purpose languages often lack such extensive analytical features. With UML, a new design methodology has emerged that does not have strong analytical capabilities. Researchers and practitioners, however, have proposed to extend UML to support special-purpose modeling (via ADLs) combined with general-purpose modeling (via UML). This can be done by using UML's extensibility mechanism and OCL to represent new types of model elements and their semantics in UML (e.g., [Hofmeister et al. 1999], [Medvidovic and Rosenblum 1999], [Robbins et al. 1998], [Selic et al. 1994], and [Selic and Rumbaugh 1998]). For instance, in [Abi-Antoun and Medvidovic 1999] and [Robbins et al. 1998] it is shown how the C2 architecture description language [Taylor et al. 1996] is represented and transformed into a UML description.

To define what software architecture is, is difficult since not many people can agree on a single definition. Perry and Wolf [Perry and Wolf 1992] describe architectures as having elements, form, and rationale. *Elements* describe the building blocks of architectures and thus denote *what* is built, *form* describes the configuration of *how* model elements are interrelated and communicate, and the *rationale* gives the reasoning behind the chosen architectural decisions (*why*).

Shaw and Garlan [Shaw and Garlan 1996] give a more elaborate definition and write that “the architecture of a software system defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database-accessing protocols, asynchronous event multicast, and piped streams” [Shaw and Garlan 1996].

Recently, there have been attempts in standardizing architectures and their usages. The IEEE Draft Standard 1471 [IEEE Architecture Working Group 1999], one such endeavor, provides the following definition for software architecture:

*Every system has an architecture, defined as follows:
An architecture is the highest-level conception of a system in its environment where: the 'highest-level' abstracts away from details of design, implementation and operation of the system to focus on the system's 'unifying or coherent form'; 'conception' emphasizes its nature as a human abstraction, not to be confused with its concrete representation in a document, product or other artifact; and 'in its environment' acknowledges that since systems inhabit their environment, a system's architecture reflects that environment. [IEEE Architecture Working Group 1999]*

Above definitions are all rather vague and apply to “architecture” as well as to “requirements,” “design,” or “operational concept.” Shaw and Garlan’s definition actually uses the term *design* as part of their definition of software architecture. Also, all of the above definitions provide to little emphasis on the analysis and interpretation of architectural descriptions. Of course, architectural descriptions are supported by powerful analysis and simulation capabilities that help resolve stakeholder concerns; however, dealing with architectural descriptions also requires analyzing and verifying the conceptual

integrity, consistency, and completeness of those descriptions in the context of requirements, design (lower-level), and implementation. This is where our work fits in. We find that ADLs have accomplished in the small what we hope more general-purpose languages will accomplish in the future. As such, ADLs do not just provide modeling languages to compose systems but also provide concepts and techniques on how to analyze and validate them. Our work additionally shows how various modeling languages (from UML to ADLs) can be used together in a consistent manner. After all, no single model is adequate in addressing all stakeholder concerns.

Examples of architectural models (ADLs) are ACME [Garlan et al. 1997], AML [Wile 1999], C2 [Taylor et al. 1996], Chemical Abstract Machine [Inverardi and Wolf 1995], Rapide [Luckham and J. Vera 1995], Darwin [Magee and Kramer 1996], SADL [Moriconi et al. 1995], and Wright [Allen and Garlan 1997]. Initially, we had not included ADLs into our view integration framework; however, we found that there is a great benefit in combining general-purpose modeling languages (e.g., UML) with specific-purpose modeling languages (see also [Medvidovic and Taylor 2000] for a comparison). We have therefore investigated ways of combining ADLs and UML in order to improve modeling. In [Egyed and Medvidovic 1999], we show how our view integration technique can be applied for consistency checking between UML class/object diagrams and the C2 architecture description language. We see our work on UML and C2 integration as initial proof of concept that our approach is also applicable beyond UML (see Section 11.1.5).

2.6 Stakeholders and Model Life Cycles

Modeling architectures and designs implies satisfying a number of potentially conflicting concerns (see Section 1). Table 1 (taken from [Gacek et al. 1995]) summarizes major architecture-related concerns with respect to goals and wishes of system stakeholders (note that we believe those concerns to be equally relevant for design). Those concerns can then serve as evaluation criteria for both architecture and design. As the table suggests, the *customer* is likely to be concerned with getting first-order estimates of the cost, reliability, and maintainability of the software based on its high-level structure. This implies

Table 1. Stakeholder Concerns as Architecture Evaluation Criteria from [Gacek et al. 1995]

Stakeholder	Concerns / Evaluation Criteria
Customer	<ul style="list-style-type: none">• Schedule and budget estimation• Feasibility and risk assessment• Requirements traceability• Progress tracking• Product line compatibility
User	<ul style="list-style-type: none">• Consistency with requirements and usage scenarios• Future requirement growth accommodation• Performance, reliability, interoperability, other quality attributes
Architect and System Engineer	<ul style="list-style-type: none">• Product line compatibility• Requirements traceability• Support of tradeoff analyses• Completeness, consistency of architecture
Developer	<ul style="list-style-type: none">• Sufficient detail for design and development• Framework for selecting / assembling components• Resolution of development risks• Product line compatibility
Interoperator	<ul style="list-style-type: none">• Definition of interfaces with interoperator's system
Maintainer	<ul style="list-style-type: none">• Guidance on software modification• Guidance on architecture evolution• Definition of interoperability with existing systems

that the architecture should be strongly coupled with the requirements to evaluate if the architecture can satisfy those requirements (see also [Boehm et al. 1998]).

Users need software architectures to clarify and negotiate their requirements for the developed software system, especially with respect to future extensions to the product. Users will be interested in the impact of the software structure on performance, usability, and compliance with other system attribute requirements.

Architects and designers are concerned with translating requirements into high-level architectures and designs. Therefore, their major concerns are about consistency between the requirements and the architecture and design during the process of clarifying and negotiating the requirements of the system [Gruenbacher et al. 2000]. *Developers* are concerned with getting an

architectural specification that is sufficient in detail to satisfy the customer's requirements but that is not so constraining as to preclude different approaches or technologies in the implementation. Developers then use the architecture (design) as a reference for developing and assembling system components, and provide a compatibility check for reusing pre-existing components. *Interoperators* use the software architecture as a basis for understanding (and negotiating about) the product in order to keep it interoperable with existing systems. The *maintainer* will be concerned with how easy it will be to diagnose, extend, or modify the software.

Modeling with UML can be seen in Figure 4. The figure shows UML views (as well as some related views) that are needed from an architect's or designer's point of view. The arrows depict the dependencies between views. The figure should not be taken too literally since we tried to capture the major flows of dependencies only. For instance, the picture shows that the classes and objects affect the implementation (e.g., code in C++) but not vice versa. This is, of course, not always true. There are cases where the implementation may trigger changes in the design and architecture (e.g., due to choice of COTS product). As a general rule, it is good practice to anticipate these dependencies and address them via prototyping and analysis. Further, the associations of the development artifacts (such as classes, use

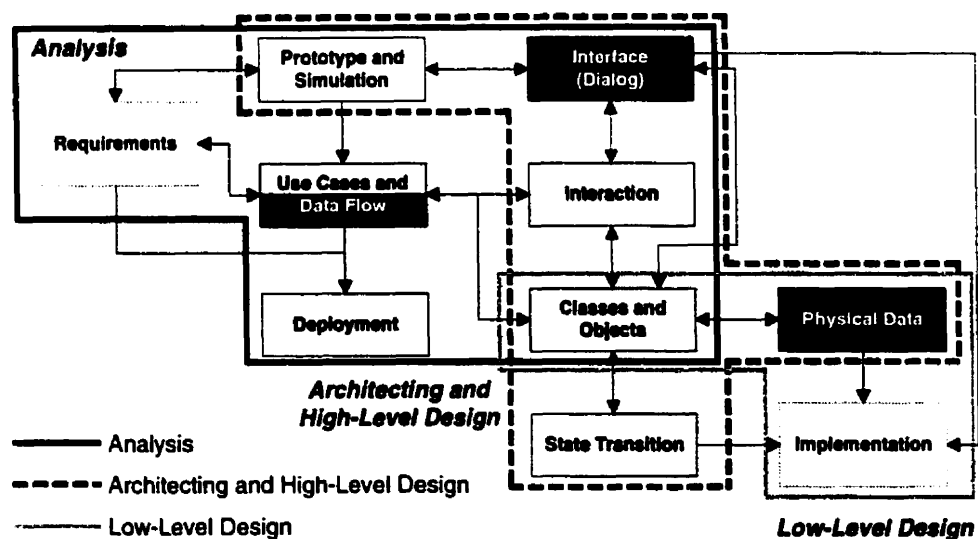


Figure 4: Architectural Views in UML

cases, etc.) to the major phases of the life cycle can indicate primary associations at best. Again, we tried to capture the major associations of those development artifacts and the views in which they are frequently used. It is this ambiguity in how to associate and relate development artifacts that already poses our first problem in model-based software development.

Traditional life cycle models such as the waterfall process model are less useful in object-oriented software development because OO activities overlap with one another potentially causing clashes between process phases. We tried to indicate this in Figure 4 where some development artifacts, such as classes and objects, are used and shared extensively during most of the development process. This ambiguity, in the definition of development stages and phases, is however also a good thing since it provides some continuity between the life cycle stages and, thus, brings the development stages closer. The conceptual breaks, which so frequently happen between the analysis and design stages, are eased.

Figure 4 also shows that there are multiple views needed to address software development, and that modeling languages such as the UML support some of them. The main message we try to convey with Figure 4, and with the above discussion, is that UML views relate to one another, but are not integrated well enough to allow their interaction. We speak of an information gap in that information in different views may relate, that relationship may however not be explicitly captured.

2.7 Information Gap and Information Discontinuity

Previously, we discussed that in order to control software complexity, we utilize abstraction as a strong driving force for software development. Abstraction comes in different dimensions. For instance, software development projects use processes to create phases and milestones to synchronize and divide activities. Views (e.g., as enabled through textual and diagrammatic representations) have typically complemented processes in achieving abstraction. However, the underlying concepts of how views enable abstraction have strong similarities to those of processes. Both try to depict problems in a discrete manner through stakeholder concerns.

Currently a major challenge in software development is how to best utilize information captured during the project life cycle. We have observed that significant portions of model descriptions do not find

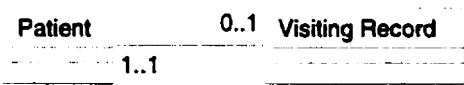
their way directly into the end product. For instance, requirements capture has been recognized as being a fundamental part of software development; however, requirements rarely find themselves automatically included in the source code or the user's manuals, the end-products of software projects. Instead, requirements tend to be by-products of software development and ultimately are only used for decision-making along the way (e.g., in how to do the design).

What this implies is that information captured about a system often cannot be automatically carried through the entire project to find itself in the final product. We speak of a degradation of project specific information over time. This degradation does not imply loss of information. For instance, requirements or design information stay available throughout the life cycle of a project, but frequently in a form that is unsuitable for further processing. Thus, model information often lacks continuity in that it cannot be transitioned automatically from its definitions to all its usages.

Consider the example in Table 2. There, the cardinality of a relationship between *Patient* and *Visiting Record* is represented in three different forms. The first case shows a requirement describing their relationship, the second case shows a UML class diagram depicting that same relationship in a graphical form, and finally, the third case shows the corresponding partial pseudocode. Even in a rather trivial example such as this one, it is not straightforward to see how the information entered in the requirements could find its way into the design without human intervention. Similarly, it is not easy to see how the design information can be transitioned into the code in an automated fashion.

In both process and view oriented development projects, the main cause of information discontinuity is the discrete basis of information capture. All information should be captured and

Table 2. Discontinuity of Project Information over Time

Requirements:	There shall be no more than one visiting record per patient and each visiting record belongs to exactly one patient.
Design:	 <pre> classDiagram Patient "0..1" -- "1..1" VisitingRecord </pre>
Implementation:	<pre> if (get_patient(aVisitingRecord) = NULL) then raise exception; </pre>

maintained continuously, but instead we tend to develop software projects through the use of artificial borders placed by views and processes. The disadvantage of too strictly defined milestones for processes have long been recognized as problematic (e.g., waterfall model); views exhibit a similar downside. Views are best used within a defined group of stakeholders, at defined times, and for defined problems. For instance, the class diagram view (a popular object oriented design view) is primarily used by architects and designers during the design phase. The use case view (a UML requirements capture view) is mostly used by customers, users, and analysts during the initial phases of the projects. Although the timeline of activities may be blurred, the basic premise stays the same. The result is a discontinuity in that information entered into one view (diagram) must, at a later time, be converted into other views to be useful. Take for instance the extreme case of requirements and code. Clearly, today we have hardly achieved the ability to automatically generate code from requirements, even when requirements are defined in great detail. This case denotes a discontinuity between requirements and code [Medvidovic et al. 2001].

The existence of this gap does not imply that process- or view-oriented development approaches have failed in their quest to simplify software development. We already discussed the benefits in separating concerns. However, both approaches have failed to live up to their promises, even though, we still need them to reduce the complexity of software projects. Views divide huge projects into more comprehensible pieces that can be understood and at least partially solved on their own. On the downside, the discontinuity in transitioning project-related data causes same or similar information to be captured multiple times for different views, which in turn causes significant extra work. Furthermore, recapturing information may also introduce additional errors, especially since the activity of replication is often a manual one.

2.8 Information Degradation

The information gap denotes a discontinuity in how information can be carried over from previous development activities. The severity of this gap varies. In some cases, some information may be carried over whereas in other cases, information must be recaptured. For instance, lower-level design

views (e.g., class diagrams) can often be used to generate skeleton code; however, interconnectivities of classes depicted in those diagrams frequently get lost (e.g., calling dependencies). In other cases, the information gap is more severe as it was already described in the requirements-to-design/code example above. We have learned that certain milestones are particularly vulnerable to information loss [Boehm et al. 1998] [Boehm and Egyed 1999]). For instance, it is much harder to automatically carry over information from the early life cycle stage to the design stage than it is to carry over information from the design stage to the code.

Figure 5 depicts how information may degrade over time. The first example shows the ideal case of information once captured being fully transitioned into later views without information loss. The second and third cases depict the more realistic scenarios of partial and full information loss over some time (information degradation). In partial degradation some information can be salvaged for later use; in full degradation all information is lost after some time.

The last case in the figure depicts an impossibility (under a closed world assumption). It is not possible that more complete information is automatically generated over time than was previously captured. Note that we do not speak of the physical amount of information here but instead of the completeness of information (percentage). Even in case of reuse (e.g., product lines) that information must have been entered at one time and made available later on. In the latter case, we speak of continuity of product information across projects as well as models.

Figure 5 also shows that information once entered but lost may have to be reentered later on, albeit in a different form (second and third case). Also note that information is not physically lost but instead becomes unusable. Although some information may not need to be carried over (e.g., if it was only used for decision-making), the example in Table 2 showed that there are cases where information should find its way through multiple stages of the project life cycle. In those cases where information must be reentered, we encounter redundancy and work duplication that are also major causes of (view) inconsistencies.

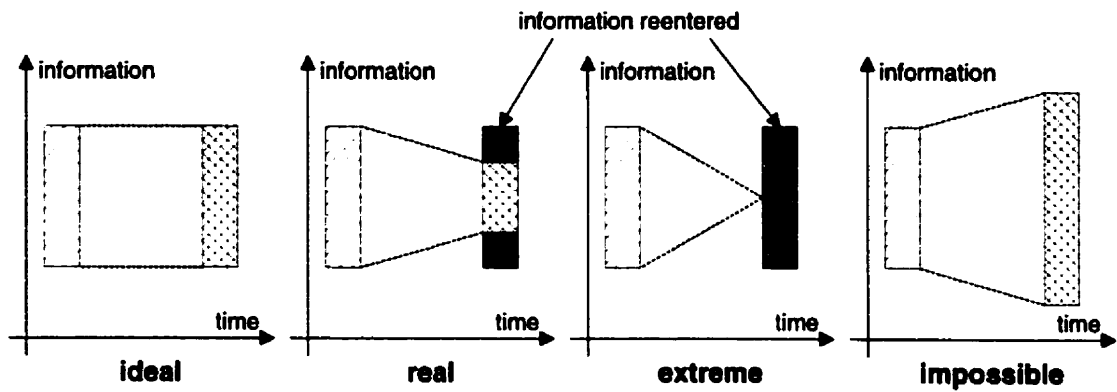


Figure 5: Information Degradation over Time

2.9 Summary

This chapter discussed the issue of model-based software development. First we highlighted the differences between models and views and gave some examples. We then discussed one such example, the Unified Modeling Language, in more detail since we will make use of it throughout this work. We also gave a brief overview of architecture description languages since we used them to validate our work outside the UML domain. Finally, we re-emphasized the existence and the problem of the information gap that results in an information discontinuity. It is that discontinuity that decreases the overall usability of models and views. View integration, the emphasis of this work, discusses ways on how to address that problem.

3 Model Integration

This section will discuss view integration and its problems and challenges. Since redundancies are the primary cause of inconsistencies and hinder integration, view integration relies heavily on locating related information (redundancies).

3.1 What are Model Redundancies?

The major drawback of model-based software development is information discontinuity. We already demonstrated that concerns, which are modeled through views, cannot be (and should not be) separated. If a set of issues about a modeled system is investigated, each through its own views, then the overlying correctness requires that common assumptions and definitions are recognized and maintained in a consistent fashion. However, consistency between views can not be easily guaranteed since views embody information redundancies (information overlap) to enable a closed-world environment. Having inconsistent assumptions about a system's expected environment voids the correctness and usefulness of views and thus rendering invalid all solutions based on those views. Thus, for views to be useful in addressing individual concerns, the problem of information redundancy has to be addressed.

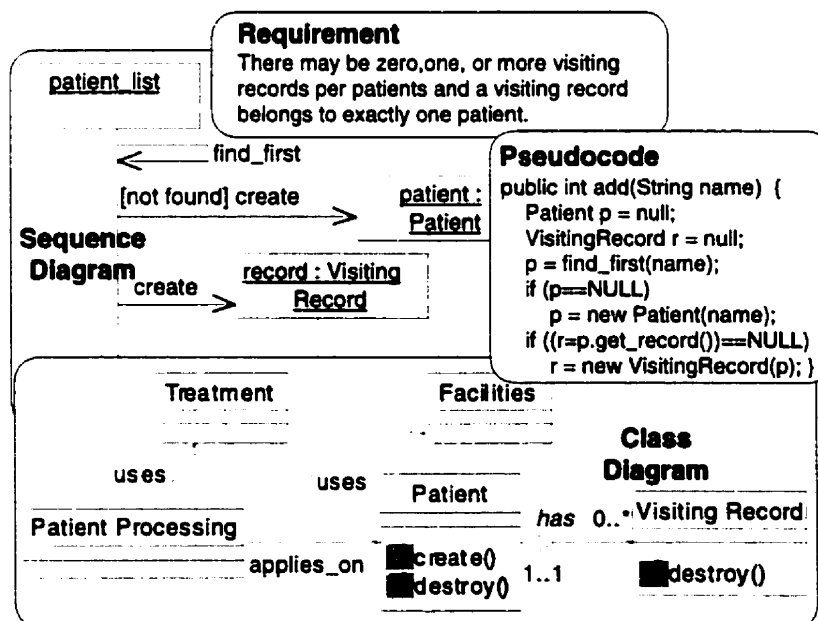


Figure 6. Different Views for Hospital System

Consider the example in Figure 6 (an extension of Table 2). The figure shows the cardinality of a relationship between *Patient* and *Visiting Record* of a hospital system. The cardinality is represented in different views. One view shows a requirement specification using natural language, another view shows the design using a UML (Unified Modeling Language [Rumbaugh et al. 1999]) class and sequence diagrams, and yet another view shows a more formal textual view using pseudocode. The example shows the redundancy between views in that same or similar information appears in different places. For instance, one can infer the cardinality between *Patient* and *Visiting Record* through all views although it is represented in different manners.

The problem of view redundancy becomes more severe because of the lack of automated support for information sharing between views. This denotes an information gap between views, which leads to an information discontinuity between related modeling elements (recall Section 2.7). The effect of the information discontinuity is that data common to multiple views is not carried over automatically. For instance, assume that the requirement in Figure 6 changes to: "There must be at least one visiting record per patient." That change implicitly causes an inconsistency with some of the other views since the cardinality in the class diagram does not yet reflect that change. The discontinuity between views is obvious in that a change in requirements is not automatically propagated to all its dependent views. Dependent views could be diagrams (e.g., designs) that implement above requirements. The side effect of the information discontinuity results in the need of having to fill the information gap in a manual fashion. Thus, if one view is changed, then this requires a manual detection of all affected modeling elements in other views as well as their manual updating to again guarantee consistency.

It is, therefore, the redundancies between views that are the primary causes of inconsistencies. Figure 7 depicts variations on how views may relate. The first scenario shows two views that do not share any information and do not otherwise depend on each other. These views can be considered fully orthogonal (unrelated).

A trivial example is of two views of separate software systems.

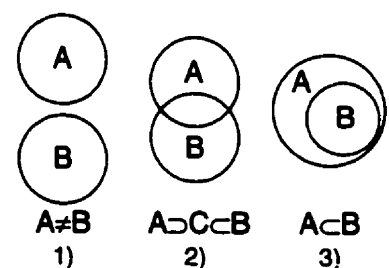


Figure 7. View Redundancy

The second scenario shows two views that overlap in the information they use or convey. These two views can be considered as depending on one another since they exhibit some redundant information. An example was already given in Figure 6 where we depicted four types of views sharing information. Each of those four views depicted information that was unique and not shared by the others (e.g., the sequence diagram shows that *patient_list* calls the *create* method only if *patient* is not found – that information is not inferable from the class diagram). However, all four views also depicted information that was common across all of them (e.g., the cardinality).

The third scenario shows the case of two views where the data used by one view is a subset of the data used in the other view. In this case, view B can be considered fully dependent on View A. Examples of such a scenario would be views describing different levels of abstraction where view B denotes a higher-level view and view A denoted a lower-level view. For consistent refinement, view A needs to describe the same information as view B, only in a lower-level of abstraction involving more detailed and elaborate descriptions (ergo more information).

There is a possible fourth scenario not depicted in Figure 7 that denotes the extreme case of both views sharing exactly the same information and no view having any additional information to present. Usually such a scenario denotes a case of an unnecessary view since there is little value in describing and maintaining two separate views without getting any additional value out of them. There are, however, exceptions. For instance, if code is generated in Java and C++ for different customers and/or platforms than that code needs to be fully redundant. Later, when we revisit automated synthesis, we will also find that transforming information from one view to another may result in such a scenario. For instance, in [Abi-Antoun and Medvidovic 1999] we find the case of an automated refinement technique from the C2 architecture description language [Taylor et al. 1996] to UML class and object diagrams. In this case, the refinement technique produces a UML design out of a C2 architecture where, initially, both are equivalent and none has additional information to present. Note that the term refinement, as Abi-Antoun and Medvidovic use it, is misleading since their technique is only a conversion from one representation scheme, C2, to another representation scheme, UML with the addition that implicit C2 concepts are explicitly stated in UML. Another example where two types of views are very similar (or equal) are UML

sequence and collaboration diagrams. Both types of views could be used to model the exact same situation using the same information. The only distinction is that aesthetically, a sequence diagram differs from a collaboration diagram. This difference in appearance may affect a stakeholder's perspective and thus justifies this case of highly redundant views.

3.2 Missing Integration in Models and Views

Having established the notion of view redundancy, we are now confronted with a major problem: How should we handle redundancy? Do we even need to care? When we described the mathematical problem solving approach (recall Figure 1) we concluded that modeling finds a solution to the real problem by finding a solution to the model problem. For that very reason, software development models were created; they serve as counterparts to mathematical models. However, are our software engineering models (like the one we showed in Figure 4) really equivalent to the mathematical model in solving problems (see Figure 8 for a comparison)?

What if the (software) model, which is created to represent the real world, is not adequate? A solution we might find to that model problem would not be correct then. This implies that we are not only confronted with the challenge of finding a (model) solution to a model problem but also we have to find a model of the real world that is adequate for our needs. This is like solving the right problem vs. solving

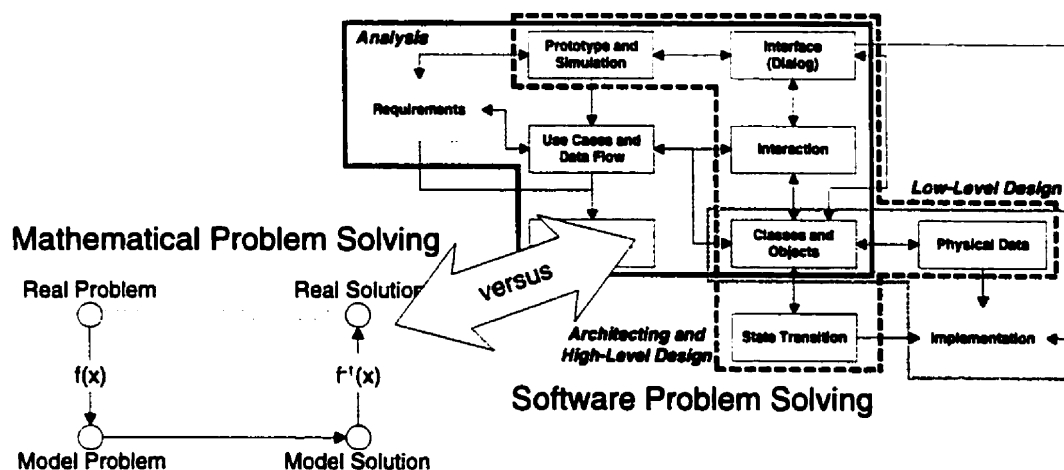


Figure 8: Two Problem Solving Approaches

the problem right [Boehm 1989]! As such, the mathematical problem solving approach is really doing three things (corresponding to the three arrows in the left side of Figure 8):

- **Model the real problem adequately**
- **Solve the model problem**
- **Interpret the model solution in the real world**

Revisiting Figure 8, which part of the software model is doing the modeling? Which part is doing the solving? And which part is doing the interpretation of that solution? We find that models generally do not sufficiently address all those issues. What this implies is that conventional software development models, such as the UML, are not sufficient in addressing all of the needs of software development. After all, what is the usefulness of an implementation of a software product if it does not satisfy the architecture or design? Similarly, of what use is the architecture if it does not satisfy its requirements?

The conclusion we draw from this case is that architecting and designing is more than what conventional development models provide. For us, *development is to model, to solve, and to interpret*. Since, modeling languages, such as the UML, just provide assistance, this work will show how they can be enhanced (integrated) to increase their usefulness. Our work can ensure that consistency and continuity is improved, thus decreasing the likelihood of defect introductions.

3.3 What is Integration?

We have used the word *Integration* or 'what it means to *integrate*' but so far we have not described it. This section will do that. The term *Integration*, as such, is part of everybody's vocabulary.

The Merriam-Webster Dictionary [Merriam-Webster 1996] defines the term *Integration* as:

The act or process or an instance of integrating: as a) incorporation as equals into society or an organization of individuals of different groups (as races) b) coordination of mental processes into a normal effective personality or with the individual's environment. [Merriam-Webster 1996]

Not surprisingly, this set of definitions is very broad and does not apply to software alone. In software engineering it applies to *technologies, organizations, and people*; it affects management,

products, humans, politics, standards, models, enterprises, and more. Sage and Lynch's work about *Systems Integration and Architecting* [Sage and Lynch 1998] provides a very comprehensive overview of what integration means in the context of software and system modeling. They found that "systems integration is an activity omnipresent in almost all of systems engineering and management." They further found that "the term lacks precise definition and is used in different ways and for different purposes in the engineering of systems."

Nuseibeh said that "separating concerns is an important step towards reducing the complexity of software systems, making them easier to develop, understand and maintain" [Nuseibeh 1994]. He, therefore, concluded that "for complex systems which have been developed from multiple perspectives or views, some form of view integration is often necessary." Jackson states it even more direct when he says that "... having divided to conquer, we must now reunite to rule" [Jackson 1990].

In software engineering, the word *Integration* is used frequently and often refers to the process of assembling components (or subsystems) into systems. As such, the term integration stands for an activity that starts later on in the life-cycle, once some software components have been developed and need assembly. Another case where the term *Integration* is used refers to the unification of standards, processes, and models. For instance, the Integrated Capability Maturity Model (iCMM) of the FAA, (which is a union of various CMM models (such as the SW-CMM [Paulk et al. 1995], SE-CMM [Kuhn 1996], and SA-CMM [Ferguson 1996]), is one such attempt to integrate existing standards into a more general standard. The Unified Modeling Language (UML) is another such case, where various object-oriented development models (Booch, OMT, and pieces of many others) were integrated into a single OO development methodology.

In this work, the term *Integration* is used in yet another way where it indicates quality aspects. A desirable quality we would like to see in a development model across all its views is *consistency*. On closer look, this form of *integration* is not that different from the other meanings described above. For instance, when we perform component integration where we evaluate the integrity of components while assembling them into bigger components (or even systems) that integration is quite analogous to performing view integration where we evaluate the integrity of views while assembling them into bigger

models. The first case describes product integration, the second view integration. Both are facets of *Integration* (see [Grady 1994] for an overview of these facets).

3.4 The View Integration Problem

3.4.1 Why Integrate Views?

Previously, we gave an overview of an object-oriented development language (UML) and indicated that it (like others) addresses stakeholder concerns to satisfy their needs. We also briefly described a process through which we can guide and advise stakeholders on how to use those models and views in creating a software product in a consistent manner. We also described the deficiency of model-based development when it comes to solving problems (to model, to solve, and to interpret).

This deficiency of views would not exist if we could have a few *perfect views* that could be used by all stakeholders (as described above) and which would be precise enough, orthogonal, but still easy to use. These views, unfortunately, do not exist. Instead, we are confronted with a large number of loosely coupled, sometimes quite independent views that, to make things worse, exhibit redundancies. This is not really what we want. [Nuseibeh 1996] wrote that “multiple views often lead to inconsistencies between these views—particularly if these views represent, say, different stakeholder perspectives or alternative design solutions.”

Thus, if we have to deal with multiple views we would like to have at least tightly coupled ones. Since views represent only individual aspects of a system model, those views are meant to be together; only together can they fully describe the model of a system. However, we also need views to be different and independent enough to provide separation of concerns for stakeholders. What we need are thus views that are independent and can stand on their own but their contents (information) being fully integrated with the contents of the other views to ensure their conceptual integrity. We need *View Integration*.

We also need integration because views often use different underlying paradigms and, thus, the results of modeling a problem in one type of view may be different than modeling the same problem in another type of view. For instance, a non object-oriented analysis and design stage would yield functional decomposition (which is more suitable to be implemented in a functional programming language). On the

other hand, using an object-oriented design technique would already structure the system in a more object-centered fashion and thus, its implementation would be more straightforward to implement in an OO language. In Figure 4, we showed object-oriented views (classes, interactions) and functional views (data flows, state transitions). In UML, like in other models, those types of views are commonly used together. Thus if two different people would start modeling the same system, one using OO techniques and the other using functional ones, they might end up with two different solutions. Even if both solutions would correctly solve the problem, they still may not make much sense together. A reason for this is that functional views are structured differently than object oriented ones.

Furthermore, if modeling is done separately (one view at a time) we may get inconsistencies between them. Notations, like UML, describe some of the semantics of its views and how they are supposed to be used. Nevertheless, those semantic descriptions are rather limited and still enable inconsistencies. Lifecycle processes are sometimes used to mitigate that problem; however, those are usually not detailed enough and not enforceable enough to solve the view integration problem.

What we need is a development model that defines views and their relationships not only syntactically but also semantically (inter-view dependencies). Such a model would also need tool support to enable automation. The tool support should validate both syntactic correctness and semantic integrity. Tools today (e.g., Rational Rose) are generally good at enforcing intra-view syntax and semantics, but are rarely able to handle inter-view dependencies. We see view integration as being about adding semantics to our models so that the integrity of the whole is improved.

3.4.2 Why Integrate Heterogeneous Views?

The reason we chose the integration of heterogeneous views is because we needed to cover a broad set of development concerns. We needed multiple, distinct, and independent views that are clearly defined and generally understandable; we thus decided to focus on architecture and design views. Architecting and design are performed early in the development process from a purely engineering point of view. Both also occur early in the development life cycle which means that problems and faults are still relatively easy (and inexpensive) to fix if identified. Should architectural and design errors be carried into the implementation phase or even further, the cost of fixing them would become some orders of

magnitude higher [Boehm 1981]. Siegfried wrote that “there is no replacement for making a sound systems architecture early in a project” [Siegfried 1996].

Architectural views should also be supported as much as possible by (UML) design views to validate their refinements. But why stop there? Modeling should also cover requirements and other types views outside the product-model domain. With heterogeneous integration we mean the integration of a well-rounded, sufficiently complete set of views to address a large set of concerns [Medvidovic et al. 2001].

3.4.3 Why Automate Heterogeneous View Integration?

We also need to automate view integration mainly because of the complexities involved in bridging the information gap manually. When we talk about automated view integration, we also talk about automated synthesis and automated analysis. What automation provides is a reduction of manual, error-prone, and repetitive activities in dealing with view redundancies. Automation also implies tool support for using models, for validating their integrity, and for addressing some other important facets of view integration (e.g., scalability).

3.5 Bridging the Information Gap: Synthesis and Analysis

We discussed previously that views do not easily share information which denotes an information gap. That information gap cannot be eliminated but its negative side effects can be eased. There are two general methods for eliminating or minimizing it in an automated fashion.

1. **Automated Synthesis:** Generating information from previous activities so that they can be used at later stages (e.g., generate/synthesize code from design)
2. **Automated Analysis:** Verification and validation of model information so that inconsistencies can be identified (and potentially resolved) automatically.

Synthesis is thus about view transformation and analysis is about view validation (e.g., consistency checking). The most effective form of bridging the information gap automatically is synthesis since it implicitly incorporates automated analysis. The rationale is that faithful and reliable synthesis between views requires the understanding of inter-view interrelationships. Automated synthesis, if it is

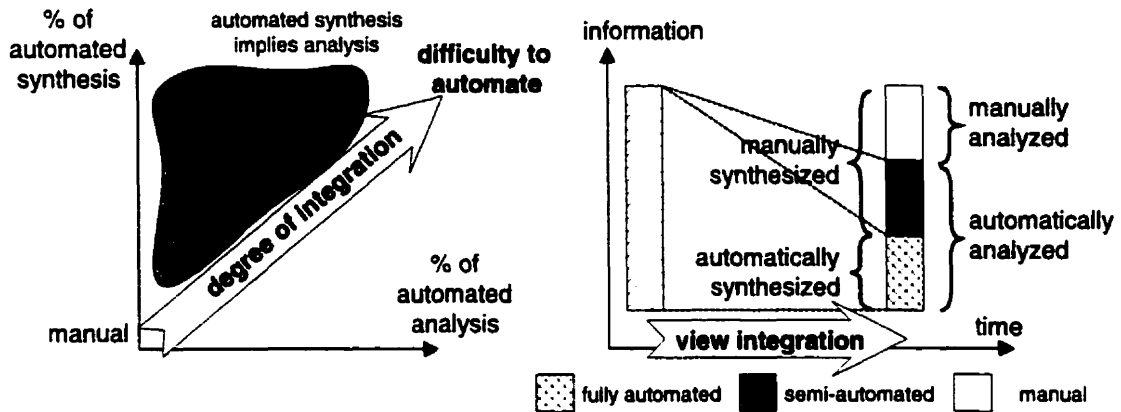


Figure 9. Integration to Enable Automated Synthesis and Analysis

done correctly, implies that the synthesized information is consistent with the source(s) (at least initially because evolutionary changes may cause inconsistencies later on). However, since synthesis incorporates analysis, synthesis is also harder to automate. Figure 9 depicts this relationship. The left side shows that the degree of (view) integration increases as automation increases. Since automated synthesis always implies automated analysis, it is not possible to increase synthesis without increasing analysis. In terms of automation, we find three degrees of integration of importance: full integration enabling automated synthesis and analysis; semi-integration enabling automated analysis only; and no integration where both synthesis and analysis have to be done manually.

The ideal form of integration is full integration, which implies full automation. If full integration is not possible, semi-automated integration at least supports some degree of automation in validating whether or not reentered information remains consistent with previously entered information (consistency checking). If continuity of data cannot be enabled through any automated means (either synthesis or analysis) then there remains no option but to do both activities in a manual fashion.

Although, we do not expect to be able to achieve full automation for both synthesis and analysis for software in general, we do believe that significant portions can be automated through a rigorous treatment of the subject (e.g., automated synthesis is achievable in very restricted domains). This work can be applied onto project-specific information as well as domain-specific information (e.g., product lines). In both cases, we need to enable information continuity [Egyed et al. 2000]. Although details of

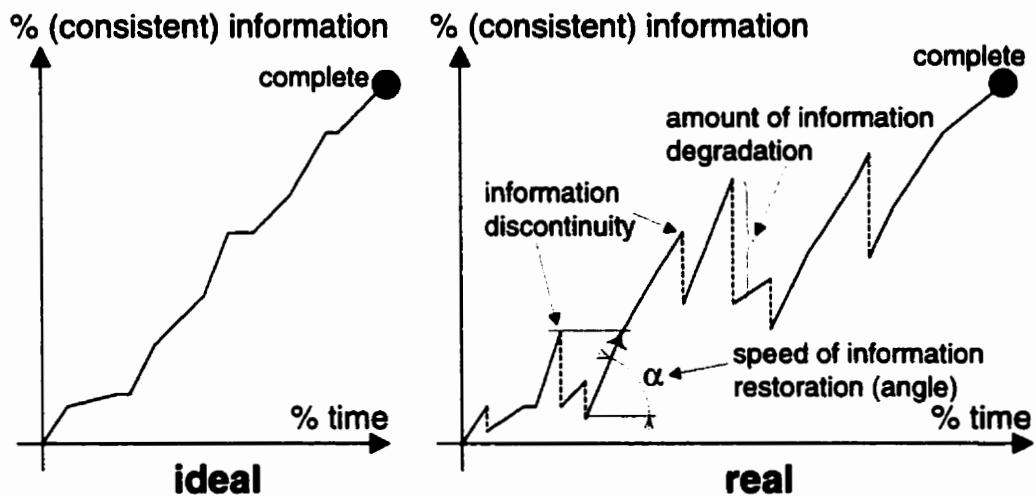


Figure 10. Information Discontinuity, Degradation, and Restoration

information continuity may vary between/within projects, the underlying concepts are alike. The goals are to decrease information degradation and increase the speed of information restoration. The left side of Figure 10 shows the ideal case of no information loss over time; the right side shows disruptions that are caused by information discontinuities and thus result in some information loss. As it can be seen in Figure 10 (right), minimizing information degradation causes a reduction of the extent of the information loss (the vertical gap is minimized). Furthermore, maximizing information restoration enables a speedy recuperation of information loss in case of information degradation. The latter case is important since it improves the ability to recover faster from an information loss.

Automated synthesis minimizes information degradation by enabling some information to be carried over automatically. Similarly, automated analysis minimizes the amount of rework required in regaining the previous level of information capture after an information degradation and therefore increases information restoration (e.g., by locating inconsistencies). In Figure 10, the positive effects of view synthesis can be seen as a reduced vertical drop after an information discontinuity. Similarly, the figure depicts the positive effects of view analysis through the angle α which indicates an increase in the speed of information restoration the steeper the angle is. Automated synthesis and analysis complement each other in the quest for information loss prevention.

3.6 Potential Integration Complexity

Having established what it means to integrate views and what the goals of view integration are (synthesis and analysis) we have to also discuss the complexities involved in this task (see Figure 11). In order to exchange and validate information between all views, each view needs to be integrated with **all** other views (assuming they all share information). In our example we have six views which, in the absence of a common reference model, would require them to be integrated in 15 different ways. Each additional view would force $(n-1)$ additional ways of integration if “ n ” is the number of all views to be integrated. In total, $n(n-1) / 2$ ways of integration are required for “ n ” views to be fully integrated. Clearly, we are confronted with a non-linear explosion of integration work ($O(n^2)$ for a big n). We will show techniques on how to address this problem later.

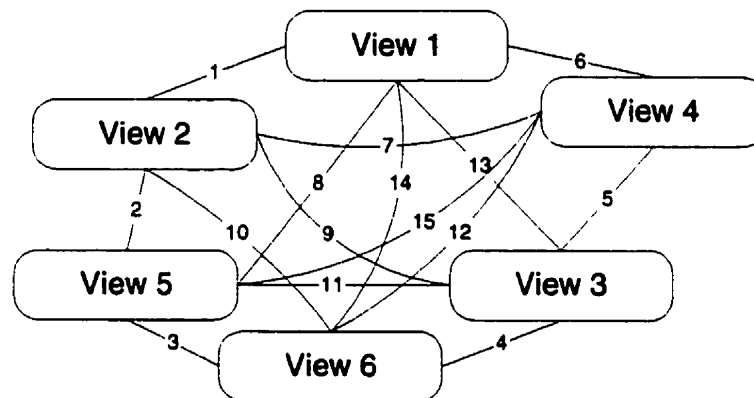


Figure 11: Complexity in Integrating Views

3.7 What is not Integration?

A usually simplistic notion of view integration is just providing a common meta model. UML is a good example since its standard provides an extensive meta-model defined by OMG (Object Management Group [OMG 1999]). A severe shortfall of UML (and its simplistic notion) is that a common meta-model only provides view representation and that it comes up short in fully integrating views with each other: Although UML and its meta-model define notational and semantic aspects of individual views in detail, inter-view relationships are not captured in sufficient detail. Without these

additional relationships, the (UML) model is nothing more than a collection of loosely coupled (or completely unrelated) views. Figure 12 illustrates this by showing UML views as separate entities within a common environment (the UML meta-model). Although, some views are weakly integrated (e.g., class and sequence diagrams), in general, UML views are independent.

Figure 12 also shows that the lack of view integration can extend beyond existing UML views to non-UML views represented in UML (e.g., ADLs). UML supports its own extension via mechanisms like stereotypes, tagged-values, etc. Using these mechanisms enables us to represent new concepts in UML. For instance, we could choose to incorporate Entity-Relationship models [Chen 1976] expressed via stereotyped and constrained class diagrams. We also discussed that some architectural description languages (e.g., C2) have already been integrated into the UML framework. The major drawback of using UML's extensibility mechanism to capture additional views is that those views are presented in UML but not fully integrated. The problem is similar to the above where view representation only describes how information can be captured in UML. Thus, view representation does not concern itself with whether information captured through new views is consistent with other parts of the existing model. View representation would allow the creation of multiple views, each of which would correctly conform to their specifications; however, their combination would not build a coherent unit. We therefore speak of

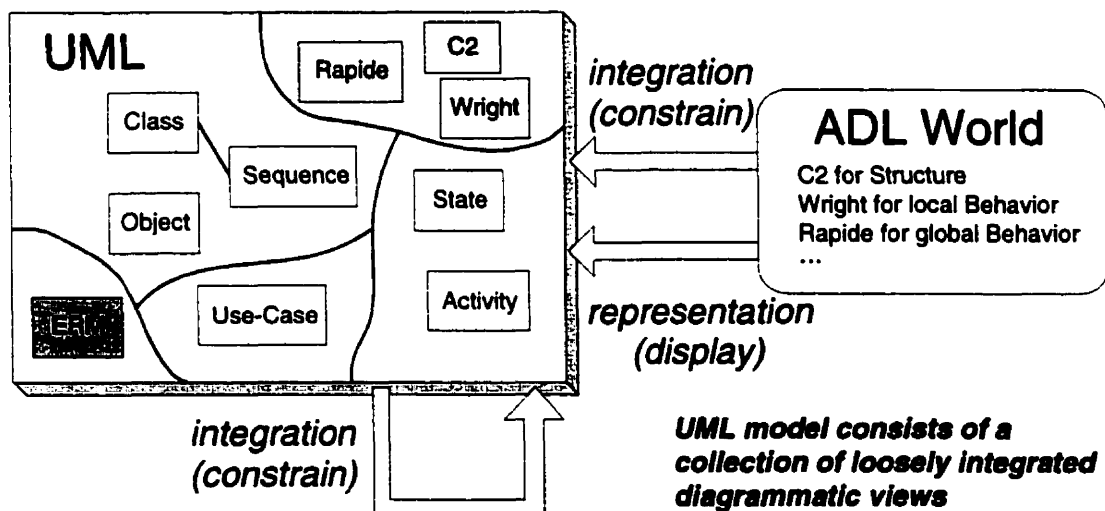


Figure 12: Views and ADLs represented in UML

view integration as an extension to view representation to ensure the conceptual integrity (consistency and completeness) of the entire model *across* the boundaries of individual views.

3.8 Summary

This section discussed the meaning of view integration. We emphasized that view integration extends most of current development models by specifying the semantics of views and their interdependencies. Based on those semantics, the relationships between multiple views can be qualified meaningfully and automated techniques can be used to validate their integrity. Besides settling some terminology issues, this section also talked about the goals of our work and the complexities involved. This section also briefly mentioned a common misconception of what is not integration (e.g., common meta model).

4 Scope and Limitations

The primary objective of this work is finding a framework that supports view integration in the context of transformation and consistency checking between heterogeneous types of views. The emphasis of this work is towards describing and localizing inconsistencies between multiple views. Consistency checking, which is part of analysis, addresses this facet of view integration. In the course of investigating the view integration problem, we found that effective consistency checking mandates transformation (synthesis) as well. This work will therefore also discuss view synthesis and how it assists analysis. To that end, this work will present a framework for transformation and consistency checking that together solve the larger problem of view integration. We have also investigated view synthesis and analysis issues outside the area of this work and therefore were able to gather extensive insights into the requirements of view connectors (e.g., [Medvidovic et al. 2001] and [Gruenbacher et al. 2000]).

Since view integration has a non-linear complexity (recall Section 3.6), this work focuses on a subset of the problem. It is out of the scope to address the entire palette of design (e.g., UML) views. Thus our work does not have the aim to be as complete and consistent as possible but instead to be as complete and consistent as is feasible. [Nuseibeh 1996] shares this view when he talks about viewpoint integration. This implies that we cannot guarantee that our integration approach will be able to cope with all situations. Furthermore, our techniques may not work under all circumstances; neither will they uncover all inconsistencies that could occur. Inconsistencies detected via our techniques must also be regarded as *potential* inconsistencies but not as factual ones. The reason is that often not all development aspects are captured explicitly by the users. Our technique, as other integration approaches, therefore have to rely on assumptions and heuristics.

To be able to cope with a wide range of views and their implications, we decided to follow a breadth and depth approach. UML supports nine types of views and we have decided to focus on class, object, sequence, and statechart diagrams initially. Section 5 will explain that those four types of views cover the most significant dimensions of the view integration problem, giving our framework sufficient breadth without having to consider all of UML. Adding other types of views does not invalidate our

framework. For instance, adding collaboration diagrams or activity diagrams to our framework is well within the boundary of our framework since both can be categorized into our view dimensions. We also decided to have a depth approach to view integration to investigate the problem in all necessary details. We therefore, decided to only take a subset of the above views (object and class diagrams) and elaborate on them to ensure that our framework also covers depth (something the breadth approach might not have revealed). For instance, we provide tool support of our approach but limit tool support to object and class diagrams only.

The emphasis of our work is generally on the technology side of the view integration problem (as opposite to the human-computer interaction side). As such, we will discuss view transformation, view comparison, and repository issues in detail. However, we will not discuss ergonomics-specific user interaction issues unless they are important for the technology side (e.g., we will not discuss how to present inconsistencies to users or what to do in order to resolve them). Our emphasis on technology is necessary since one the biggest unsolved challenges of view integration is scalability. We already indicated one scalability aspect in Section 3.6, but there are other even more severe challenges that need to be addressed to enable automated consistency checking. In terms of scalability our approach has several new and unique solutions.

Another focus of our work is on product models. In Section 2 we listed and briefly discussed design and architecture models that are within the umbrella of product models. Product models describe the to-be-built software system, but not its environment. Our work does not investigate the relevance of our approach towards other types of models such as process models or property models (see Section 2.6). Another reason why we emphasize UML in this work is because UML has become a de facto standard for object oriented software development. Nevertheless, UML's definition is often ambiguous and many interrelationships between views are undefined. Since it is not within the scope of this work to formalize UML we will use others' interpretations and formalizations whenever necessary and applicable (e.g., [Cheng et al. 1995] or [Övergaard 1998]). We consider it within the scope of this work to investigate UML's ability to deal with view integration issues and to evaluate its deficiencies in that context.

A final restriction of this work is the issue of resolving inconsistencies. We will primarily investigate the problem of how to identify and locate inconsistencies and we will not investigate how to automatically resolve them. The reason for that is that inconsistency resolution requires extensive human-computer interactions and those are well outside the scope of this work (see above).

The following will summarize the scope of this thesis:

1. Describe a view integration framework as a foundation to support the definition and identification of (potential) inconsistencies.
2. Find techniques to support view integration activities defined in the above framework. For this, we will partially rely on existing technology and will introduce some of our own.
3. Combine view integration techniques to make them work together. This step is primarily necessary because we use some techniques from other researchers and those techniques need some adaptation us to be useful for our purposes (e.g., [Koskimies et al. 1998]).
4. Show how view integration techniques can be applied to simplify consistency checking based on consistency rules (constraints) and how scalability issues can be addressed.
5. Summarize and list potential inconsistencies that can occur between heterogeneous types of views.
6. Build tool support for all integration activities in the context of our in-depth approach covering class and object diagrams.
7. Compare and assess our work towards other view integration approaches.

5 Model Elements and Views

So far we talked about the view integration problem at a very high level. In order to explore its details more rigorously, we need to first settle some terminology issues regarding the meaning of model elements, views, and models. This section will also categorize views that will serve as a foundation for our view integration framework later on.

5.1 What are Model Elements, Views, and Models?

Model Elements are the core elements that comprise views and models. For instance, a class diagram primarily comprises classes and relationships. Upon closer inspection, we also find operations, methods, attributes, parameters, constraints, and other model elements. The Unified Modeling Language (UML) defines a very rich set of modeling elements, a subset of which is depicted in Figure 13. One of the interesting features of UML is that its meta model was described using a subset of itself. Figure 13 shows the subset that depicts both the UML meta model and UML's meta-meta model. The most basic element of UML is an *Element*. A *Model Element* is derived from an element and all other UML elements are children of *model element*.

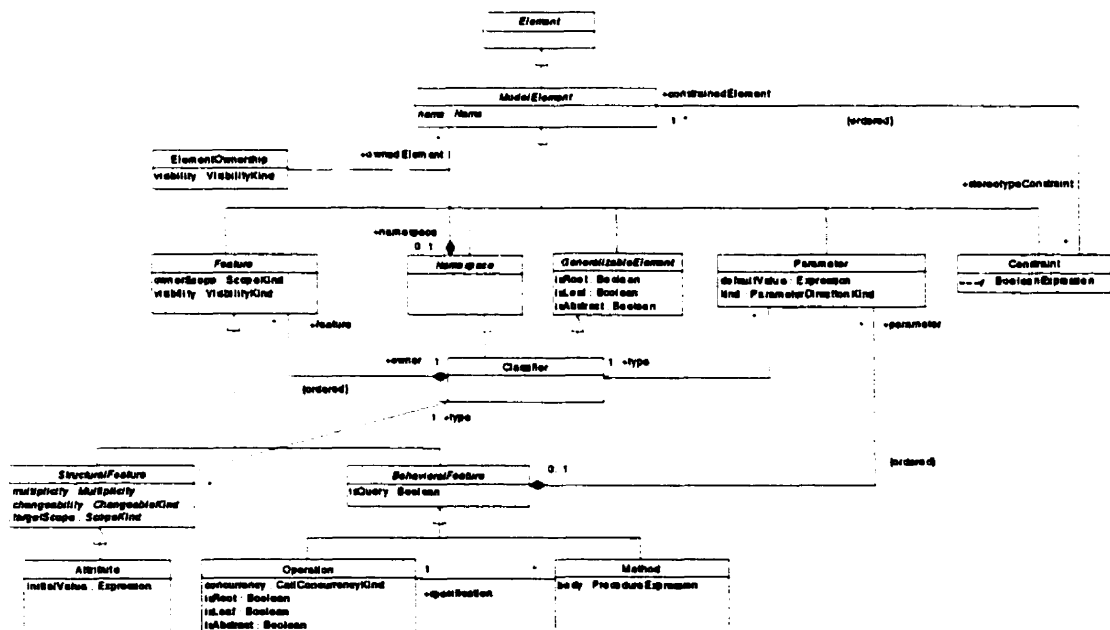


Figure 13. UML Core Elements as defined in [OMG 1999]

Elements are the foundation of a model and can be atomic or composite. UML defines a series of atomic elements and then builds on them to describe more complex types of elements. Composite model elements comprise of atomic ones and are more abstract. The elements depicted in Figure 13 are both atomic and composite. To instantiate a particular application (e.g., software system), one has to instantiate the meta model. In our work, we use instantiations of the meta model to demonstrate inconsistency examples of UML. For instance, Section 6 shows specific examples of inconsistencies.

Views are collections of model elements. As it was discussed in Section 2, views address stakeholder concerns and thus incorporate model elements that are needed to describe the problems and solutions of those concerns. Nuseibeh describes views “to be loosely coupled, locally managed, distributable objects. Each [view] encapsulates partial knowledge about a system and its domain—expressed in a suitable representation scheme—together with partial knowledge about the overall process of development” [Nuseibeh 1994].

The term *Model* is ambiguous. Software practitioners use it to describe any form of abstraction through which the real world could be approximated and analyzed. Sometimes, the term model is used even more generally to describe development concepts and philosophies. The term *Methodology* is sometimes used analogously. In this work, we normally refer to a model as the collection of modeling information related to a particular software project. We define a view as providing a specific context for using models and model elements to describe and analyze stakeholder concerns. Since views are independent from one another, we see the model, which captures all view-related information, as providing a framework for view integration.

View integration has to happen on a meta-model level so that its instances (our product model) “inherit” the view integration features we wish to ensure. For the most part, we will not work with UML’s meta-meta model, although there are exceptions (e.g., Sections 7.7 and 11.1.6) because one of our findings is that UML’s meta model inadequately handles some of the view integration details needed to support scalability. Our findings thus also result in some recommendations on how to improve UML on both the meta level as well as the meta-meta level.

5.2 Model Elements, Model Instances, and User Objects

“An instance [of a model] is a run-time entity with identity, this is, something that can be distinguished from other run-time entities. It has a value at any moment in time. Over time the value can change in response to operations on it” [Rumbaugh et al. 1999]. This definition applies to instances of elements. We also refer to those instances as user objects. For example, if a class defines a “Person” with properties like name and age then instances of that class could be “person1” of type *Person* with values “Alice” and “23” or “person2” of type *Person* with values “Peter” and “15.” In this example, *Alice* and *Peter* are user objects whose values can change over time (e.g., age could be updated).

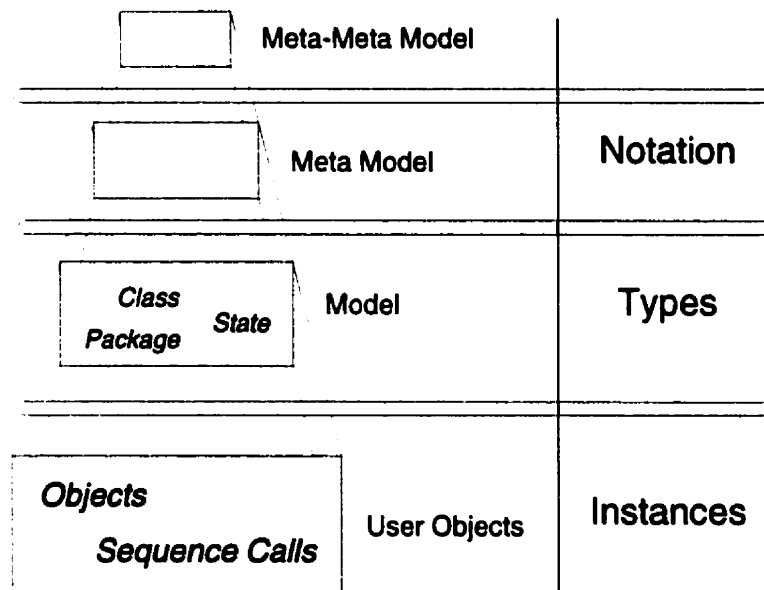


Figure 14. The Four-Layer Meta-Modeling Architecture of UML [Medvidovic et al. 1999b]

The distinction between meta models, models, and user objects is, however, more subtle. Figure 14 depicts the four-layer meta-modeling architecture of UML as defined in [Medvidovic et al. 1999b]. The meta-meta model layer defines a language for specifying UML (the meta-meta model is depicted in Figure 13). The *meta model* layer, in turn, defines legal specifications for a given modeling language. For example, the UML meta model defines the legal notation and semantics of UML specifications. The *model* layer is then used to represent specific software systems (e.g., like the *Person* class above). The model, an instance of the meta model, describes a software product. Finally, the *user objects* layer

corresponds to specific instances of a given model (e.g., like the *Alice* and *Peter* objects above). User objects are the instances of model elements and capture run-time aspects of the product model.

5.3 UML Model, Model Elements, and Views

[Rumbaugh et al. 1999] defines UML as having structure and dynamics (see also Table 3). Structural views describe software system entities (components) and their relationships (interconnections). In UML, static views, use case views, implementation views, and deployment views support structure. Behavioral views describe interactions between software system entities (component) and/or relationships (interconnections). In UML, state machine views, activity views, and interaction views are behavioral. The main model elements of UML are listed on the far right column of Table 3. Views are constructed out of those elements. For instance, class diagrams can be constructed out of

Table 3. UML Views and Diagrams adapted from [Rumbaugh et al. 1999]

Major Area	View	Diagrams	Model Elements
Structural	Static view	Class diagram	Package, subsystem, class, association, generalization, dependency, realization, interface
		Object diagram	
		Package diagram	
	Use case view	Use case diagram	Use case, actor, association, extend, include, use case generalization
	Implementation view	Component diagram	Component, interface, dependency, realization
	Deployment view	Deployment diagram	Node, component, dependency, location
Dynamic	State machine view	Statechart diagram	State, event, transition, action
	Activity view	Activity diagram	State, activity, completion transition, fork, join
	Interaction view	Sequence diagram	Interaction, object, message, activation
		Collaboration diagram	Collaboration, interaction, collaboration role, message
Extensibility	all	all	Constraint, stereotype, tagged values

classes, associations, generalization, dependencies, and so forth. This list of model elements is not complete; instead, Rumbaugh, Booch, and Jacobson only listed the more significant ones. In this work we will make use of a subset of UML covering those model elements that are shaded in gray. We, therefore, cover both structural and dynamical aspects UML has to offer. In the next section, we will also show that our selection also covers the most relevant other view dimensions.

For a detailed description of the UML notation and semantics please refer to [Rumbaugh et al. 1999]. In [Medvidovic and Rosenblum 1999] and [Robbins et al. 1998] it can be further seen how the UML extensibility mechanism can be applied to model other model representations (e.g., ADLs). In Section 7.7 we will further show how this mechanism can be used to model view integration elements not currently present in UML.

5.4 View Dimensions

We extended the view classification of Rumbaugh-Booch-Jacobson [Rumbaugh et al. 1999] shown in Table 3 to cover broader aspects of view integration (see Figure 15). Since views can be very distinct in what they are meant to convey, we have tried to capture their most significant commonalities. We found that views can be classified via three dimensions: their level of generality, their level of abstraction, and their level of behaviorism. The following sections describe what those levels convey.

5.4.1 Level of Generality

The level of generality of views indicates how universally true information communicated in these views is. For instance, a UML class view depicts a generic representation of modeling information since it describes invariant facts about software components that must always hold (e.g., humans are mammals). A counterexample would be an object view that describes less generic information (e.g., Alice is a human and Alice is also a mammal). Based on more specific information, it is not very intuitive to infer general information. For instance, we cannot generalize that humans are mammals based on the observation that Alice (an instance of human) is a mammal. An object diagram therefore does not communicate the same level of generality as a class diagram. In the three-dimensional projection of Figure 15 a class view would be depicted apart from an object view along the generality dimension.

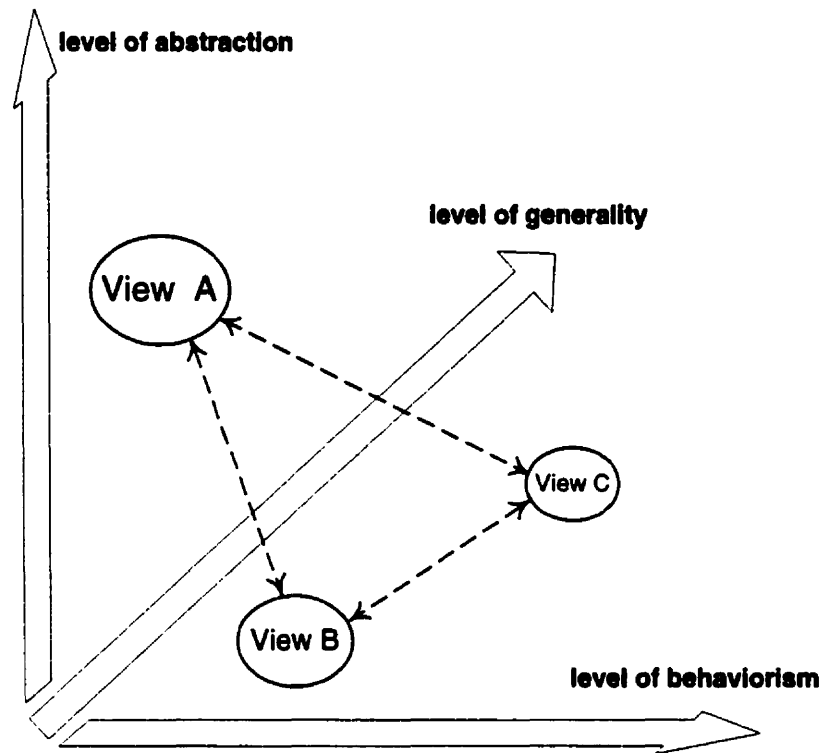


Figure 15. Views Dimensions

5.4.2 Level of Abstraction

The level of abstraction of views indicates how decomposed modeling information depicted in views is (level of granularity). For instance, a UML package view depicts an abstract representation of modeling information since it engulfs complex and detailed element (e.g., the package *animal* engulfs a wide range of species such as *monkeys*, *horses*, and *cats*). A counterexample would be a class view that describes less generic information (e.g., *monkey*, *horse*, *cat*). In the context of less abstract information, it becomes more difficult to comprehend the “bigger picture.” For instance, it is much harder to memorize that horses, cats, and monkeys need air than it is to memorize that animals need air. Although, the class view communicates the same information as the abstract package view, the level and amount of detail there obscures abstract information. In the three-dimensional projection of Figure 15, a class view would be depicted apart from a package view along the abstraction dimension.

The abstraction dimension frequently reflects a system's decomposition. Usually, higher levels (layers) show the system in a more abstract fashion, whereas lower levels show the system in more detail. Each layer should represent the *complete* system. Abstraction also enables model partitioning into subsystems.

Note that abstraction and generality seem closely related. There is, however, a strong distinction: whereas a less general view only needs to depict a part of the generic picture, a less abstract view still needs to depict the entire abstract picture. The less abstract view is also allowed to provide additional details, however, the less generic view usually only "instantiates" specific examples. Also, generic views tend to use model elements; less generic views tend to use user objects. Abstract views and less abstract views, on the other hand, usually stay within one such domain (both are either model elements or user objects).

5.4.3 Level of Behaviorism

The level of behaviorism of views indicates how much interactive information is communicated via these views. For instance, a UML statechart view depicts the behavior and interactions of modeling elements at any given time (e.g., elevator door must be closed before the cabin can go up or down). A counterexample would be a class view that describes interactions in general (e.g., cabin depends on door). Based on less behavioral information, it is hard, if not impossible, to infer interaction. For instance, based on the information that "Cabin depends on Door," we have no way of knowing how that dependency is actually realized. The class diagram, therefore, does not communicate the same level of behaviorism as the statechart diagram. In the three-dimensional projection of Figure 15, a class view would be depicted apart from a state view along the behaviorism dimension.

5.5 View Space and its Relation to Views

The three dimensions of *abstraction*, *generality*, and *behaviorism* form a three-dimensional space—a view space—into which (UML) views can be placed. In the context of this space, views can trade-off abstraction, generality, and behaviorism relative to one another. For instance, there could be one type of view that is less generic but more behavioral than another view. Figure 16 depicts the three-

dimensional view space with class, sequence, object, statechart, and C2SADEL views placed in it. For instance, in the generic dimension, the class view is more generic than the sequence view, whereas, in the behaviorism dimension, the class view is less behavioral. Class views are also more abstract than sequence views. Figure 16 also depicts the partitioning of dimensions into ranges. Abstraction can be divided into abstract and concrete, generality can be divided into generic and specific, and behaviorism can be divided into structural and behavioral.

The positioning of views into the view space is not absolute. For instance, class views can depict software systems through various levels of abstraction. The white arrow on the abstraction plane in Figure 16 depicts that range of freedom. Similarly, a class view can depict software systems through various levels of generality (e.g., hiding some interdependencies in views). The white arrow on the

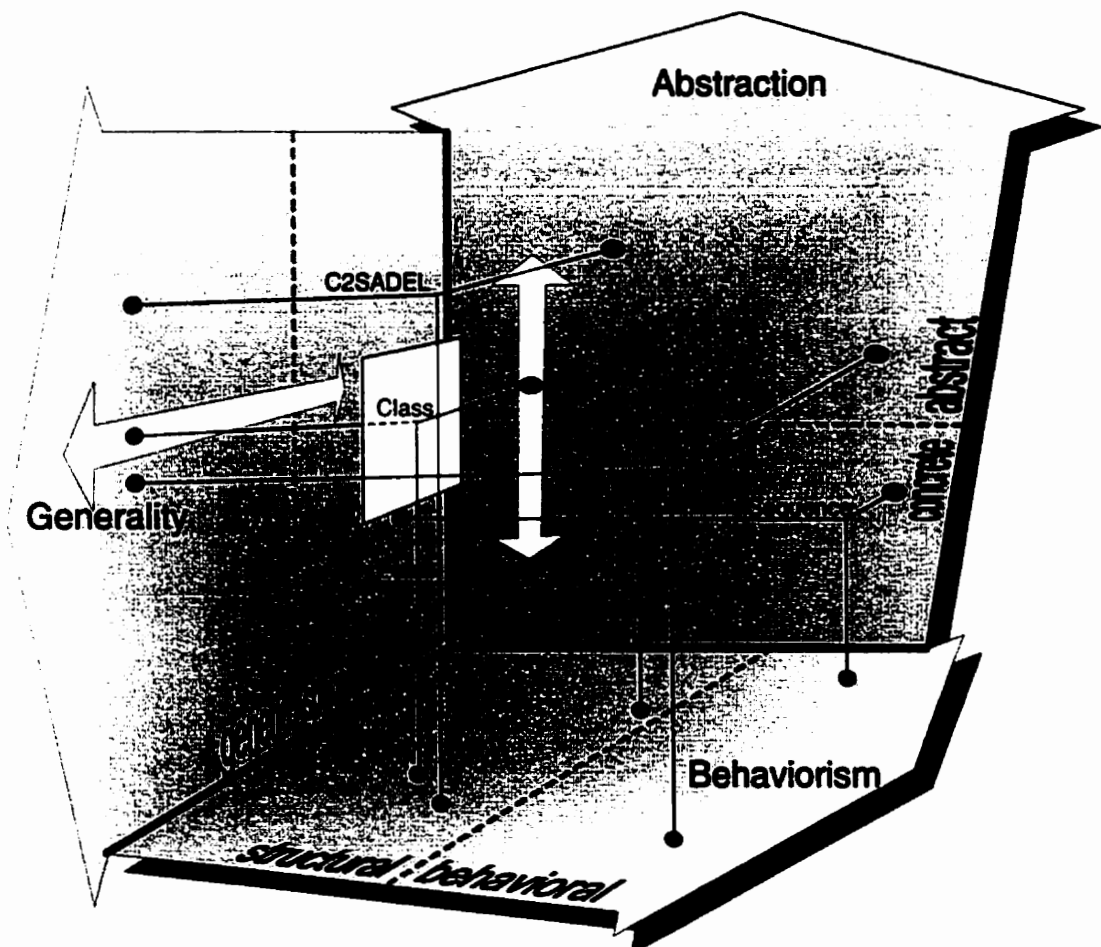


Figure 16. Views and the View Space

generality plane indicates that freedom. The levels of freedom in how views are assigned to dimensions are, however, limited. For instance, a class view depicts a very general form of structure. The amount of freedom of modeling behavioral system information via class diagrams is therefore very limited. Figure 16 indicates the freedom of class views in the form of a rectangle. The rectangle is actually a prism since a class views does have some behavioral freedom to describe various levels of behaviorism (e.g., different relationship types such as dependencies or associations denote different behaviors).

The view space in Figure 16 also depicts the existence of the information gap visually (recall Sections 2.7 and 3.5). This gap denotes the view integration dilemma where it is not obvious how one may bridge the space between views. Also being able to group views into the view space allows us to reason (or at least speculate) about some interesting issues concerning views:

- 1) What is the optimal number of views? (e.g., views covering all view dimensions)
- 2) What views should be used together? (e.g., coverage of the view space)
- 3) What is the perfect view? (e.g., a view that spans all view dimensions)

The separation of views into view dimensions allows us to split them into upper and lower ranges (e.g., abstract or concrete). We find eight regions in the view space. Table 4 lists those eight regions and also indicates which view(s) we currently support in covering those regions. We mentioned

Table 4. Eight Regions of the View Space

Abstraction	Generality	Dynamism	Candidate Views
Abstract	Generic	Behavioral	Statechart views
Abstract	Generic	Structural	Class views, C2SADEL views
Abstract	Specific	Behavioral	
Abstract	Specific	Structural	Object views
Concrete	Generic	Behavioral	Statechart views
Concrete	Generic	Structural	Class views
Concrete	Specific	Behavioral	Sequence views
Concrete	Specific	Structural	Object views

in Section 4 that we scoped the limits of our view integration framework to four types: class, sequence, statechart, and object views. Table 4 shows how those four types cover seven out of eight view regions. The eighth view region (abstract, specific, and behavioral) is not covered by any current UML view and is therefore left empty.

Note that Table 4 is not meant to categorize views. Instead it is meant to categorize software-specific information within views. For instance, class diagrams are both useful as abstract and concrete views. Nevertheless, the information modeled in abstract class diagrams is different from the ones modeled in a concrete class diagram due to decomposition. Note that both Figure 16 and Table 4 depict the C2SADEL [Medvidovic et al. 1999a] me view which is not part of UML. We integrated our work with C2SADEL, an architecture description language (ADL), to demonstrate our framework's ability to handle other types of views outside the UML domain. A pre-requisite for doing this is the ability to integrate a new view into the view space. [Egyed and Medvidovic 2000] discusses that integration in more detail since it is outside the scope of this work.

5.6 Interdependencies of Model Elements

Views (class views, sequence views, etc.) are comprised of model elements. In a graphical representation, there are typically box and arrow types of model elements. For instance, a class view consists of class "boxes" and association, dependency, or generalization "arrows." Both, boxes and arrows, are considered model elements; however, these model elements capture only interdependencies within views (and not between them). There are, additionally, a variety of relationships among model elements that cannot be captured within single views. For instance, an object that is an "instance" of a class or a class that is "part-of" a package describe inter-view relationships. Some of those relationships between views are captured in UML implicitly and others explicitly. For instance, no explicit relationship type exists to associate a class to a package. A class is simply linked to a package (grouping effect), denoting an implicit relationship. The abstraction dependency relationship between classes is explicit in that the dependency relationship is declared as its own type within UML. Multi-view development covers typically two types of interdependencies between model elements:

- 1) Dependencies within model elements belonging to the same view (intra-view)
- 2) Dependencies between model elements belonging to different views (inter-view)

The former are part of regular views, such as class views, and the latter are usually not depicted in views (although several of them are defined in UML). Inter-view dependencies (as dependencies between views are often called) are also known as traces. The knowledge of how traces interrelate is known as traceability. “Traceability can be easily defined as forward and backward links between a system and its allocated requirements, and between those requirements and actual design elements” [Gieszl 1992]. In Sections 7.3 and 7.6.2, we will show why knowledge about traces is important for view integration.

5.7 Summary

This section refined our concepts of model elements, views, types, and instances. The contribution of this section towards view integration (and consistency checking) is the existence of a view space into which views (and model elements) can be categorized. Our view space is three-dimensional and denotes the level of generality, abstraction, and behaviorism a view can exhibit (we will show later how scalability and automation profit from that discovery). This section further discussed that there are two types of modeling information—information that describes a product and information that relates product aspects among various views. The latter is commonly referred to *traces*. It is those traces that describe the relative positioning of views in the view space (e.g., abstract traces describe that one view is more abstract than the other).

6 Model Inconsistencies

“Consistency checking between [views] is a vehicle for integrating these [views]. It is the activity in which two or more [views] compare knowledge and ascertain whether or not the relationships that supposedly hold between them do indeed hold” [Nuseibeh et al. 1994]. This implies that one important goal of view integration is to provide automatic assistance in identifying view inconsistencies. Although ensuring the conceptual integrity of models and views may not be fully automatable, there are various types of inconsistencies that can be identified and even resolved in an automated or semi-automated fashion. This section will show examples of inconsistencies in UML.

6.1 Examples of Inconsistencies

Having defined views in terms of their dimensions, we will now complement that by showing more concrete examples of (potential) modeling inconsistencies that can occur between and within those view dimensions.

6.1.1 Inconsistency between Class Layers

The first example shows a simplified air traffic control system (see Figure 17). The system is presented in two layers and, as it was discussed in Section 5.4.2, each layer must present the system in a complete fashion. The first layer shows the interaction of the *Flight* component that has some dependencies to *Mechanic*, *Pilot* and *Flight Controller*. The second layer refines this relationship by decomposing the *Flight* component into *Flight Plan*, *Aircraft*, and *Flight Authorization*—the *Flight Plan* being dependent on the *Pilot*, the *Aircraft* with its instance *Boeing 747* being dependent on the *Mechanic*, and *Flight Controller* being dependent on *Pilot*.

Although *Flight Controller* and *Flight* are present in the lower level diagram, it remains unclear whether their relationships are equivalent to those in the higher-level diagram. It would be dangerous (or, in this case, incorrect) to conclude that there is a dependency simply because there are lines going from *Flight*, via *Pilot*, to *Flight Controller*. Upon closer inspection, we find that both *Flight Plan* and *Flight Controller* depend on *Pilot*. Since *Flight Plan* is part of *Flight*, it follows that even *Flight Plan* depends on *Pilot*. However, the fact that both *Flight* and *Flight Controller* share the same dependency to *Pilot*

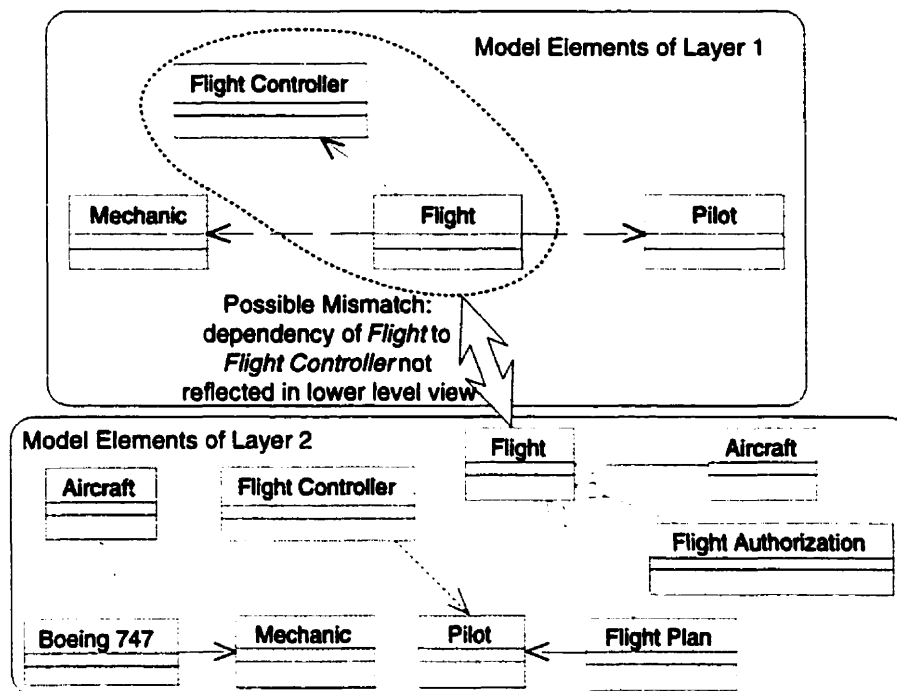


Figure 17: Potential Mismatch between two Layers (Completeness)

does not indicate that *Flight* depends on *Flight Controller* as the higher-level diagrams requires. Given the lack of additional information, we can conclude that there is an inconsistency between the abstraction (top diagram) and the refinement (bottom diagram).

6.1.2 Inconsistency between Class and Sequence Diagram

Figure 18 depicts another example of an inconsistency in a simple aerodynamic system. The figure shows a class and a sequence diagram. The class diagram shows that the *Car* consists of the parts *Tires* and *Engine*. The sequence diagram further indicates a scenario where the impact of speed is analyzed based on the shape of the car. The sequence diagram shows this for a particular instance of *My Car*. Based on the sequence diagram, we can observe that the aerodynamics class accessed operations of *Tires* and *Cars*. *Car*, in turn, accesses *Engine*. If we compare this data with the class view, we find an inconsistency. Since *Tires* is part of *Car*, only the *Car* object (*My Car*) should be able to call methods of *Tires* (*myTires*). The sequence diagram violates that rule. Possible ways of resolving that inconsistency are (1) to change the relationship between *Car* and *Tires* ; or (2) to change the direction of calls in the

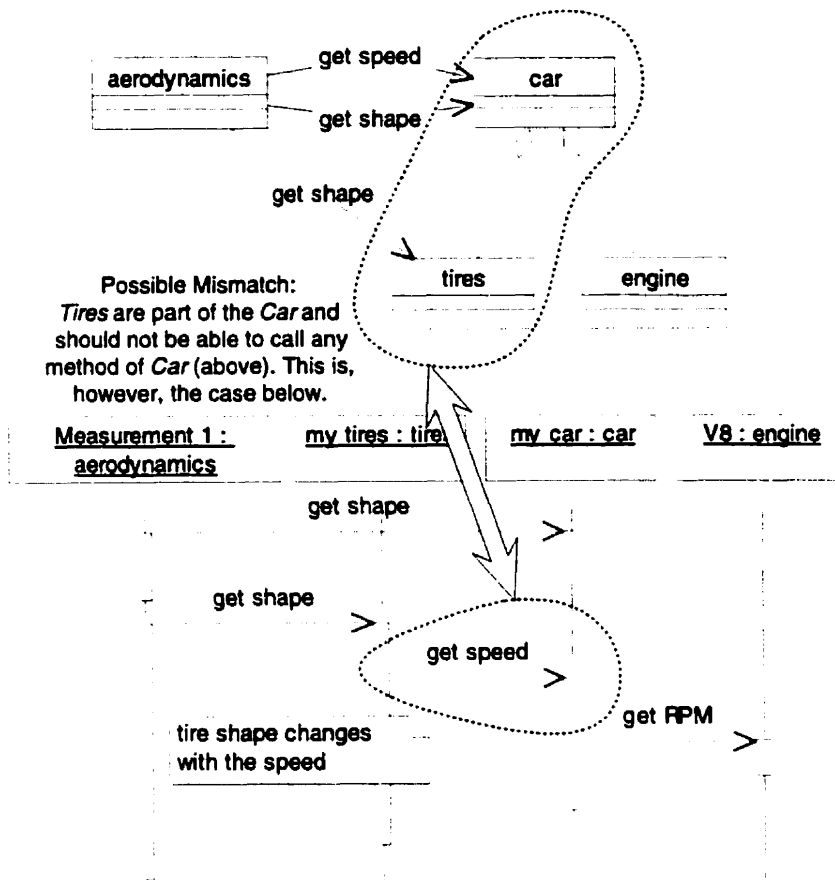


Figure 18: Potential Mismatch between Class Diagram and Sequence Diagram

sequence diagram (e.g., the latter could be done by having *Aerodynamics* pass along the speed of the car as a parameter).

6.1.3 Cardinality Inconsistency

Figure 19 shows an example of an inconsistency in a hospital system. The class diagram (top) shows the relationships between a *Patient* and his/her *Visiting Record* during a stay in a hospital. Even though a patient may have stayed in the same hospital more than once before, he/she should nevertheless have only one current visiting record at any given time. This static rule is violated in both the object diagram (lower left) and the sequence diagram (lower right). The object diagram shows an instance of Patient *John Smith* and it also shows that he has two current visiting records. Similarly, the sequence diagram shows that a new visiting record is created for John Smith even though one already exists. Note

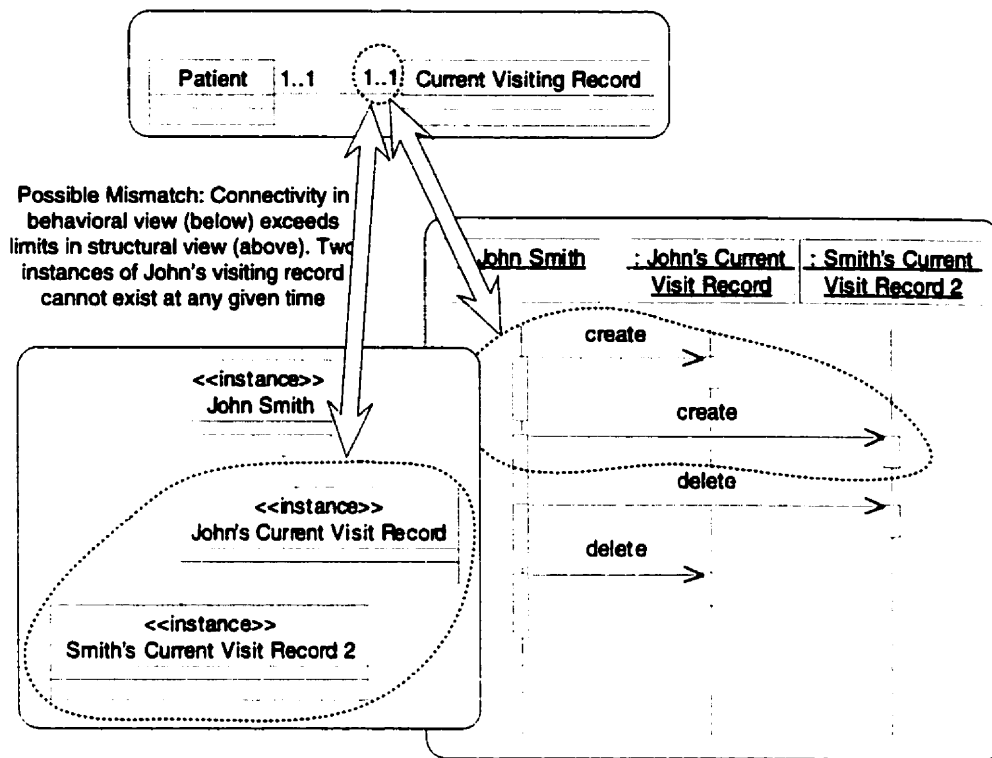


Figure 19: Potential Mismatch Between a Structural View and two Behavioral Views

that there is no inconsistency between the object and sequence diagram. In above example, inconsistencies can only exist between the class and object diagrams, and class and sequence diagrams since both depict scenarios and it is generally impossible to reason about the validity of scenarios in context of other scenarios.

6.1.4 Inconsistency between State and Sequence Diagrams

The next sample mismatch discussed here is depicted in Figure 20. It shows another perspective of the hospital system where we can see the system from a clerk's point of view. The clerk is using the screen to create visiting records for patients. The state diagram of the *Screen* class (top) shows that information about a patient is entered and validated and, after the patient database is checked, a visiting record is created. The inconsistency is in the treatment of *Patient*. The sequence diagram creates a new object of type *Patient* but the state diagram does not support that action. The sequence diagram (bottom right) shows that data is validated, patient information is retrieved and, since that given patient is not found, both patient and visiting record are created.

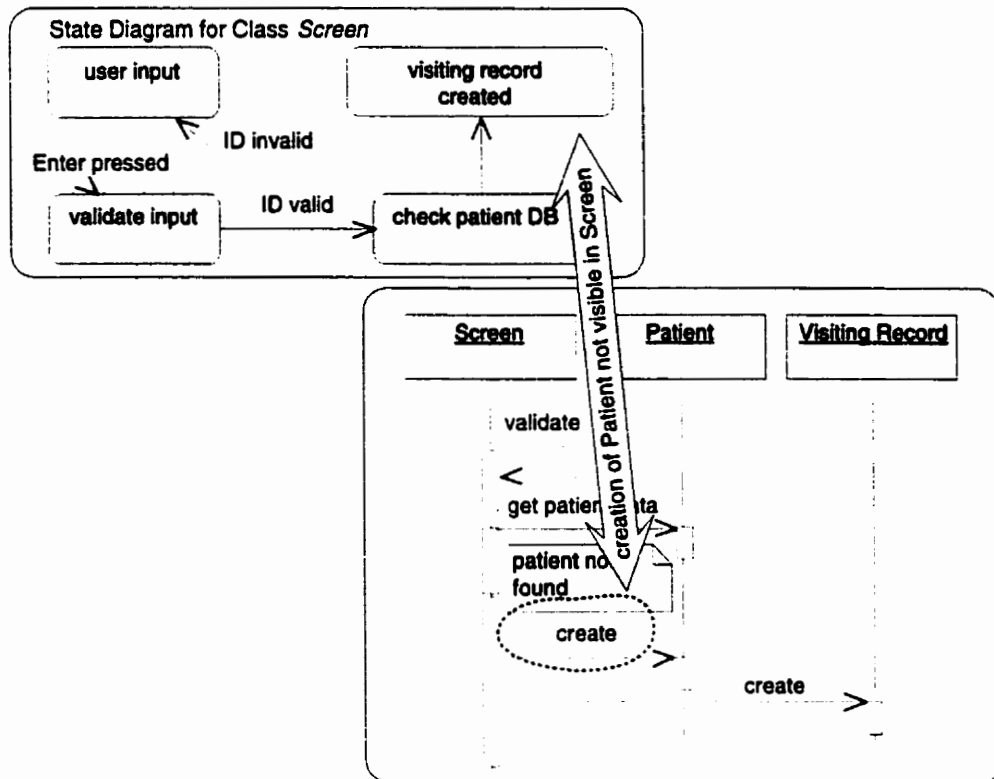


Figure 20: Potential Mismatch between State-, Sequence- and Collaboration Diagrams

Figure 20 adds a few extra challenges to our view integration problem since this example does not always use the same names for same/similar things. For instance, *get patient data* and *check_patient_DB* may seem identical to us (humans); however, for the computer this may not be the case. Thus, we are confronted with having to identify trace relationships (e.g., through a data dictionary). The other challenge we see in Figure 20 is even more interesting since it still remains hard to see what parts of the diagrams correspond to one another. For instance, to which model element in the sequence diagram does the state *Visiting Record Created* relate? If we relate it to the *create* arrow that calls *Visiting Record* from *Screen*, we would be incorrect. The *create* arrow just calls the method. No *Visiting Record* and no *Patient* are created at that point. So the state *Visiting Record Created* clearly does not correspond to that arrow nor the next one, but, instead, it corresponds to the point when the *create* method is finished and execution control is returned to *Screen*. So, the state *Visiting Record Created* corresponds to the void

between the two *create* calls. This implies that model elements may not always be traceable from one to the other although a dependency may exist. Whereas the naming problem in this example can be approximated with the use of a naming dictionary (which keeps track of all synonymous model element names), finding an adequate way of relating state diagrams with sequence diagrams is not as simple.

6.2 List of Inconsistencies

Although, a human analyst would be able to reason about the existence of these kinds of inconsistencies, for a computer to come to the same conclusion is not trivial. For simple examples like the ones above, the need for automated assistance in identifying and resolving inconsistencies may not be obvious; however, in more complex examples involving hundreds or thousands of modeling elements, the task of finding and resolving inconsistencies may become very time consuming and error prone—frequently having strong effects on project schedule and cost. Thus, automated assistance in identifying and resolving inconsistencies would result in major benefits.

6.2.1 Inconsistencies in the Abstract Dimension

1. Concrete relation has no corresponding abstraction

This case indicates that a lower-level model relationship between classifiers is not reflected between their higher-level counterparts. This inconsistency may indicate that the higher-level view does not capture the complete extent of the component interactions. Figure 21 depicts an example of one such

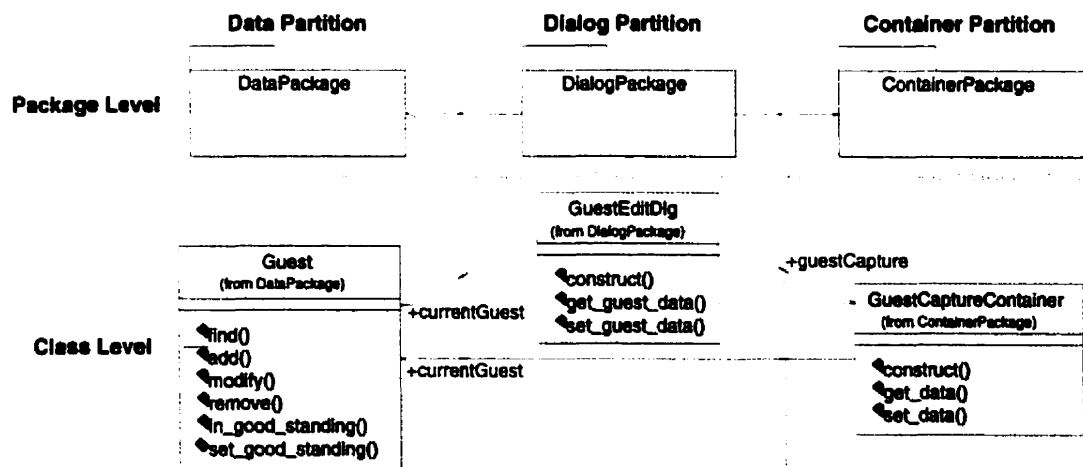


Figure 21. Concrete Relation has no Corresponding Abstraction

inconsistency between a package diagram and a class diagram. At the package level, we find three major classifiers: *DialogPackage*, *DataPackage*, and *ContainerPackage*. Also, there are dependency relationships between *DialogPackage* and all the others. The class diagram reveals more information about the contents of the packages—*DataPackage* contains the class *Guest*, *DialogPackage* contains the class *GuestEditDlg*, and *ContainerPackage* contains the class *GuestCaptureContainer*. The inconsistency depicted in the figure is based on the observation that there is a lower-level association relationship from *GuestCaptureContainer* to *Guest* but no such relationship between their corresponding higher-level packages (from *ContainerPackage* to *DataPackage*).

2. Abstract relation has not been refined

Higher-level model relationship between classifiers is not reflected between their lower-level refinements. This may indicate that the lower-level view does not correctly realize the higher-level one.

3. Concrete classifier has no corresponding abstraction

Lower-level model element is not assigned to any higher-level element. Although, this inconsistency is more an indication of incompleteness than an error, it nevertheless indicates a problem. The example in Figure 22 shows the case of the class *ReservationCollection* that was not assigned to any higher-level package although it interacts with classes that are part of those packages.

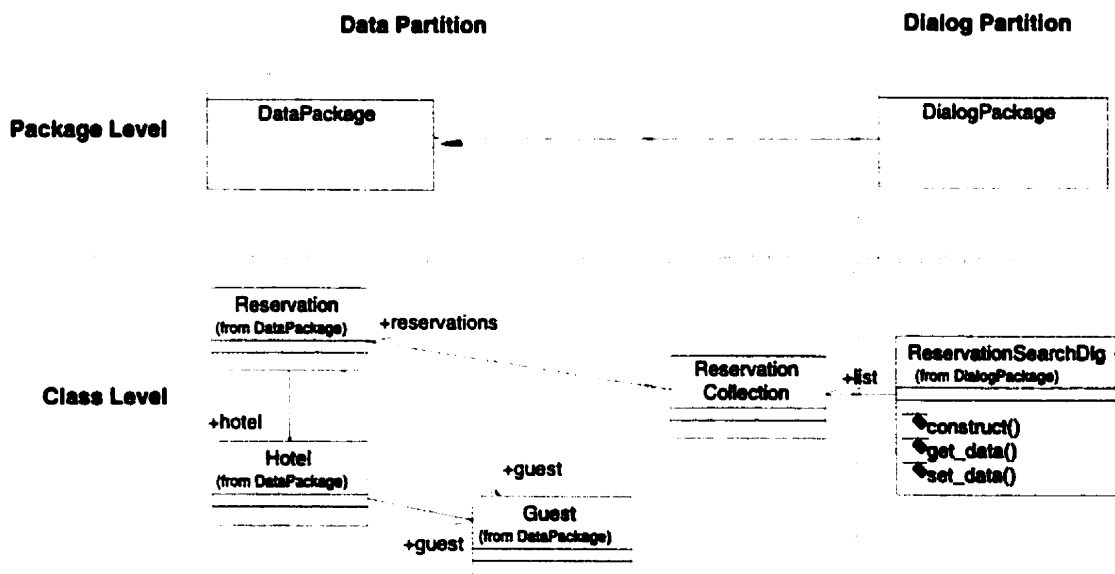


Figure 22. Concrete Classifier has no Corresponding Abstraction

4. Abstract classifier has not been refined

Higher-level model element is not assigned to any lower-level elements. Figure 23 depicts an example of one such inconsistency between a package and a class diagram. The package diagram shows the two classifiers *DataPackage* and *JavaAWT*. The class diagram depicts a refinement of *DataPackage* only, which includes the classes *Reservation*, *Hotel*, and *Guest*. The package *JavaAWT* was not refined. The inconsistency depicted in the figure is based on the observation that there is a dependency from *DataPackage* to *JavaAWT* that was not realized at the class level since the classes corresponding to *JavaAWT* are missing.

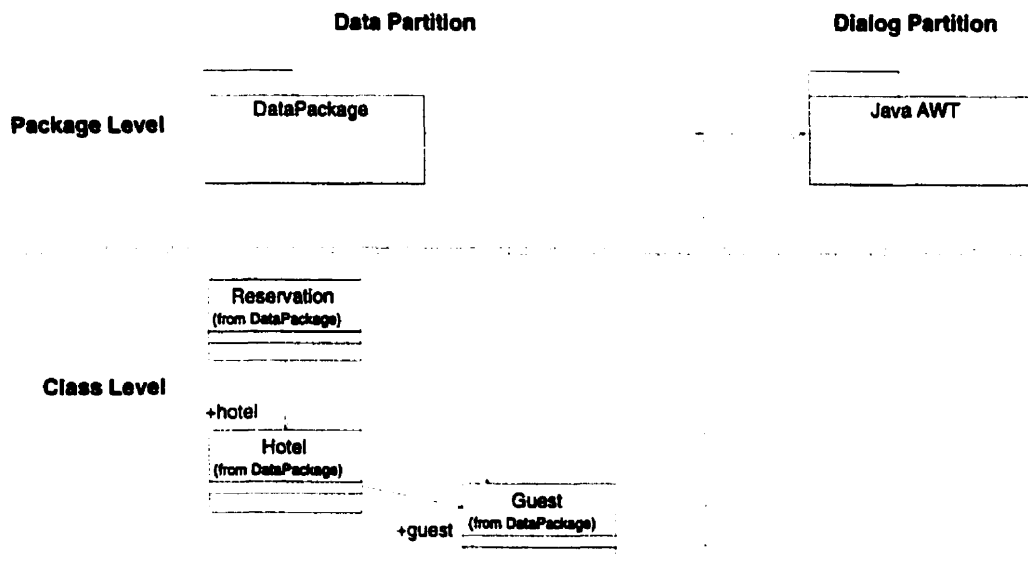


Figure 23. Abstract Classifier has not been Refined

5. Concrete relation is of different type than its corresponding abstraction

Higher-level relation type does not conform to lower-level relation type. For instance, if at a higher level a relation of type dependency is used, but at the lower level an equivalent relation of type association is used then this denotes a type inconsistency between layers.

6. Concrete classifier is of different type than its corresponding abstraction

Higher-level classifier type does not conform to lower-level classifier type. Figure 24 depicts a higher-level classifier *AccountActivity* that is decomposed into a number of sub classifiers (*Transaction*, *Payment*, *Expense*, *Cash*, *Check*, and *Creditcard*). The classifier *AccountActivity* is an interface class,

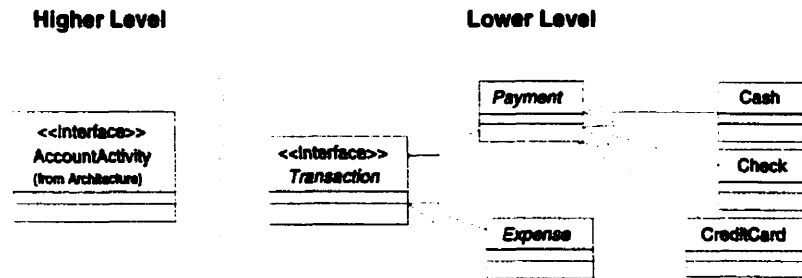


Figure 24. Concrete Classifier is of Different Type than its Corresponding Abstraction

which implies that it will only provide operation stubs without implementing them. The lower-level view has an interface class (*Transaction*) but also two realizations (*Payment* and *Expense*). The inconsistency shown here is based on the observation that the higher level class specifies a partition of the system that only handles an interface. The lower-level representation, however, violates that constraint by also defining realizations.

7. Concrete relation uses abstract classifier instead of its refinement

Model relationship from a lower-level classifier (of one partition) to a higher-level classifier (of another partition) is not reflected between their corresponding high-level classifiers and/or between their corresponding lower-level classifiers. Figure 25 depicts an example of one such inconsistency between a package diagram and a class diagram. At the package level, we find two major classifiers called *DialogPackage* and *DataPackage*. The class diagram reveals more information about the contents of those two packages—*DataPackage* contains the classes *Guest*, *Hotel*, and *Reservation*; and *DialogPackage* contains the class *ReservationEditDlg*. The inconsistency depicted in the figure is based

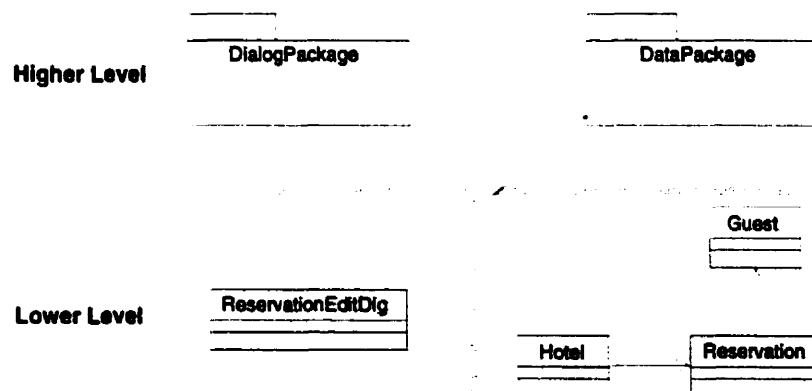


Figure 25. Concrete Relation uses Abstract Classifier Instead of its Refinement

on the observation that there is a dependency relationship from *ReservationEditDlg* to *DataPackage* but no corresponding higher-level relationship from *DialogPackage* to *DataPackage*. Furthermore, there is no lower-level dependency from *ReservationEditDlg* to any of the three classes *Guest*, *Hotel*, and *Reservation*.

8. Abstract relation uses concrete classifier instead of its abstraction

Model relationship from a higher-level classifier (of one partition) to a lower-level classifier (of another partition) is not reflected between their corresponding high-level classifiers and/or between their corresponding lower-level classifiers.

9. Abstract classifier is replicated at the concrete level although refinement exists

An abstract classifier can be replicated at the lower-level if no refinement exists (or is necessary). However, in the case of Figure 26, a refinement exists but it was not used in the middle layer. Note that this case probably denotes more an oversight than an actual inconsistency.

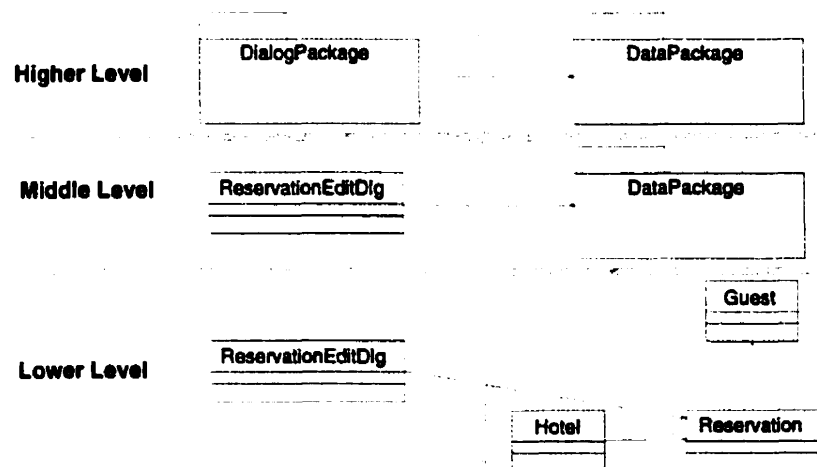


Figure 26. Abstract Classifier is Replicated at the Concrete Level Although Refinement Exists

10. Concrete classifier is assigned to multiple abstract classifiers

A lower-level element corresponds to multiple higher-level elements. This inconsistency may indicate that the partitioning of the system is ambiguous or that the lower-level element in question does in fact belong to an altogether third higher-level element (e.g., a library class that is used widely

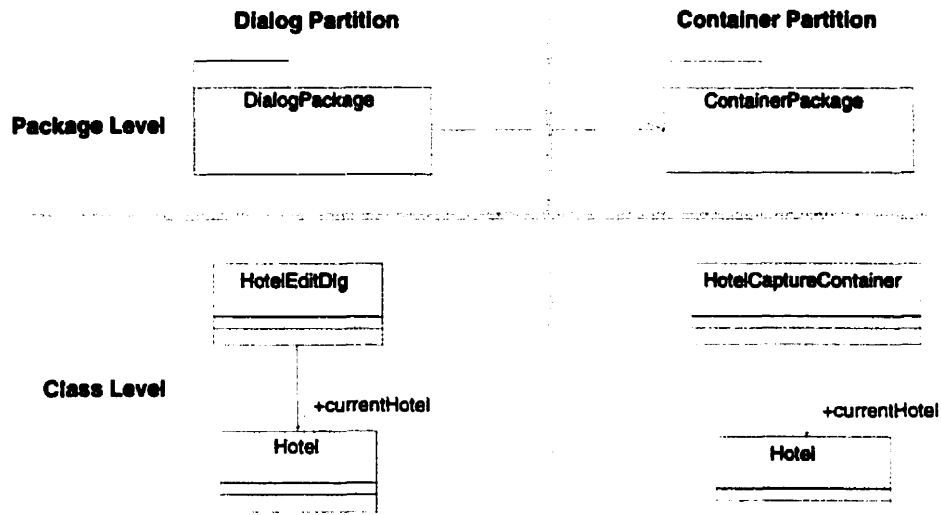


Figure 27. Concrete Classifier is Assigned to Multiple Abstract Classifiers

throughout the application). Figure 27 shows one such example where the class *Hotel* is assigned to two separate packages.

11. Cardinality of refinement does not match its abstraction

Cardinality between higher-level classifiers does not conform to lower-level classifiers. Figure 28 depicts an example of a cardinality mismatch between classes of a higher level and their corresponding lower level classes. The higher level view shows the two classes, *Guest* and *Transaction*, with an association between them indicating that there can be one or more (i.e., many) transactions per

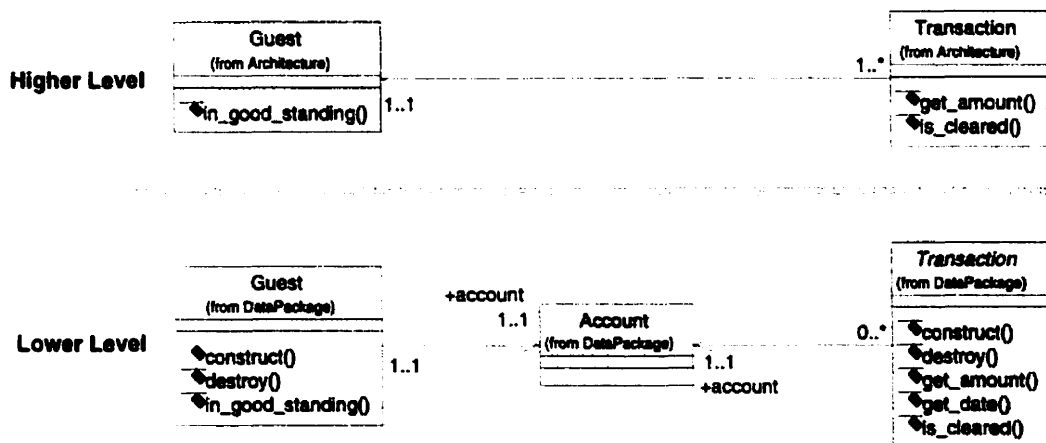


Figure 28. Cardinality of Refinement does not Match its Abstraction

guest and each transaction can involve only one guest. The lower level view introduces the helper class *Account* that masks the true cardinality and interdependency between *Guest* and *Transaction*. However, it can be inferred that if there is at least one account per guest and at least zero transactions per account, then there can be at least zero transactions per guest. The views are inconsistent since the higher level view assumes at least one transaction per guest.

12. Direction of concrete relation does not match its abstraction

Relationship between higher-level classifiers does not conform to lower-level classifiers. Figure 29 depicts an example of an association relationship from the higher-level class *Account* to *Deposit* (indicating that *Account* calls methods produced by *Deposit*). The lower level view introduces the helper class *Transaction* and indicates an association relationship from *Transaction* to *Account* as well as a realization relationship from *Deposit* to *Transaction*. Since the realization relationship implies that *Deposit* realizes (implements) *Transaction*, one can safely assume that *Deposit* inherits all features from *Transaction* which includes *Transaction*'s association to *Account*. It follows that *Deposit* should have an association to *Account* and not vice versa as the higher level view suggests.

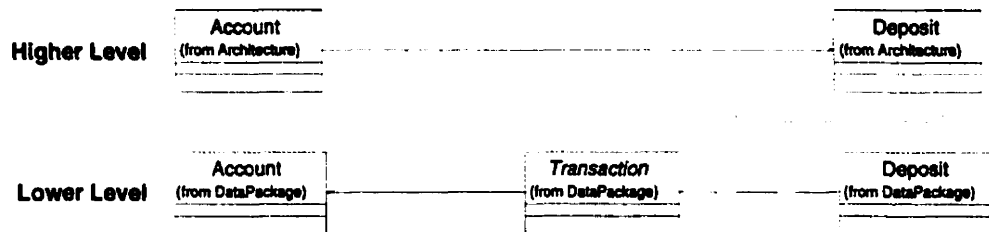


Figure 29. Direction of Concrete Relation does not Match its Abstraction

13. Concrete classifier does not replicate a method of its abstraction

A lower-level element corresponding to a higher-level element does not exhibit the same operations (services). Figure 30 depicts an example of a higher-level classifier *Guest* that was decomposed into two sub classifiers (*Guest* and *Account*). The classifier *Guest* describes an interface of two operators called *in_good_standing* and *get_balance*. Similarly, the decomposed lower-level describes an interface on its own. The inconsistency shown here is based on the observation that the lower-level view must at least exhibit the same services as the higher-level view. In our concrete example, the lower-

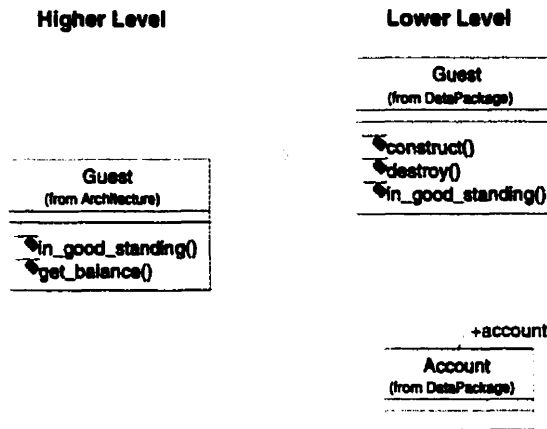


Figure 30. Concrete Classifier does not Replicate a Method of its Abstraction

level view provides the *in_good_standing* interface as required by the higher-level view but does not provide a *get_balance* interface.

14. Concrete classifier does not replicate an attribute of its abstraction

A lower-level element corresponding to a higher-level element does not exhibit the same attributes. This type of inconsistency is analogous to the above one (Figure 30).

14. Concrete method is of different type than its abstraction

A higher-level methods does not have the same interface as its corresponding lower-level method. For instance, in Figure 31, the higher-level class *Account* defines two methods called *get_amount* and *do_transaction*. The *get_amount* method further specifies that it returns a value of type *integer*. The lower-level diagram refines *Account* into classes *Account* and *Transaction*. Together, the two classes

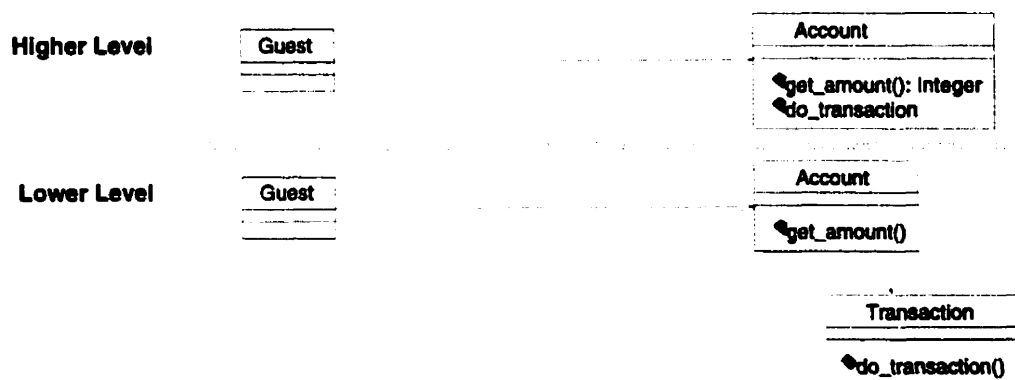


Figure 31. Concrete Method is of Different Type than its Corresponding Abstraction

provide the same methods as the higher-level class *Account*; however, the interface for the method *get_amount* is different.

16. Concrete attribute is of different type than its corresponding abstraction

A higher-level class corresponding to some lower-level classes does not have the same attribute interface. This case is very similar to the one above.

17. Abstract and public method is hidden in refinement

A lower-level classifier does not provide the same public interface as its corresponding higher-level abstraction. Consider the example in Figure 31 again. There we saw the case of an incompatible method interface. The example, however, also showed another type of inconsistency. There we can see that the class *Guest* accesses the class *Account*, which implies that the class *Guest* could access either one of the two methods *Account* provides. At the lower level we see that both classes provide the same interface (let us ignore the return type inconsistency). However, at the lower level the class *Guest* can only access one of the two methods (*get_amount*). The other method is not directly accessible to *Guest*.

18. Abstract and public attribute is hidden in refinement

A lower-level classifier does not provide the same public attributes as its corresponding higher-level abstraction. This type of inconsistency is analogous to the case above.

19. Abstract pre-conditions may not become stronger in refinement

Pre condition of instance may not become stronger (or be strengthened). Note: child classes may weaken the pre conditions of methods but not the other way around. If during software development one wishes to substitute off-the-shelf components, then we need to ensure that the minimal requirements (e.g., pre-conditions) are satisfied. For refinement this implies that a component can substitute another component, although the latter must either satisfy the same pre-conditions or weaken it. For instance, if a method “do_transaction” has the pre-conditions that it only works if an account was already created, then it could be substituted with another method that does not have that pre-requisite.

20. Abstract post-conditions may not become weaker in refinement

Post conditions of inheritance may not become weaker. Note: child classes may strengthen the post condition of methods but not the other way around. This case is analogous to above.

21. Abstract invariant may not become weaker in refinement

Invariant of inheritance may not become weaker (or be weakened). Note: child classes may strengthen their invariants but not the other way around. This case is analogous to above.

6.2.2 Inconsistencies in the Generic Dimension

1. Specific relation has no corresponding generalization

Figure 32 depicts a generic class view involving three classifiers: *Reservation*, *GuestCollection* and *Guest*. To illustrate the possible interactions between those generic classifiers, the sequence diagram depicts one possible scenario of how an instance of *Reservation* (*r1*) calls an instance of *GuestCollection* (*gc*) and an instance of *Guest* (*g*). It can be observed that the instance of *Reservation* interacts with the instances of *GuestCollection* and *Guest*. The inconsistency depicted is based on the fact that the generic view supports the interaction from *Reservation* to *GuestCollection*, but, does not allow *Reservation* to interact with *Guest*.

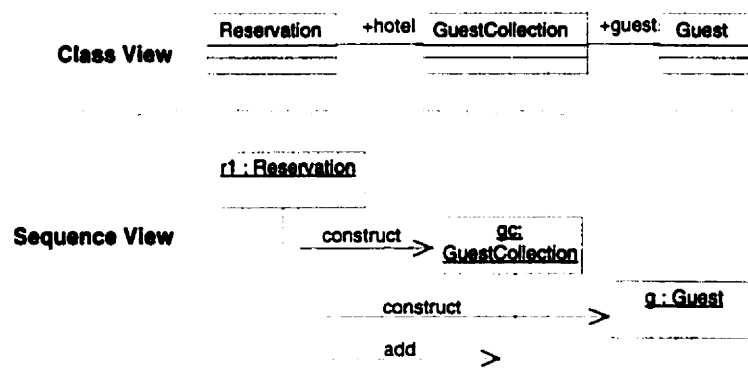


Figure 32. Specific Relation has no Corresponding Generalization

2. Generic relation has never been instantiated

This case denotes more an incompleteness (oversight). It indicates that a generic relation has never been instantiated in any one of the specific models.

3. Specific classifier has no corresponding generalization

This case corresponds to case (1) above. The only difference is that a specific classifier (instead of relation) is used that has not been defined at the generic level (e.g., an object that does not have a class).

4. Generic classifier has never been instantiated

This case denotes more an incompleteness than an inconsistency. It indicates that a generic classifier has never been instantiated in any one of the specific models.

5. Specific relation is of different type than its corresponding generalization

Specific relation instance does not conform to relation type. This case is analogous to the abstract/concrete counterpart discussed in 6.2.1.

6. Specific classifier is of different type than its corresponding generalization

Specific classifier instance does not conform to classifier type. This case is analogous to the abstract/concrete counterpart discussed in 6.2.1.

7. Cardinality of generic classifiers does not match specific scenarios

Cardinality between generic classifiers (e.g., classes) does not conform to specific scenarios (e.g., sequences). Figure 33 depicts a relationship between the two generic classifiers *Guest* and *Account* that was specialized into a more detailed sequence diagram depicting an interaction scenario. The sequence diagram depicts one instance of *Guest* called *Peter* and two instances of *Account* called *a1* and *a2*. The sequence diagram further indicates that both instances of *Account* are known to *Guest* which implies that at the generic level there should be at least two *Accounts* per *Guest* (there could possibly be more, but not fewer). The inconsistency between the diagrams is based on the observation that the sequence view depicts a cardinality of classifier instances, which is not supported by the class diagram.

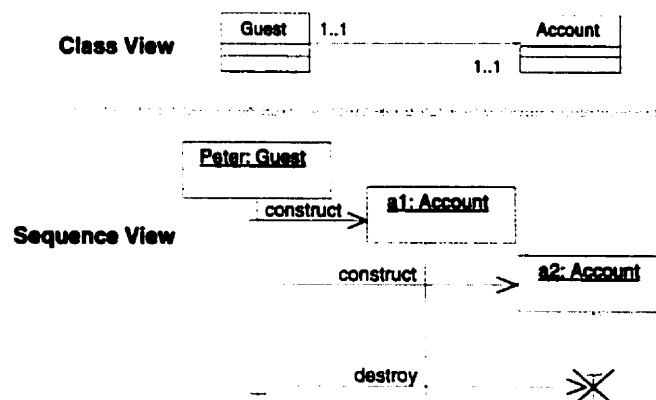


Figure 33. Cardinality of Generic Classifiers does not Match Specific Scenarios

8. Direction of specific relation does not match its generalization

Figure 34 depicts a generic class view involving the two classifiers *GuestEditDlg* and *GuestCaptureContainer*. To illustrate the possible interactions between those generic classifiers, the sequence diagram depicts one possible scenario of how an instance of *GuestEditDlg* (*gedlg*) calls an instance of *GuestCaptureContainer* (*dcc*). The inconsistency shown here is based on the observation that instances call each other (*set_data* in both direction); however, the generic view only supports uni-directional interaction. As such, only the instance of *GuestEditDlg* is allowed to call *GuestCaptureContainer* and not vice versa.

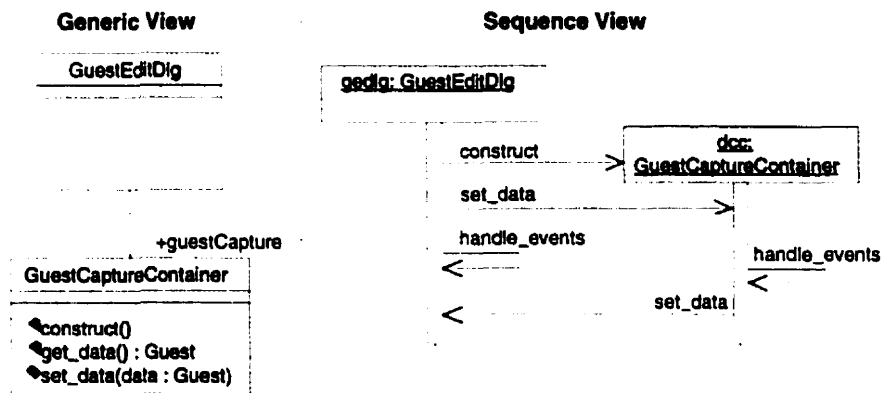


Figure 34. Direction of Specific Relation does not Match its Generalization

9. Generic method has never been instantiated

This case is analogous to its abstract/concrete counterpart discussed in 6.2.1.

9. Generic attribute has never been instantiated

This case is analogous to its abstract/concrete counterpart discussed in 6.2.1.

11. Specific method is of different type than its corresponding generalization

This case is analogous to its abstract/concrete counterpart discussed in 6.2.1.

12. Specific attribute is of different type than its corresponding generalization

This case is analogous to its abstract/concrete counterpart discussed in 6.2.1.

13. Specific view uses a method that is not defined in generic classifier

Figure 35 depicts a generic class view involving the two classifiers *ReservationSearchDlg* and *ReservationCollection*. To illustrate the possible interactions between those generic classifiers, the

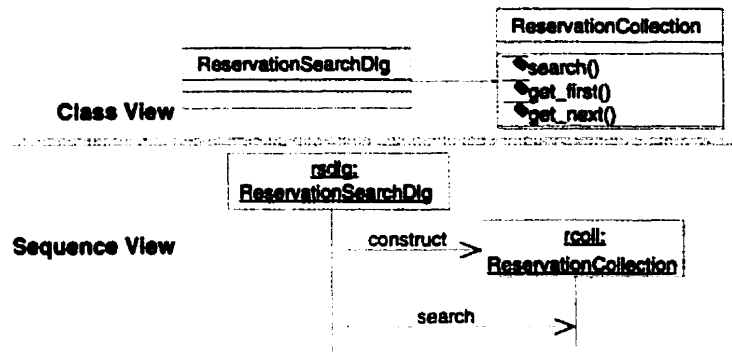


Figure 35. Specific View uses a Method that is not Defined in Generic Classifier

sequence diagram depicts one possible scenario of how an instance of *ReservationSearchDlg* (*rsdlg*) calls an instance of *ReservationCollection* (*rcoll*). The scenario first creates the collection (using operator *construct*), and then searches it (using operator *rcoll*). The inconsistency shown here is based on the observation the *rsdlg* uses two operators to access *rcoll* (*construct* and *search*) but one of it was not defined in the generic view.

14. Specific view uses an attribute that is not defined in generic classifier

This case is analogous to the previous case.

15. Specific classifier has not been assigned to generic classifier

Note that this case is more an indication of oversight than inconsistency. Figure 36 depicts an example of a generic class view involving the two classifiers *ReservationSearchDlg* and *Reservation*. To

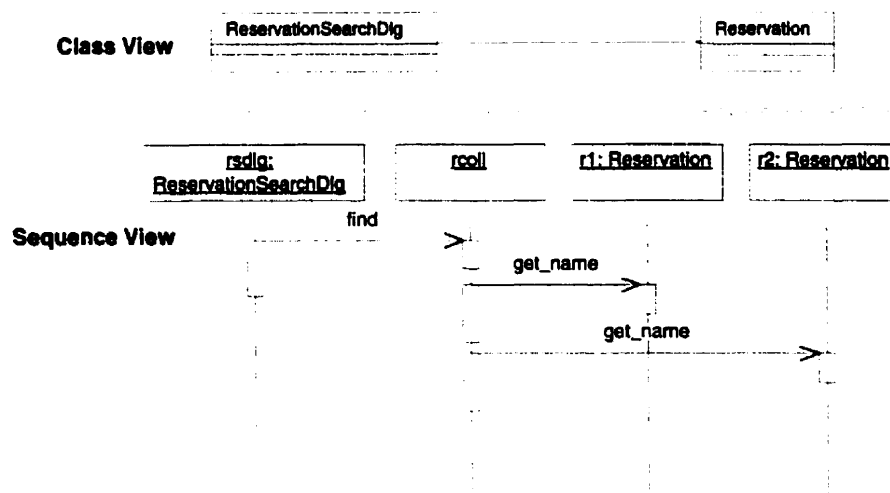


Figure 36. Specific Classifier has not been Assigned to Generic Classifier

illustrate the possible interactions between the generic classifiers, the sequence diagram depicts one possible scenario of how an instance of *ReservationSearchDlg* (*rsdlg*) calls another instance called *rcoll*, which in turn calls two instances of *Reservation* (*r1* and *r2*). The inconsistency shown here is based on the observation that no classifier type was associated with *rcoll*.

16. Specific relation has not been assigned to generic relation

Analogous to next case.

17. Generic pre-condition is violated in specific view

Figure 37 defines a generic condition that the *construct* method may only create an object of type *Guest* if the operation is successful. In the sequence view (specific view) we can see that the operation *construct* created an object, although it is labeled a failure.

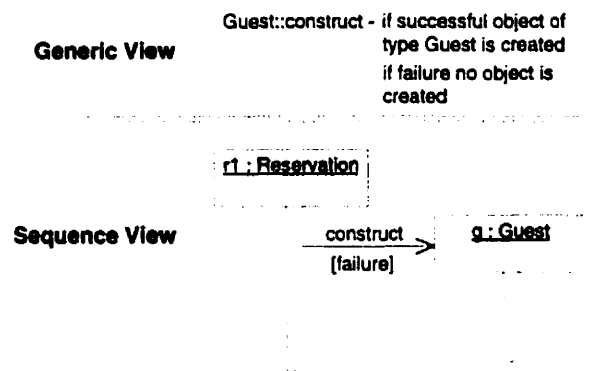


Figure 37. Generic Pre-Condition is Violated in Specific View

18. Generic post-condition is violated in specific view

Analogous to previous case.

19. Specific method used was declared private in generic view

Denotes a case where a generic view published a method that was declared public, but the specific view declared that same method as private.

20. Specific attribute used was declared private in generic view

Analogous to previous case.

6.2.3 Inconsistencies in the Behavioral Dimension

1. Imported guard was not declared in structural view

Figure 38 depicts an example of a generic class view involving the three classifiers *Guest*, *GuestDlg*, and *GuestDB*. The figure also illustrates the statechart diagram belonging to the class *GuestDlg* which shows that the class has three different states depending on the extend of the information capture. The state transition from *guest unspecified* to *guest identified* uses a guard saying that the transition only happens if the object “*guest::is_valid()*” returns true. The inconsistency is based on the observation that the class corresponding to object *guest* does not have a method or attribute with that name.

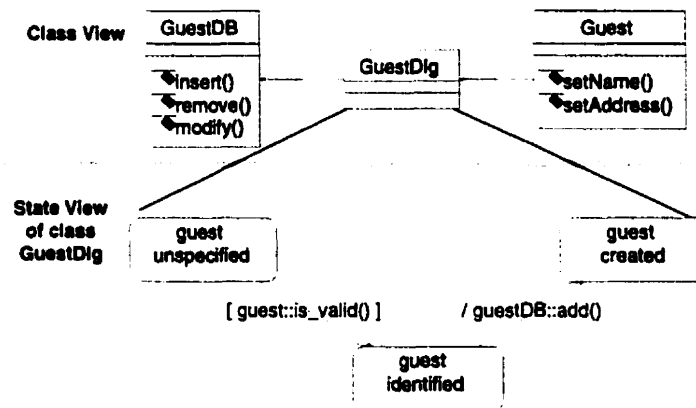


Figure 38. Structural View does not Support all Behavioral Needs

2. Imported trigger was not declared in structural view

This case is similar to the previous case. Figure 38 also depicts an example in case of the state transition from *guest identified* to *guest created*. There it says that the trigger of “*guestDB::add()*” needs to have been called for that transition to happen. Since the *add* method is not part of *GuestDlg* it is expected that it must be part of another class. The class *GuestDB*, however, does not have a method of that name.

3. Structural view does not allow an interaction as required by guard

This inconsistency type indicates a case where interactions on the behavioral side are not reflected in the structural declaration. Figure 38 shows one such example in context of the state transition

from *guest unspecified* to *guest identified*. It can be seen that this transition depends on an imported guard from the class *Guest*. Even if *Guest* would have a method called *is_valid()*, there would still exist an inconsistency with respect to the allowed interactions. It must be noted that the current definition of the structural view (the class diagram) does not allow *GuestDlg* to access *Guest*.

4. Structural view does not allow an interaction as required by trigger

This inconsistency is analogous to the previous one. Figure 38 would not have an inconsistency with respect to this case if the state transition between *guest identified* and *guest created* would have a method called *add()*. The class *GuestDlg* already depends on *GuestDB*, thus, the structural definition supports the required interaction.

5. Relationship between classes is not reflected in statechart

This type of inconsistency is the counterpart to the four types we discussed previously. It states that if there is a structural dependency between any two classes, then statecharts belonging to those classes must also interact.

6. Method was declared “query” but is used for non-circular state transitions

This case indicates that the declaration of methods has an impact on how that method can be used. In Figure 39, we can see a class declaration which defines a set of attributes and methods as well as some properties of them. The figure also defines a statechart of that class indicating that the class may go

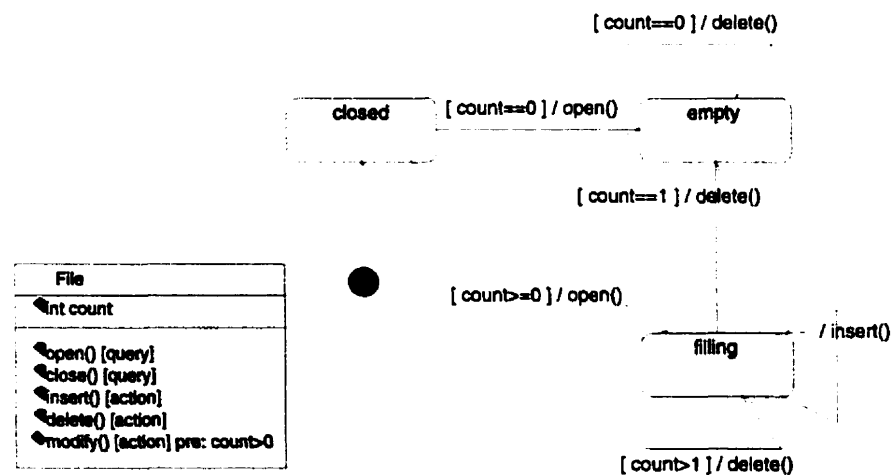


Figure 39. Structural Declaration does not Match its Usage

through the three states *closed*, *empty*, and *filling*. It can be observed that the method *open* was declared as a query which denotes that it cannot change states. The “query” declaration is meant to indicate methods that do not alter states if they are called. This declaration is inconsistent with the statechart view where the *open* method is used for state transitions between *closed* and *empty* as well as between *closed* and *filling*.

7. Method was declared “action” but is used for circular state transitions only

Figure 39 also shows an inconsistency of this type. If a method is declared as action, it is implied that this method may cause a state transition if invoked. For instance, the method *delete* causes both regular and circular state transitions. The inconsistency is with method *insert* which was defined as action but is never used for transitions between two different states.

8. Method was declared “activity” but is used for state transitions

If a method is declared as “activity,” it is implied that this method is of longer duration. For instance, a method call that waits for a user to press a button is of that type. Activities are usually associated with states. Should an activity be associated with a transition than this denotes an inconsistency.

9. Guards leaving state are not mutually exclusive

Figure 39 also shows an example of this case. The state transition *open* has two exits from the state *closed*. In order to decide which transition to use (e.g., the one to *empty* or the one to *filling*), guard conditions must be attached. For the state diagram to be deterministic, those guard conditions must be mutually exclusive. Currently, there exists an inconsistency in that the two guard conditions overlap in case of “count==0.”

10. Guards/Trigger pre- or post condition does not match method pre- or post condition

This type of inconsistency indicates that guard and trigger conditions must match their declaration. In Figure 39, the method *delete* has the condition that it may only be invoked if “count>0.” If it should be invoked otherwise, the result would be undefined. The statechart view, however, defines a state transition for *delete* with the guard condition “count==0,” which denotes an inconsistency.

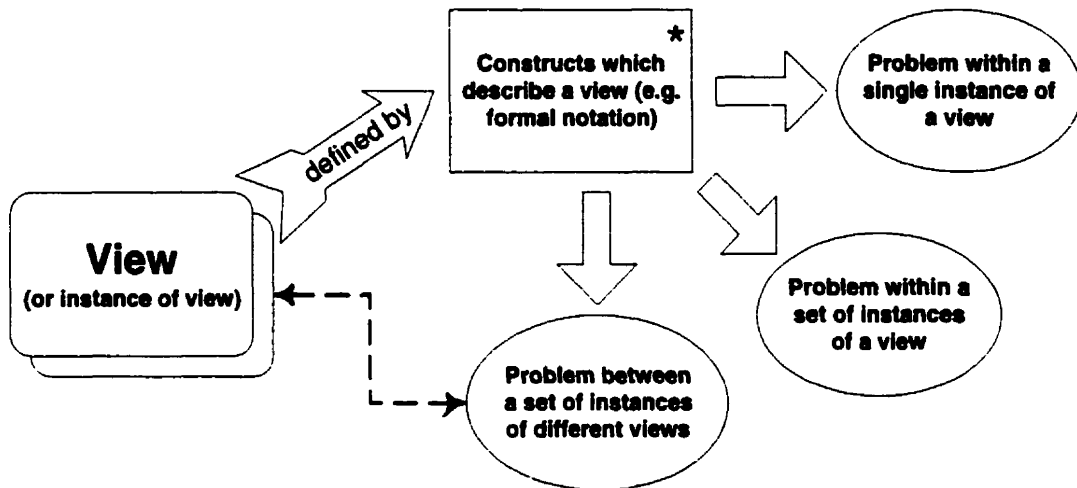


Figure 40: Categories of Mismatches

6.3 Classification of Inconsistencies

With respect to view inconsistencies we distinguish between three basic types (Figure 40): (1) inconsistencies within a single instance of a view; (2) inconsistencies between a set of view instances of the same view dimension; and (3) inconsistencies between a set of view instances of different view types and dimensions.

In the previous section we did not represent a complete list of inconsistencies, but have instead focused on inconsistencies between views of different types and dimensions. The reasons for that are simple: most current view integration approaches make the simplistic assumptions of consistency checking between the same view instances and types (first and second categories) [Grundy and Hosking 1996] [Wang et al. 1997]. Actually, even UML defines simple consistency rules at those levels. With this simplistic assumption, consistency checking is often doable via simple comparison. In Section 7 we will show that view integration is complicated by having to consider different view dimensions and types. Our approach works for all three categories.

6.4 Act in the Presence of Inconsistencies

This section emphasized the existence of inconsistencies that are due to the gap between models and views. "While an approach that is intolerant of inconsistencies and multiple perspectives may be

adopted (and *is* adopted by many organizations that wish to enforce a disciplined development policy), there appears to be mounting evidence that such an approach is not realistic, and that software developers prefer to work with multiple views ... and languages ... in which inconsistency is tolerated” [Nuseibeh 1994].

What this implies is that inconsistencies are not inherently bad. They only become bad if one is not aware of their existence or does not react to them. “Inconsistencies are inevitable in software development ... processes and products. They provide a focus for further development ..., and can be regarded as ‘desirable’ in that they highlight issues that need further attention. As such, they should be tolerated, analyzed and acted upon” [Hunter and Nuseibeh 1997]. There have been numerous approaches to how one should act in the presence of inconsistencies (e.g., [Balzer 1991], [Narayanaswamy and Goldman 1992], and [Hunter and Nuseibeh 1998]).

6.5 Summary

This section discussed potential negative impacts of modeling via multiple views. We started off by presenting some examples of inconsistencies among two or more types of diagrams. We then generalized and presented a list of inconsistencies along the three dimensions presented in the previous section. We identified those in the course of evaluating a large number of scenarios. We also emphasized that this work addresses the most complicated and presently mostly unsolved types of inconsistencies—the inconsistencies between different types of views and view dimensions. Finally, we very briefly discussed the impact of inconsistencies. With that we wanted to show that having inconsistencies is not a bad thing per se; however, not knowing about them or not resolving them in time may result in serious consequences (e.g., negative impact on cost, schedule, etc.).

7 Our View Integration Framework

Nuseibeh, Kramer, and Finkelstein [Nuseibeh et al. 1994] wrote that the term view inconsistency indicates that some form of rule that expresses a relationship between model elements has been broken. It is these kinds of inconsistency rules we are aiming for in our integration work. However, rules alone are only limited useful if they cannot be applied automatically to check for consistency. This implies that there is more to view integration than consistency rules and model constraints. What we also need is an environment where we can apply those rules in a meaningful way.

7.1 Overview

As discussed in previous sections, views are abstractions of information relevant to specific concerns. Views are structured in such a fashion that they arrange and present information in the most meaningful way to stakeholders (developer, architect, customer, etc.). Figure 41 depicts the outline of a generic development framework. Software system information is stored in a system model. Stakeholders, who are primarily architects and designers from a product modeling perspective, access that model and manipulate it throughout the course of the development life cycle. The model itself is not (or should not) be accessed directly. Instead, model information is projected (abstracted) into views. Views are then manipulated and changes within these views are then reconciled with the underlying model.

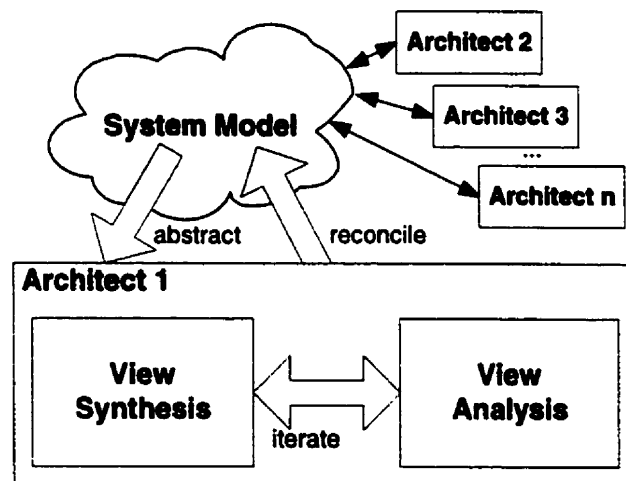


Figure 41: Model-based Development—a view independent representation

We refer to the manipulation of views as synthesis although it must be noted that this type of synthesis is mostly manual. Through synthesis, model information is created, modified, and deleted. We need view analysis to ensure the consistency and conceptual integrity of views and their changes. The view analysis component is the focus of this work and is described next.

7.2 View Integration Framework

To address view integration, we have investigated ways of describing and identifying the causes of modeling mismatches across UML views. To this end, we have devised and applied a view integration framework, accompanied by a set of activities and techniques for identifying inconsistencies in an automatable fashion. Our view integration framework is accompanied by a set of activities and techniques which are depicted in Figure 42 (this figure is a refinement of Figure 41). As it can be seen, our view analysis component incorporates three major activities called *Mapping*, *Transformation*, and *Differentiation*.

The system model in Figure 42 represents the model base (e.g., UML model) of the designed software system (recall Figure 41). To create and manipulate these models, there is a need for a synthesis component (View Synthesis). To date, numerous (system) models and synthesis tools have been proposed. For instance, our framework is integrated with the Unified Modeling Language, which is used as the system model, and Rational Rose™, which is used as its synthesis tool. We need *View Analysis*

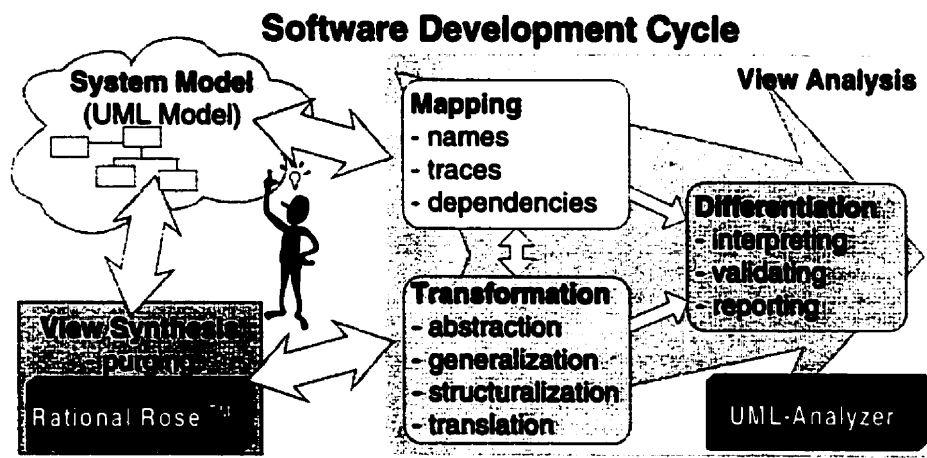


Figure 42. View Integration Framework

during software evolution whenever modeling information is added or modified since changes must be validated against the rest of the system model to ensure their consistency.

Our view integration approach exploits the redundancy between views. For instance, if view A contains information about view B, this information can be seen as a constraint on B. The view integration framework is used to enforce such constraints and, thereby, the consistency across views. In addition to constraints and consistency rules, our view integration framework also defines *what* information can be exchanged and *how* information can be exchanged. This is critical for a scalable and automated inconsistency identification process. View Analysis, a continuous activity, involves the following major (sub)activities:

- ***Differentiation***: traverses models and views to identify potential inconsistencies within and between modeling elements. Inconsistencies can be identified automatically through violations of consistency rules that are validated against the system model. Automated differentiation strongly depends on mapping and transformation, the other major activities of view analysis.
- ***Mapping***: identifies and cross-references related modeling elements that describe overlapping and thus redundant pieces of information (e.g., as in Figure 6 on page 28). Mapping is often done manually via naming dictionaries or traceability matrices. Mapping simplifies differentiation in that it defines *what* modeling elements should be compared.
- ***Transformation***: extracts and converts modeling elements of views in such a manner that they (or pieces of them) are more understandable in the context of other views. Transformation simplifies differentiation in that it makes model elements of different types and shapes easier to compare (*how*). It can be automated using abstraction, generalization, structuralization, and translation (discussed later).

Each view integration activity represents a complex problem in itself. This work will discuss the details of those activities later. It must be stressed that those activities are not separate but, instead, they need to be put together to simplify and improve the overall task of inconsistency detection.

Figure 43 depicts the relationship between *Mapping*, *Transformation*, and *Differentiation* in the context of four inconsistency detection scenarios. In order to compare the two user-defined views A and

B (containing user-defined modeling elements), we could either a) compare them directly; b) transform (convert) A into 'something like B' so that A becomes more easily comparable to B; c) transform B into 'something like A' so that B becomes more easily comparable to A; or d) transform both A and B into 'something like C' so that they are more easily comparable in the context of C. The role of mapping is to scope down transformation and differentiation by specifying *what* information needs to be exchanged and *what* information needs to be compared. The role of transformation is to enable more direct comparison by converting modeling elements into similar types and thus defining *how* modeling elements can be compared. Transformation also extends the model in that new (automatically generated) modeling elements are derived from user-defined ones. For instance, the "something-like" boxes represent derived model information that must also be stored.

Figure 43 also shows that multiple input sources must frequently be used to automatically transform a modeling element from one type into another. Although model elements of different types of views may overlap in what they are meant to convey, the form and the boundaries of those descriptions may vary. For instance, in order to generalize the relationships between two classes one needs to analyze

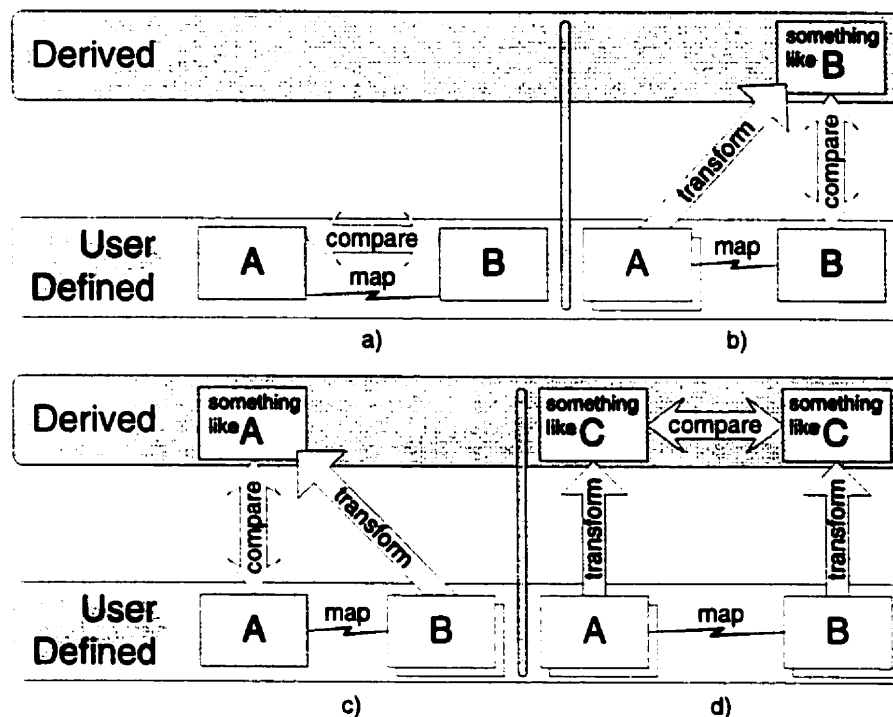


Figure 43. View Transformation and Mapping to Complement View Comparison

the instances of those classes and their interrelationships. This implies that in order to convert modeling information from type A into type B, the transformation method may have to consult additional information about A and B (multiple sources of information) to transform A, B or both of them.

This work will primarily investigate the activities of transformation and differentiation, and secondarily the activity of mapping since we try to emphasize automation since transformation and differentiation embody the strongest prospects for automation. Although mapping is equally important for view integration, we nevertheless find that mapping is also the hardest to automate. Mapping requires intrinsic knowledge about the relationships between modeling elements of systems, a problem also known as the traceability problem [Gotel and Finkelstein 1994]. Although mapping has received considerable attention in the research community, to date no truly automatable solutions have been found to adequately address it.

The other reason why we consider *automating* mapping less important is related to the frequency of that activity. Mapping between modeling elements needs to be defined only once per model element permutation (between any two model elements) whereas transformation and differentiation between that same permutations may happen multiple times throughout the project evolution. The latter is caused by the continuous need of ensuring consistency between views that is triggered by changes. Not being able to automate mapping, however, does not preclude automated view integration. Our framework has been built in such a manner that it can handle incomplete mapping information. Furthermore, our consistency checking approach can also detect missing or invalid mapping information.

To demonstrate our integration approach, we will illustrate it in the context of the Unified Modeling Language (UML). Our framework and its activities are also applicable to other sets of heterogeneous views, although some transformation and comparison rules might have to be adapted. To date, we have applied our view integration framework on several UML views (class, object, sequence, and statechart views). We have also expanded the use of the framework beyond UML, to architectural styles (e.g., C2 [Taylor et al. 1996], pipe-and-filter [Shaw and Garlan 1996], layered, etc.) where we validate consistency between C2 and UML views [Egyed and Medvidovic 2000] and between AAA models [Gacek 1998] and UML views [Egyed and Gacek 1999].

This work will also demonstrate a prototype tool, called UML/Analyzer, which implements a part of our framework. Although it is our goal to provide as much automation as possible, we do not believe that full automation for view integration is always feasible; consistency checking will likely incorporate a sizeable human element. However, we do believe that even partial automation can save considerable time and effort and we will give examples later. The following subsections will describe our approach in detail.

7.3 Simple Model Transformation

The importance of transformation is the conversion of modeling information between different types and views in such a manner that they become more easily comparable. This section will first discuss the aspects of transformation that are more challenging than simply converting information. We will therefore introduce simple transformation techniques, explain them in some detail, and then elaborate on how those simple transformation techniques can be integrated into complex ones.

At first glance, view transformations may appear as being strongly dependent on the types of views involved. Although this is generally true, we nevertheless found that views can be grouped into categories and transforming between those categories often involves similar concepts. In Section 5, where we discussed models, views, and model elements, we found that views can be categorized into three major dimensions: abstraction, generality, and dynamism (recall Figure 15).

Figure 44 depicts our view space from a different perspective. As discussed in Section 5.4, model elements belonging to views can be placed into regions in the view space. Figure 44 shows that, by transforming model elements, their positions in the view space change. For instance, a transformation process could make a set of model elements more abstract (up arrow), more structural (left arrow), or more generic (forward arrow). Similarly, transformation could achieve the opposite: making model elements more concrete, behavioral, or specific. The circular arrow in the middle denotes a case of transforming model elements from one type to another without “moving” in the view space. We categorize transformation according to this structure,.

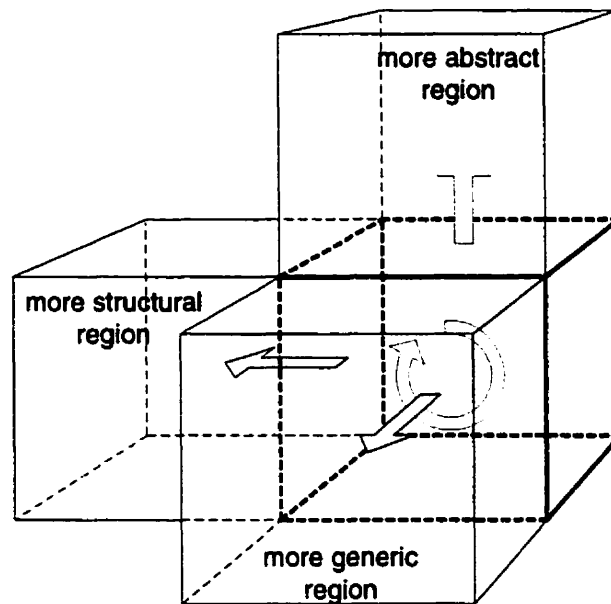


Figure 44. Transforming Model Elements between Regions in the View Space

The eight quadrants of the view space we discussed in Section 5.5 should be seen as categorizing the relative dependency between any two given views (or model elements). For instance, if there are three class diagrams A, B, and C of decreasing level of abstraction, then between A and B, A is abstract and B is concrete but between B and C, B is abstract and C is concrete. The quadrants (regions) depicted in Figure 44 therefore show the relative relationships between any two given views.

Having three dimensions of views implies four types of transformation axes (Figure 45 depicts these four types): *Abstraction* to capture transformation between abstract and concrete views; *Generalization* to capture transformation between generic and specific views; *Structuralization* to capture transformation between structural and behavioral views; and *Translation* to capture transformation within a single quadrant (both input and output type are of the same level of generality, behaviorism, and abstraction as in the transformation between sequence and collaboration diagrams).

We call the upward transformation process in Figure 45, which yields a more abstract view, *abstraction*. Abstraction takes information and simplifies it. For instance, a class diagram can be abstracted into a more abstract class diagram (see Section 7.3.1). It is important to note that abstraction

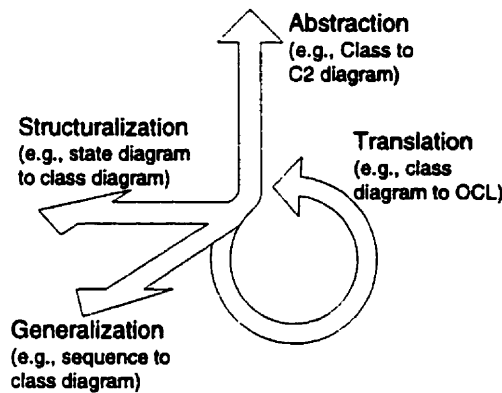


Figure 45. View Dimension and View Transformation Axes

changes a view's positioning in the view space only along the abstract-concrete dimension. A pure abstraction process does not change a view's generality or behaviorism.

The sideways transformation process in Figure 45 is called *structuralization*. Structuralization takes information and extracts its configuration (structure). For instance, a state diagram can be structuralized into a class diagram (see Section 7.3.3). It is again important to note that structuralization changes a view's positioning in the view space only on the behaviorism level without changing a view's abstraction or generality.

The forward arrow in Figure 45 is called *generalization*. Generalization takes information and merges different interpretations to yield more comprehensive information. For instance, a sequence diagram depicts only scenarios. It is hard to generalize from a single scenario onto general structure and/or behavior. However, by merging scenarios together we get a more general view (e.g., a class diagram). Again, generalization only changes a views positioning at the generality level.

Figure 46 shows the transformation techniques we are currently supporting in our framework. The arrows depict the transformation methods and the heads of the arrow depict the directions of the transformations. The thick arrows are currently tool supported via our UML/Analyzer tool discussed in Section 8. The dashed thick arrow is also tool supported via the SCED tool developed by Koskimies, Systä, Tuomi, and Männistö [Koskimies et al. 1998]. The remaining arrows are model supported and will also be discussed later in Section 7.3. Model supported implies that in this work we will describe ways of

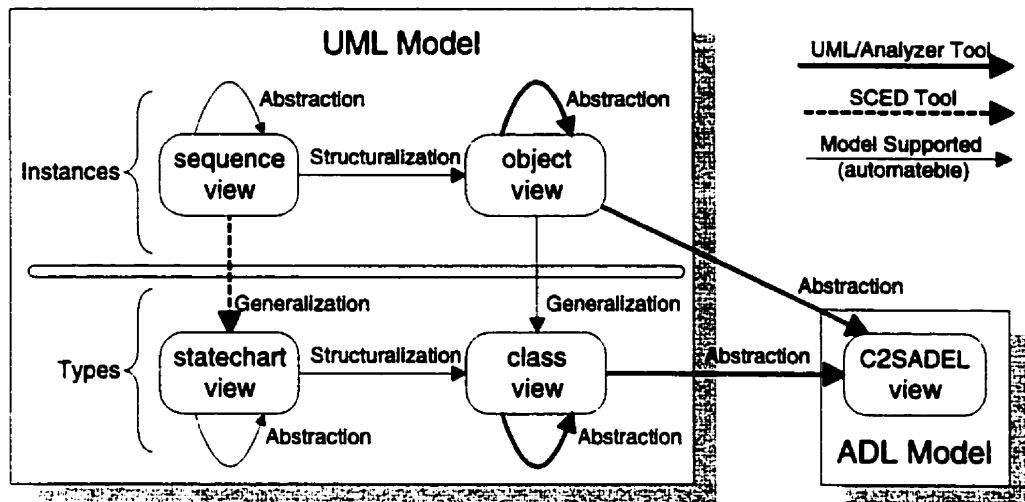


Figure 46. Transformations Currently Supported

automating them, although tool support has not been created yet. The scope and limitations section (Section 4) already discussed the reasons for this decision.

All arrows in Figure 46 are generally uni-directional because the direction of automatic transformation tends to go from concrete to abstract, from behavior to structure, and from specific to generic. The rationale for this is that our transformations tend to go from more information to less information (e.g., abstraction removes lower-level details not significant on an abstract level, generalization merges commonalities between scenarios, and structuralization omits behavioral information not significant for the configuration). Since reversing transformation implies going from less information to more information, it follows that transformations are generally not reversible. This observation is also confirmed by Koskimies' tool (SCED).

Additionally, it must be noted that even if transformations would be generally reversible (e.g., from structure to behavior) there would be only little value added since for consistency checking the direction of transformation is not important (assuming that a reverse transformation does not convert any additional information that the regular uni-directional transformation would not have transformed - recall scenarios b) and c) in Figure 43). For consistency checking it is only significant that information is transformed (regardless of direction) since consistency checking needs transformation only for comparison purposes.

The following subsections are devoted towards simple model transformation. In order to automate view transformation, we need to automate the abstraction, structuralization, generalization, and translation activities. It turns out that the transformation processes for each of the four transformation types are quite distinct; however, different instances of the same transformation types exhibit similarities. For instance, the techniques used to abstract class diagrams and the ones used to abstract state diagrams are very similar. The same observation can be made about generalization and structuralization. Translation is the cover term for all remaining transformations not covered by above three types. Translation may, thus, vary more strongly.

Above we discussed the importance of transformation in enabling the conversion of model information between different types and views so that they become easier to compare to one another. The following will first discuss the basics of our simple transformation methods which are more challenging than just simply converting information. We will introduce simple transformation techniques, explain them in more detail, and then elaborate on how those simple transformation techniques can be integrated into complex ones. This section will also briefly discuss available tool support.

7.3.1 Abstraction

The process of abstraction deals with the simplification of information by removing details not necessary at a higher, more abstract level. We distinguish between two types of abstraction—*classifier abstraction* and *relation(ship) abstraction*. Both types of abstractions are based on diagrammatic views that use box-and-arrow representations (e.g., class diagrams, state diagrams, etc.). The following subsections describe both abstraction mechanisms.

7.3.1.1 Classifier Abstraction

Classifier abstraction is probably the more intuitive abstraction type since it closely resembles hierarchical decomposition of structures provided in many views. For instance, in UML, layers of packages can be built using a feature of packages that allows them to contain other packages. Thus, a package can be subdivided into other packages, forming a tree hierarchy. In classifier abstraction it are classifiers (e.g., packages, class, states, etc.) that can be grouped (“collapsed”) to yield a more simplified

(ergo abstract) view. The relationships of the collapsed, concrete classes then become part of the interface of the more abstract one. Since UML does not support the composition/decomposition of all types of classifiers, we introduce the concept of a *composite model element* in UML that allows model elements to contain other model elements.

Figure 47 (left) shows a generic example of a classifier abstraction at three levels. The lowest level contains a concrete view (e.g., class diagram) and depicts four classifiers (A-D) and four relations between them (α to δ). The middle diagram is an abstraction of the lower-level diagram where the classifiers B and C are grouped together and form a composite classifier named BC. The relation γ as well as the classifiers B and C are hidden inside the composite classifier BC and not visible any more at the middle-level (however the public interfaces of B and C must still be represented by its abstraction BC). The third and top-most level is a further abstraction that adds another composite classifier BCD that contains the composite classifier BC from the middle level as well as the classifier D from the lowest level. Thus, composite classifiers may also contain other composite classifiers forming a tree like hierarchy. The tree structure is visible through the traces linking the three levels of abstractions in Figure 47 (left).

In the case of the abstracted composite classifiers (e.g., BC), the question remains as to what type they are. For instance, if a couple of concrete classes are collapsed into a single composite class then the result should be a composite classifier that “inherits” the interfaces from the concrete classes. But is the composite box still a *class*? UML distinguishes different types of classes; regular class, utility class,

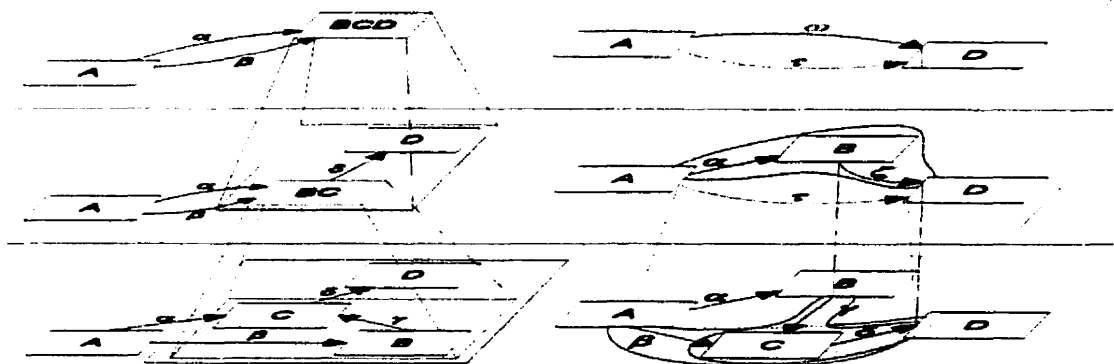


Figure 47: Classifier (left) and Relation (right) Abstraction—Two Approaches

meta-class, abstract class. Furthermore, UML supports stereotypes that can be used to create additional types. If now an abstract class and a regular class are collapsed then is the composite class abstract or regular? For a transformation tool to support automated abstraction, a set of rules must be provided to specify these transformation patterns. These rules will be discussed later.

7.3.1.2 Relation Abstraction

In a relation abstraction [Egyed and Kruchten 1999] it is the relation and not the classifier that serves as a vehicle for abstraction. Relations (with classifiers) can be collapsed into more abstract relations. Relation abstraction is needed since maintaining a strict hierarchy of classifiers (as classifier abstraction requires) is not always possible. For instance, during refinement, model elements may be introduced that may not refine abstract classes but may instead refine the relationships between them.

Figure 47 (right) shows an example of a relation abstraction using again three levels. The most concrete level (bottom) contains four classes A, B, C, and D as well as four relations α , β , γ , and δ . The $\beta \rightarrow C \rightarrow \delta$ pattern is collapsed in the middle layer by introducing the composite relation τ . Similarly, $\gamma \rightarrow C \rightarrow \delta$ is collapsed into the composite relation ζ . The third level takes α and B from the lower level as well as ζ from the middle level and further collapses them into the composite relation ω . The last abstraction again shows that composite relations may themselves contain other composite relations. Like composite classifiers, composite relations form a tree-like structure between the levels. Circular dependencies between composite elements (both classifiers and relations) are not allowed. For instance, ζ is not allowed to be an abstraction of ω .

7.3.1.3 Semantic Rules for Abstraction

The main challenge during abstraction is to hide less important model elements (e.g., classes in class diagrams) and to only depict the remaining classifiers and their relations as part of the higher-level. The challenge we need to address is that at a concrete level, the dependencies between abstract model elements are not explicitly stated. Instead, those dependencies are hidden within the lower-level model elements that we would like to hide. Classifier and relation abstractions, therefore, utilize a technique that

allows groups of model elements (classifier or relations) to be collapsed into high-level composite model elements that summarize their lower-level semantic dependencies. This paper will describe the patterns, rules, and an algorithm necessary to do this.

7.3.1.3.1 Abstraction Examples

Take, for instance, Figure 48, which depicts three simple class diagrams. The first diagram (top) describes the relationships between *Compact Car*, *Car*, and *Driver*. It asserts that a *Car* is *operated-by* a *Driver* and that a *Compact Car* is a type of *Car*. Assuming that we do not care about the class *Car* but

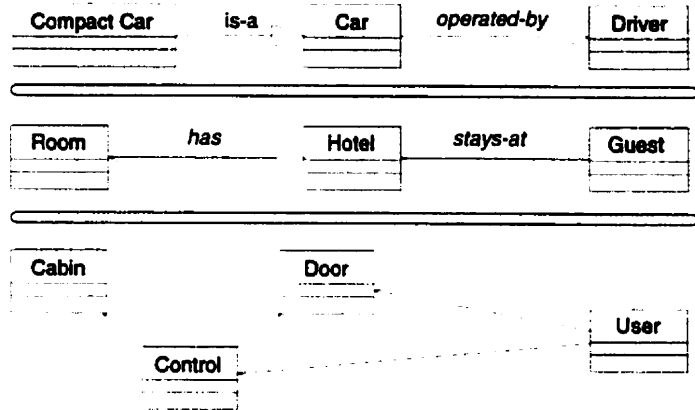


Figure 48. Class Patterns

instead would like to know the direct relationship between *Compact Car* and *Driver*, then we are actually asking for an abstracted version of that class diagram where the “helper class” *Car* and its relationships have been replaced by a simpler relationship. To find out whether there is indeed such a simpler relationship between *Compact Car* and *Driver*, we need to analyze the semantic dependencies between *Compact Car*, *Car*, and *Driver*.

The information that a *Car* is operated by a *Driver* (association relationship) implies a property of the class *Car* (class properties are methods, attributes, and their relationships). The information that *Compact Car* is-a *Car* (inheritance) implies that *Compact Car* inherits all properties from *Car*. It follows that *Compact Car* inherits the association to *Driver* from *Car* which implies that a *Compact Car* is also operated by a *Driver*. This knowledge of the transitive relationship between *Compact Car* and *Driver*, implies that the classifier *Car*, as well as the relations *is-a* and *operated-by*, could be collapsed into a composite, more abstract relationship linking *Compact Car* and *Driver* directly. That composite relation should be of type *association*. This example shows a case where knowledge about the semantic properties

of classifiers and relations allows us to eliminate a helper class and derive a more abstract class diagram.

The above example therefore indicates a class abstraction pattern of the form:

Given: Inheritance -> Class -> Association

Implies: Association

This pattern may be used to collapse any occurrence of the “given” pattern into an occurrence of the “implies” pattern. The second diagram in Figure 48 (middle) depicts a *Guest* who *stays-at* a *Hotel* which has *Rooms*. What the diagram does not depict is the (more abstract) relationship between *Guest* and *Rooms*. Semantically this diagram implies that *Rooms* are part of a *Hotel* which, in turn, implies that the class *Room* is conceptually *within* the class *Hotel*. If, therefore, *Guest* depends on *Hotel*, *Guest* also depends on *all* parts of *Hotel*—including *Rooms* (note that this assumption is weaker in that *Guest* may not actually depend on *all* parts of *Hotel* – we will discuss implications of this later). It follows that *Guest* relates to *Room* in the same manner as *Guest* relates to *Hotel*. We again found an abstraction pattern by analyzing the semantic relationships between the *Hotel*, *Guest*, and *Room* configuration:

Given: Association -> Class <- Aggregation

Implies: Association

Note that the directions of arrows have relevant semantic meanings. If *Hotel* were part of *Room* then we could not automatically assume the correctness of above pattern.

The third example depicted in Figure 48 shows an elevator system where a *User* operates the *Control* (panel) class and depends on the *Door* when to enter/leave the cabin (only if the door state is “open”). The figure hides the more abstract relationship(s) between *User* and *Elevator*. Assuming we have the knowledge that *Cabin*, *Door*, and *Control* are part of *Elevator*, we need to analyze how these three classes could be merged together to form one, more abstract class called *Elevator*. In the case of *Door* and *Control*, both are classes. A combined composite class of the two should, therefore, be another class. Grouping is a conceptually simple operation since it involves just the replacement of a group of classes by a single class. In the case of classes we could, therefore, devise the following simple rule:

Given: Class -> Association -> Class

Implies: Class

Applying the above rule, we find that classes *Cabin*, *Door*, and *Control* become one class. We already know from the second example that, if *User* depends on a part of a class, then *User* also depends

on the composite class. Thus, *User* must also have a *Dependency* and an *Association* relationship to the composite class *Elevator*. As mentioned before, the types of classifier abstractions are relevant. For instance, if we were to abstract two interface classes, then the resulting class will also be an interface class.

7.3.1.3.2 Abstraction Patterns

For a transformation tool to automatically support abstraction, abstraction rules must be provided. Figure 49 shows two simple structures for abstraction rules (the top structure is for the classifier abstraction and the bottom structure is for the relation abstraction). Abstraction rules follow a simple concept. They define an input and an output pattern analogous to the “given” and “implies” pattern we used previously. Furthermore, the output pattern should be simpler (more abstract) than the input pattern, thus, guaranteeing that each applied abstraction rule indeed yields an abstraction.

A classifier abstraction rule should have an input pattern of at least two classifiers with or without a relation between them. The corresponding output pattern should then be at least a single classifier. An example of this kind of abstraction pattern was given in the *Cabin-Door-Control* diagram in Figure 48. There, we found an input pattern representing a structure of two classes with an association relationship between them. The output pattern was a single class.

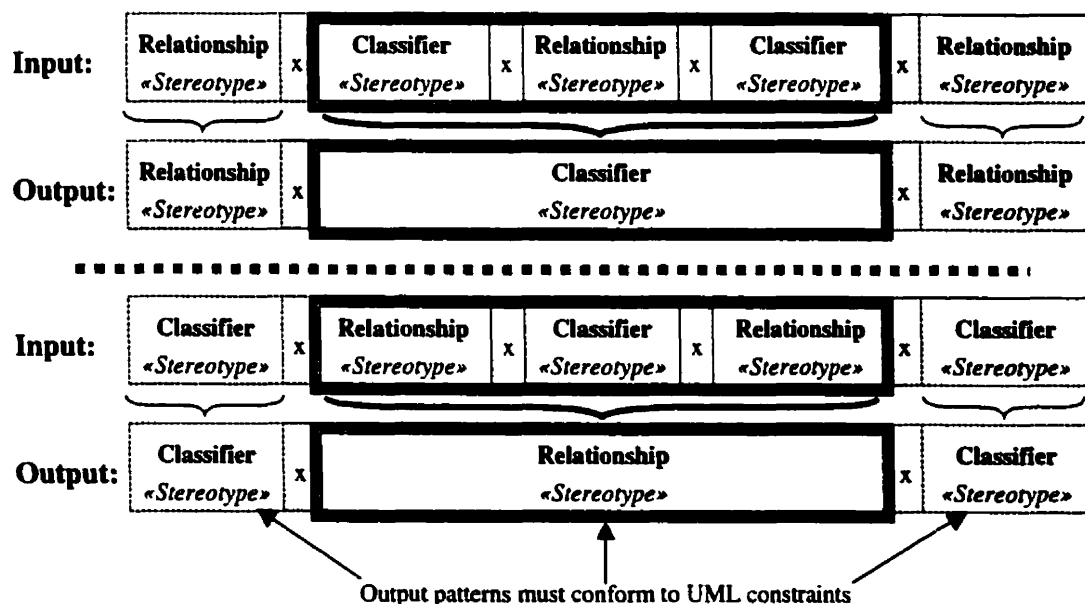


Figure 49: Simple Input/Output Structure Patterns for Abstractions

The lower half of Figure 49 shows the equivalent simple rule structure for the relation abstraction. There the input pattern consists of two relationships connected via a classifier, which can be abstracted into a single relationship. Figure 48 gave two examples for relation abstraction (*Guest-Hotel-Room* and *Driver-Car-Compact-Car*). Like in the classifier abstraction, the input pattern should be more complex than the output pattern. Otherwise, the abstraction algorithm presented later could be non-deterministic. The rules in Figure 48 represent a simple 2(3) to 1 abstraction pattern; however, our mechanism also applies to more complex input and output patterns, as will be discussed later.

7.3.1.3.3 Rules and Reliabilities

Figure 50 shows a list of input and output patterns (rules) for class abstraction rules. The structure used in Figure 50 follows that discussed in Figure 49. The left side depicts the class input patterns and the right side (after “equals”) depicts the class output patterns. Rule 3 in Figure 50 corresponds to the *Compact-Car-Car-Driver* pattern in Figure 48, rule 24 corresponds to the *Guest-Hotel-Room* pattern, and rule 46 corresponds to the *Elevator-User* pattern. We also analyzed the semantic dependencies between other classes and their relationships and, thus, were able to derive 47 additional abstraction rules. Note that the direction of relations is indicated by their name. If the relation name is used with no add-on, then a forward relation (a relation from left to right) is meant. If the string “Reverse” is added then a backward relation (a relation from right to left) is meant.

The number at the end of each rule indicates its reliability. Since patterns and rules are based on semantics, the rules may not always be valid. We use reliability numbers as a form of priority setting to distinguish more reliable rules from less reliable ones. This priority setting is applied when deciding what rules to use when, such that more predictable rules are applied first. The reliability numbers can be between 0 and 100 (100 for high and 0 for low). Since composite model elements are derived through class abstractions rules and since those rules have reliability numbers attached, it follows that the composite model elements inherit the reliability number from the rule(s) from which they were created. For instance, if a composite relation was created through rule 24, then the composite relation has the reliability of 100. If a composite model element itself consists of another composite model element then the reliability numbers are multiplied as factors of 100. E.g., if a very reliable rule (90) was followed by a

(1)	Generalization x Class x Generalization equals Generalization	100
(2)	Generalization x Class x Dependency equals Dependency	100
(3)	Generalization x Class x Association equals Association	100
(4)	Generalization x Class x Aggregation equals Aggregation	100
(5)	Dependency x Class x Generalization equals Dependency	50
(6)	Dependency x Class x Dependency equals Dependency	100
(7)	Dependency x Class x Association equals Association	50
(8)	Dependency x Class x Aggregation equals Dependency	70
(9)	Association x Class x Generalization equals Association	70
(10)	Association x Class x Dependency equals Dependency	50
(11)	Association x Class x Association equals Association	100
(12)	Association x Class x Aggregation equals Association	100
(13)	Aggregation x Class x Generalization equals Aggregation	50
(14)	Aggregation x Class x Dependency equals Dependency	50
(15)	Aggregation x Class x Association equals Association	90
(16)	Aggregation x Class x Aggregation equals Aggregation	100
(17)	Generalization x Class x DependencyReverse equals DependencyReverse	100
(18)	Generalization x Class x AggregationReverse equals AggregationReverse	100
(19)	Dependency x Class x GeneralizationReverse equals Dependency	100
(20)	Dependency x Class x AggregationReverse equals Dependency	80
(21)	Association x Class x GeneralizationReverse equals Association	100
(22)	Association x Class x DependencyReverse equals DependencyReverse	50
(23)	Association x Class x AggregationReverse equals Association	70
(24)	Aggregation x Class x GeneralizationReverse equals Aggregation	100
(25)	Aggregation x Class x DependencyReverse equals DependencyReverse	80
(26)	GeneralizationReverse x Class x Dependency equals Dependency	50
(27)	GeneralizationReverse x Class x Association equals Association	70
(28)	GeneralizationReverse x Class x Aggregation equals Aggregation	80
(29)	DependencyReverse x Class x Generalization equals DependencyReverse	50
(30)	DependencyReverse x Class x Aggregation equals DependencyReverse	100
(31)	DependencyReverse x Class x Association equals DependencyReverse	50
(32)	AggregationReverse x Class x Generalization equals AggregationReverse	80
(33)	AggregationReverse x Class x Dependency equals Dependency	100
(34)	AggregationReverse x Class x Association equals Association	100
(35)	GeneralizationRev x Class x GeneralizationRev equals GeneralizationRev	100
(36)	GeneralizationReverse x Class x DependencyRev equals DependencyRev	50
(37)	GeneralizationReverse x Class x AggregationRev equals AggregationRev	50
(38)	DependencyReverse x Class x GeneralizationRev equals DependencyRev	100
(39)	DependencyReverse x Class x DependencyRev equals DependencyRev	100
(40)	DependencyReverse x Class x AggregationRev equals DependencyRev	50
(41)	AggregationReverse x Class x GeneralizationRev equals AggregationRev	100
(42)	AggregationReverse x Class x DependencyRev equals DependencyRev	70
(43)	AggregationReverse x Class x AggregationRev equals AggregationRev	100
(44)	Class x Generalization x Class equals Class	99
(45)	Class x Dependency x Class equals Class	99
(46)	Class x Association x Class equals Class	99
(47)	Class x Aggregation x Class equals Class	99
(48)	Class x GeneralizationReverse x Class equals Class	99
(49)	Class x DependencyReverse x Class equals Class	99
(50)	Class x AggregationReverse x Class equals Class	99

Figure 50: Abstraction Rules for Class/Object Diagrams

less reliable rule (50) then the overall reliability of the resulting composite model element is $90 * 50 / 100 = 45$. The heuristics were derived through experimentation with dozens of models (some of which provided by industry). The heuristics should, however, not be interpreted as fixed. Domain- or company-specific needs may well require their adaptation.

7.3.1.4 Complex Abstraction

Thus far, we discussed the basics of our abstraction approach. This section will discuss extensions to cover more complex types of abstraction issues.

7.3.1.4.1 Serial Abstractions

Serial abstractions were already implied previously when we talked about the possibility of applying multiple abstraction rules in a sequence. Figure 51 illustrates such a case. There, at the upper left, an association relationship *between Person and Car* is described. It is stated that the *Person inspects* the *Car*, that *Mechanic is-a Person* and that *Volkswagen is-a Car* (note that the *is-a* relationship denotes inheritance).

If it is of interest to know the more abstract relationship between *Mechanic* and *Volkswagen*, then our abstraction process can be applied in sequence to eliminate both helper classes *Person* and *Car*. For instance, rule 2 could be used to eliminate *Person* and replace it with a simple *association* relationship (upper-right). Alternatively, rule 17 could be used to eliminate *Car* (lower-left). In both cases, we are left with less complex, more abstract three-class configurations. By applying our abstraction

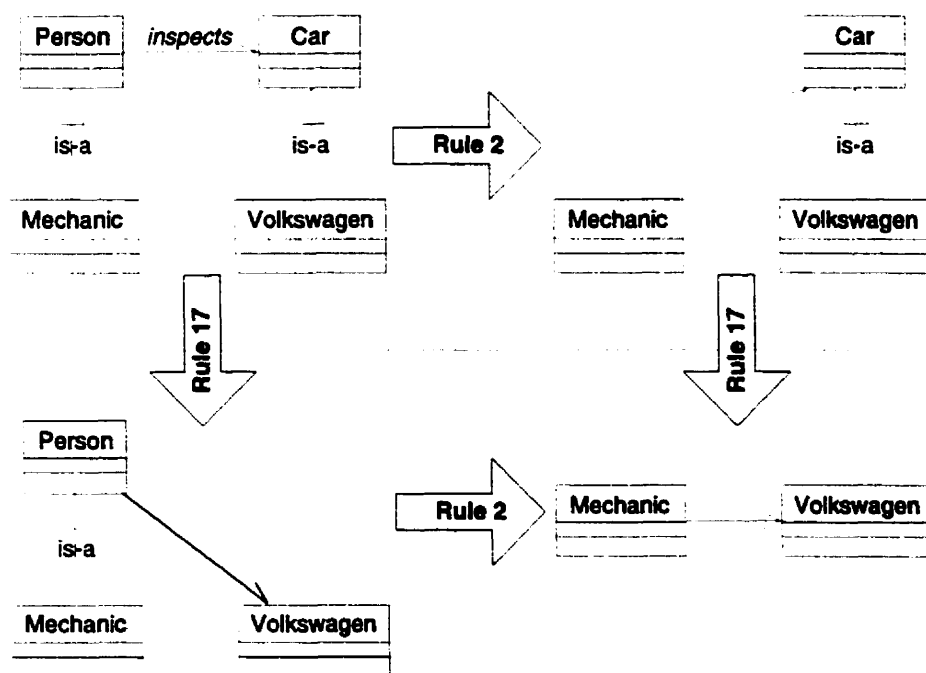


Figure 51. Serial Abstraction

rules on the already partially abstracted diagrams, we can further abstract that configuration. For instance, if we had chosen rule 2 above, then rule 17 could be applied next. Similarly, if we had chosen rule 17 initially, then rule 2 could be used next. In both cases, the resulting abstraction results in an (composite) association going from *Mechanic* to *Volkswagen*. Since these abstraction rules (2 and 7) were equally reliable (we used the same rules, only in a different order) and resulted in the same composite type (an association), both results can be merged together into a single association.

Serial abstraction therefore enables more complex abstraction tasks where a larger number of classes can be eliminated. Generally, all abstraction paths need to be explored. In case an abstraction path yields a different result than another abstraction path (between the same two classes), both should be considered valid unless one path yields a significantly more reliable result than the other one.

7.3.1.4.2 Complex Rules

All the rules presented in Figure 50 are of the simple three-class to two-class pattern. Those patterns could be made more complex as well. For instance, the example in Figure 51 could be transformed into a more complex abstraction rule of the form:

Given: Generalization -> Class -> Aggregation -> Class <- Generalization
Implies: Association

It follows that complex abstraction rules can be used in place of serial abstractions. The advantages of using more complex abstraction patterns are better reliability and better coverage. The reliability is improved since ambiguous abstraction paths and, thus, ambiguous abstraction results are contained. For instance, in Figure 51 we found two abstraction paths. With a complex abstraction rule like the one above, that ambiguity is eliminated and only a single abstraction is found. Complex abstraction rules also improve the abstraction coverage (the situations they are applicable to) since those

```
State x Link x State equals State 100
State x LinkReverse x State equals State 100
StartState x Link x State equals StartState 50
State x LinkReverse x StateState equals StartState 50
State x Link x EndState equals EndState 50
EndState x LinkReverse x State equals EndState 50
Link x State x Link equals Link 100
LinkReverse x State x LinkReverse equals LinkReverse 100
```

Figure 52: Abstraction Rules for State Diagrams

Package x Dependency x Package equals Package 100
Package x Generalization x Package equals Package 100
Package x DependencyReverse x Package equals Package 100
Package x GeneralizationReverse x Package equals Package 100
Dependency x Package x Dependency equals Dependency 100
Dependency x Package x Generalization equals Generalization 50
Generalization x Package x Generalization equal Generalization 100
Generalization x Package x Dependency equals Dependency 100

Figure 53: Abstraction Rules for Package Diagrams

complex rules tend to be more specialized. For instance, there are situations where more specialized rules (e.g., domain specific rules) can be created and validated more easily than more general rules. The following will discuss that briefly.

7.3.1.4.3 Specific Rules

The current set of rules listed in Figure 50 cover the generic semantic relationships within UML class diagrams. More specialized rules can be generated based on domain specific knowledge. Also, currently only *association*, *generalization*, *dependency*, and *aggregation* relationships are supported. Additional information, such as stereotypes or tagged values, could be used to further specialize those rules. This information may be used 1) to refine the meaning of rules and 2) to extend the rule set to another set of rules. Furthermore, both abstraction methods can be generalized onto other types of diagrams. Figure 52 and Figure 53 depict abstraction rules for package and state diagrams respectively. Again, those rules could be specialized if needed.

7.3.1.5 Abstraction Algorithm

Figure 54 depicts the basics of our abstraction algorithm. As input, a concrete model (e.g., class diagram) needs to be provided. Furthermore, it must be specified which model elements should be

1. input: concrete model elements and collection of classifiers/relations
2. find all paths between each classifier/relation pair
3. for each path
abstract path by recursively applying abstraction rules until:
- a composite model element has been found,
- no abstraction rule can be applied, or
- the reliability of composite model element becomes too small
(reuse existing composite model elements if applicable)
4. for all abstractions corresponding to a single classifier/relation pair
eliminate less reliable results or duplicate results of the same type

Figure 54. Abstraction Algorithm

abstracted. All possible paths between the selected model elements are then identified. Each path is abstracted individually using the rules from Figure 50. The abstraction process terminates if (1) a suitable abstraction is found, (2) no more abstraction rules can be applied, or (3) the reliability number of the composite model element becomes too small (too unreliable). Note that the abstraction process has to be applied repeatedly for every model element permutation of the input collection. The resulting complexity of the algorithm ($O(n^2)$) is, however, reduced by reusing intermediate composite model elements that have been abstracted previously. Finally, at the end, less reliable paths as well as duplicate paths are eliminated. For instance, if two different abstraction paths between the same two classes yield the same types of abstractions, then they can be merged together.

7.3.1.6 Specialized Abstraction

Our generic abstraction mechanism works for box-and-arrow types of diagrams. Abstracting only boxes and arrows, however, only covers



Figure 55. Cardinality Examples between Classes

some features that diagrams incorporate. Figure 55, depicts a class diagram showing the relationships between *Hotel*, *Guest*, and *Room* again. Additionally, the figure depicts the cardinality between those classes. For example, a *Guest* may stay at one or many *Hotels* and a *Hotel* may have zero to many *Guests*. Also, a *Hotel* may have one to many *Rooms* and each *Room* belongs to one *Hotel* (the diamond at the end of that line shows a part-of relationship that has cardinality one unless defined otherwise). Cardinality issues are specific to class diagrams in UML. Its abstraction requires additional measures that do not apply to package or state diagrams.

Abstracting cardinality must, therefore, be treated separately. Otherwise, the process of abstracting *Hotel*, *Guest*, and *Room*, loses the cardinality between *Guest* and *Room*. How this can be avoided is depicted in Figure 56 where various cardinality scenarios and their abstractions are shown. The first scenario (a) indicates that for each classifier A there is exactly one classifier B and, similarly, for each classifier B there is exactly one classifier C. It follows that for each classifier A there must be

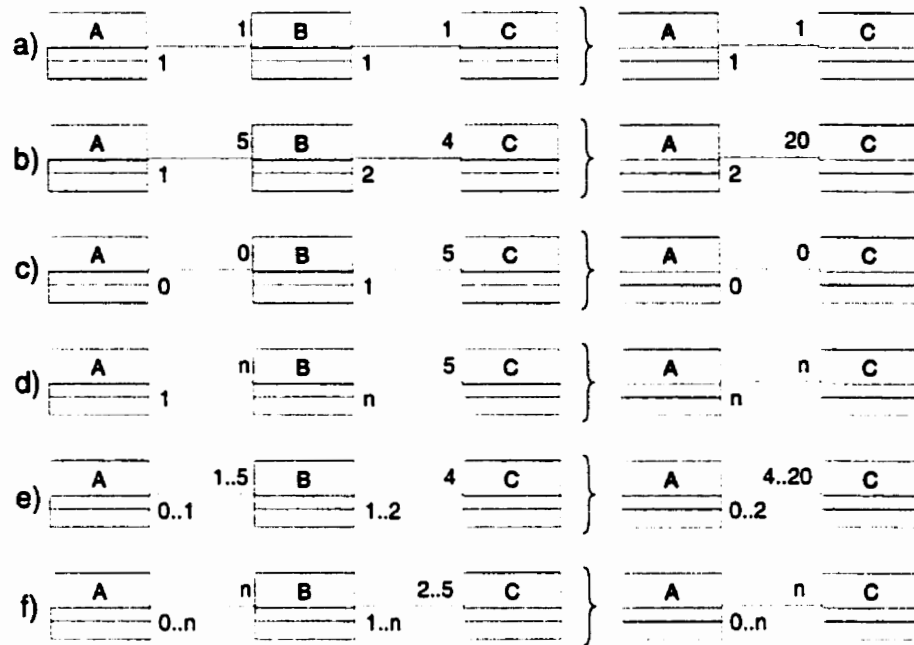


Figure 56. Cardinality Examples and their Abstractions

exactly one classifier C. Going through examples "a" to "d," we find that we can derive abstracted cardinalities by multiplying individual cardinalities associated with the same direction.

Cardinalities may also have ranges as depicted in examples "e" and "f" in Figure 56. In the case of ranges, the minimum values must be multiplied to yield the abstract minimum range and the maximum values must be multiplied to yield the abstract maximum range. In case a finite value (e.g., one, two, etc.) is multiplied with an infinite one (e.g., *, n, many, etc.), the result will always be infinite (the exception is if the finite value is zero, in which case the result will be zero). In case a value is multiplied with a range (example "e"), the value must be multiplied with the lower and upper bounds individually.

For our abstraction algorithm to also address cardinality issues, we have to multiply the cardinality numbers whenever two associations or aggregations are abstracted. Other abstraction issues related to other types of diagrams must be addressed similarly on an individual basis if applicable.

7.3.1.7 Example

Figure 57 gives a simple example of how abstraction rules are applied to generate a simpler, more abstract class diagram from a collection of three diagrams. Class diagram 1 shows the relationships

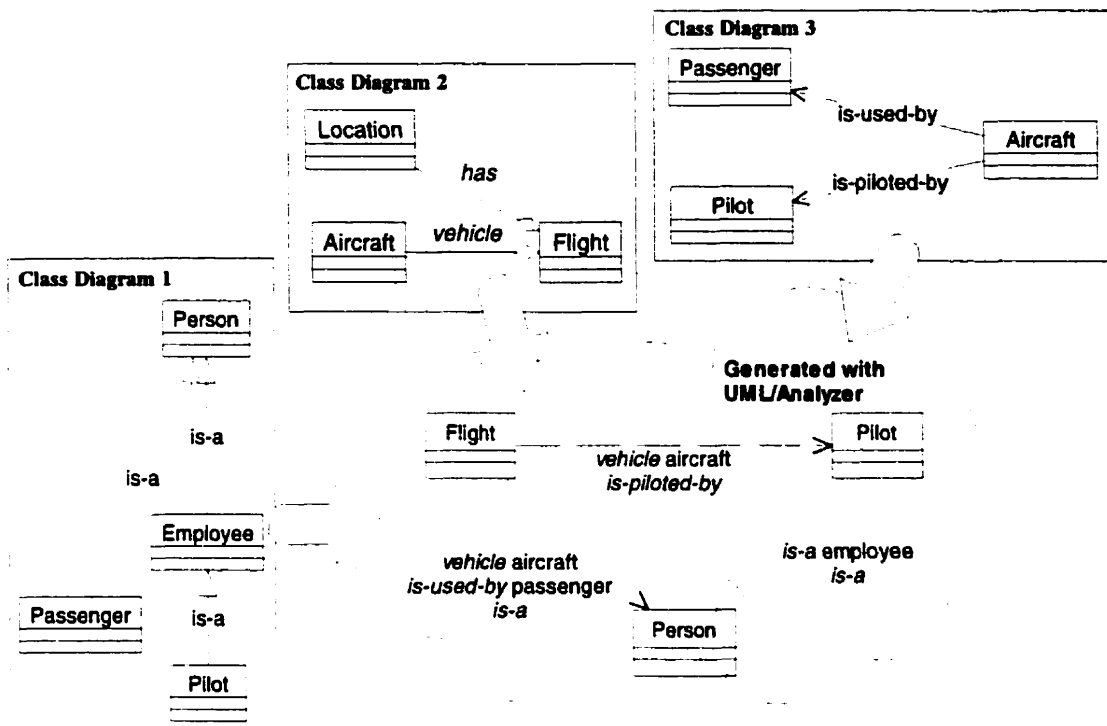


Figure 57. Simple Example of Generated Abstractions from three Input Diagrams

between people in a simplified Air Traffic Control system. It depicts a parent class *Person* for the main actors *Pilot* and *Passenger* (generalization connectors are used). Diagram 2 describes a *Flight*, which has a *Location* at any given time and which uses an *Aircraft* as a vehicle (aggregation connectors are used). Diagram 3 shows the relationship of the people and the *Aircraft* (dependency connectors are used).

Using these three diagrams as input to our abstraction algorithm, we can generate a simpler model that only shows the relationships between *Person*, *Pilot*, and *Flight*. Since these components are not connected directly to one another in the input class diagrams, our abstraction process can derive their transitive relationships using the rules defined in Figure 50. The relation names in Figure 58 are preceded by “«derived»” to indicate that they were generated automatically.

Between *Flight* and *Person*, there is a transitive relationship from *Flight* to *Aircraft* to *Passenger* and finally to *Person*. Figure 58 shows the abstraction process. Both *Aircraft* and *Passenger* can be eliminated by applying rules from Figure 50. Note that the process looks slightly different if *Passenger* is

eliminated first (the rules would be applied in reverse order). In this example, the result is the same, but this may not always be the case.

The relation abstraction process outlined above was adopted by Rational Software and implemented in a tool called Rose/Architect (by Ensemble Systems for Rational Software [Egyed and Kruchten 1999]). As part of this work, we also implemented our own version of classifier and relation abstraction into a tool called UML/Analyzer. Our version incorporates all features of Rose/Architect, and also addresses repository issues, scalability issues, reliabilities, classifier abstraction, more complex abstraction patterns, and C2 architectural style support. Our UML/Analyzer tool is discussed in Section 8.

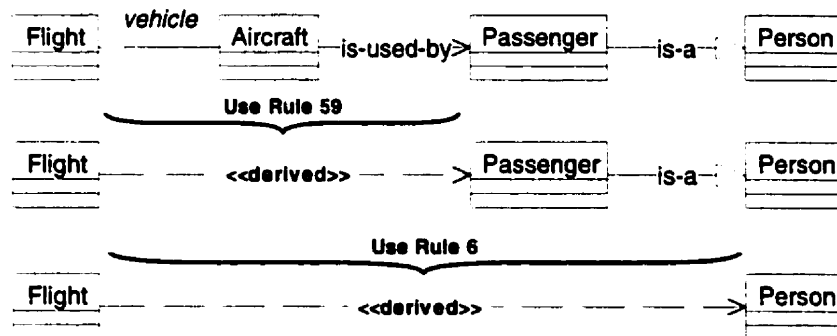


Figure 58: Generating transitive relationship from Flight to Person

7.3.2 Generalization

Generalization aggregates specific information into a more generic form. For instance, a test scenario depicts one very specific form of interaction. Since it is usually not possible to generalize from single scenarios (or single examples), generalization takes multiple scenarios and unites them into more generic scenarios (e.g., sequence diagram) or even generic diagrams (e.g., class diagram). UML uses sequence, collaboration and object diagrams to describe specific issues. To generalize those into more generic diagrams (e.g., class and state diagrams), we have adopted and extended the sequence-to-state transformation method from Koskimies et. al [Koskimies et al. 1998] as well as object-to-class transformation from Ehrig et al. [Ehrig et al. 1997].

Generalization techniques for various UML views are similar in that they have to take multiple sources of input and merge them together to infer more generic knowledge. For instance, in the case of

object to class generalization, an object diagram shows specific examples of how objects access other objects. By merging all information about any two objects' interactions, we can infer the general interaction of their types—the classes. Likewise, Koskimies' [Koskimies et al. 1998] sequence to state generalization takes multiple sequence diagrams as input and merges them by interpreting calls/returns as state changes. Koskimies' approach has the downside that state changes do not always correlate one-to-one with procedure calls. Thus, we improved their approach by also relating calls to class methods, which in turn specify whether or not they alter the state. Using this additional information, the sequence to state consolidation becomes more reliable. Currently some tool support is available to automate generalization (e.g., Koskimies' SCED tool [Koskimies et al. 1998] automating sequence to state transformation).

7.3.2.1 Sequence to Statechart Generalization

At least two groups of researchers have been developing ways for generalizing sequence views into statechart views ([Schönberger et al. 1999] and [Koskimies et al. 1998]). Their approaches are very similar and both are discussed in related works in Section 10. Basically, both make the assumption that state changes as depicted in statechart views are triggered by method calls. Methods are operations of classes that can access and modify attributes (e.g., variables) of their own classes or other dependent classes. Sequence diagrams describe when and in what order methods are called. Similarly, a statechart depicts in what order a class state can change.

Both generalization approaches have drawbacks and the reasons for those are simple. Sequence views contain some statechart-relevant information, but not all of it. For instance, the information whether a method call in a sequence diagram causes (or does not cause) a state change in a state diagram is ambiguous. Thus, it is difficult and sometimes impossible to generate a complete statechart out of sequence diagrams. We extended Koskimies' approach in the following:

- Use of class information to infer method type (query, action, activity): We use that information to improve reasoning on whether a method call causes a state change (e.g., yes if action; no if query).
- Use of “return” links to differentiate between end-of-methods and new method calls: We use this information to qualify sequence links (e.g., to add semantic meaning to arrows).

Figure 59 shows an example of transforming two sequence diagrams (top half) into statechart diagrams (lower half). The sequence diagrams depict the interaction between a generic *File* class and a domain specific *GuestAccess* class. *File* is used to store guest information. *GuestAccess*, in turn, is used to provide a nicer interface for *File*. The first sequence diagram depicts a scenario where *guest* (instance of *GuestAccess*) constructs a new object called *f1* (instance of type *File*) and then *opens* the file. The second sequence diagram depicts a scenario where *guest* reads from *f2* (of type *File*) and after the read operation is finished (after return) adds some data to *f2*. The sequence diagrams in Figure 59, although only scenarios, allow one to infer a lot of generalizable information.

- (1) If a method (e.g., *construct*) creates a new object, then this method must be a state transition initiating from a start state
- (2) If a method (e.g., *open*) is a regular message link, then this method must be either an activity, action, or query (no state change, regular state change, or circular state change).
- (3) If a method (e.g., *read*) has to be completed before another method (e.g., *add*) can be started then those methods cannot lead into parallel states.

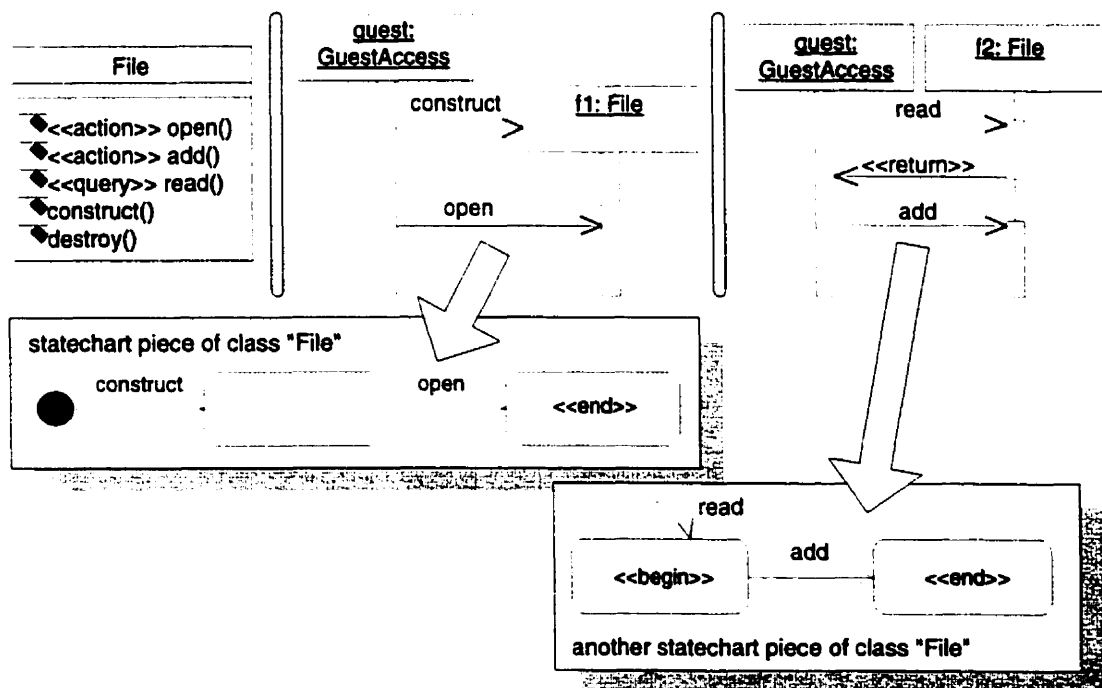


Figure 59. Sequence to State Generalization—Basics

Given sequence diagram information, we could not infer of what type *open* and *read* are about. They could be either circular state transitions, regular transitions between states, or activities as part of a state. To reason about those ambiguous properties, Figure 59 additionally shows a specification of class *File*. There we can tell that *open* and *add* are actions and that *read* is a query. Using that information, we can derive two statechart diagrams. The first one (depicted in the lower-left of Figure 59) shows the *construct* method initiating out of a start state. Since *open* is an action, it follows that it causes a state transition between two states. The stereotype <<end>> attached to one of the states indicates that no additional information is available to reason about what is going on thereafter. Since nothing can happen before a start state (from an object's perspective), this case requires no explicit handling. The second state diagram in the lower-right of Figure 59 shows two states and two state transitions. The *read* state transition is circular since the *read* method is of type query and cannot change a state. The *add* state transition, however, bridges between two distinct states since it is of type "action." Since sequence diagrams do not describe the relationships among themselves, we cannot yet reason about how the first state diagram fits with the second one. The second derived state diagram, therefore, uses the stereotypes <<begin>> and <<end>> to indicate that other things can happen before or after those states. For instance, it is possible that the <<end>> state of the first state diagram is identical to the <<begin>> state of the second. Figure 60 depicts another set of sequence diagrams and their corresponding statechart diagrams. Two additional rules can be observed:

- (4) If a method (e.g., *destroy*) removes an object, then that method must be a state transition terminating in an end state
- (5) If a method (e.g., *delete*) was not declared, then assume it to be a state transition

Combining the statechart diagrams in Figure 59 and Figure 60, generalization yields five interpretations (all about states of the class *File*). Although those interpretations are separate from each other, it is nevertheless possible to combine them to yield a more compact state diagram. Statecharts can be combined via common state transitions that normally indicate common states. For instance, Figure 60 depicted two similar state transitions; one going via *open*, *add*, *delete*, and *close*; the other omitting

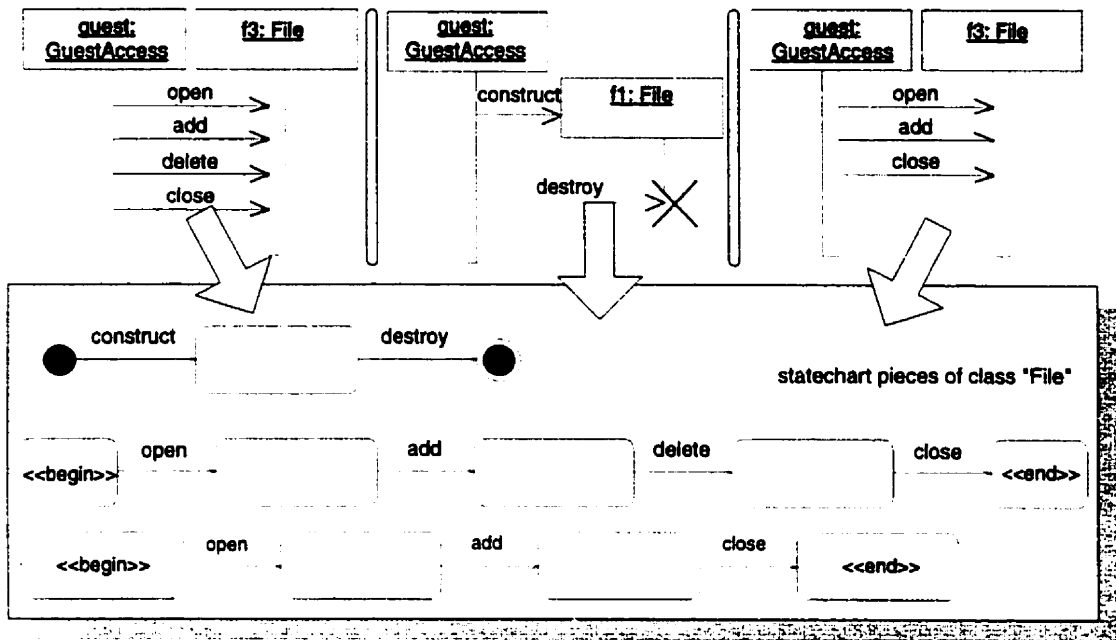


Figure 60. Sequence to State Generalization-Extended

delete. We can therefore infer that the method *delete* may or may not follow the method *add*. Combined with Figure 59, it can be seen that either the method *open* or the method *destroy* may follow *construct*.

Combining all those pieces of information, we find a more compact-minimal-statechart diagram (see Figure 61). Note that transformation for synthesis and transformation for analysis have distinct goals. For instance, in transformation for synthesis, a minimal statechart diagram is much more beneficial, whereas for view integration (analysis) bits and pieces are more beneficial. The latter is the case since the

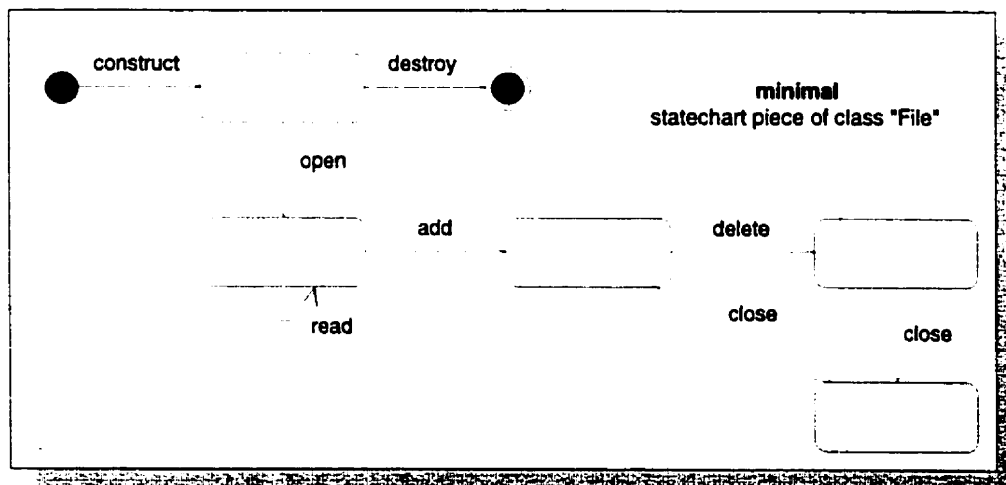


Figure 61. Minimal Statechart Diagram

minimized version makes the assumption that sequence diagrams cover all relevant usages. Since it is possible (even likely) that not all behavioral variations have been modeled as scenarios, the above assumption may not be valid in these cases.

7.3.2.2 Object to Class Generalization

An object diagram depicts instances of classes and how they interact. As such, an object diagram represents a scenario of one possible instantiation of a class diagram. Since object diagrams are in structure similar to class diagrams (in fact, UML uses the same view type to represent both), it seems natural to use class diagrams for generalization purposes. Figure 62 depicts a simple object diagram for

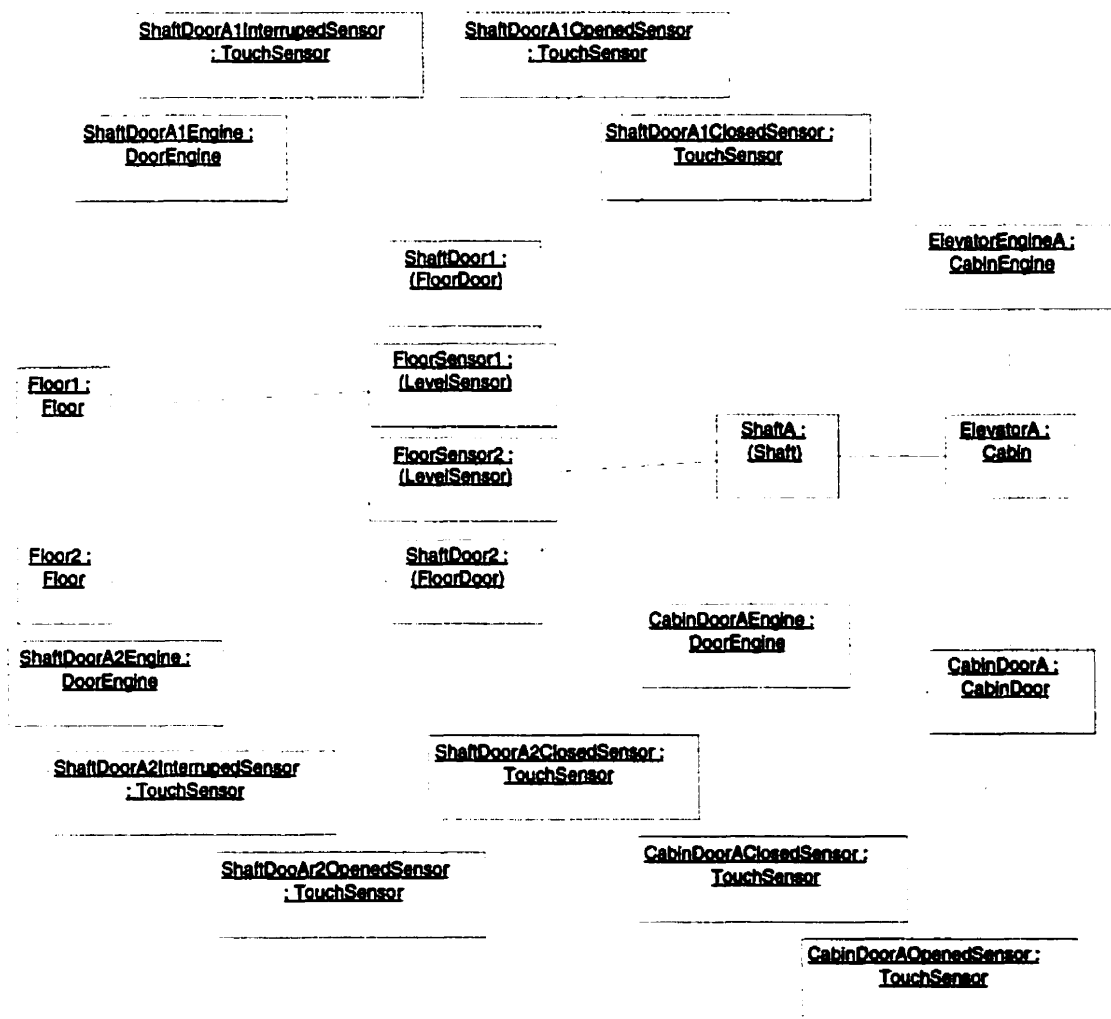


Figure 62. Object Diagram

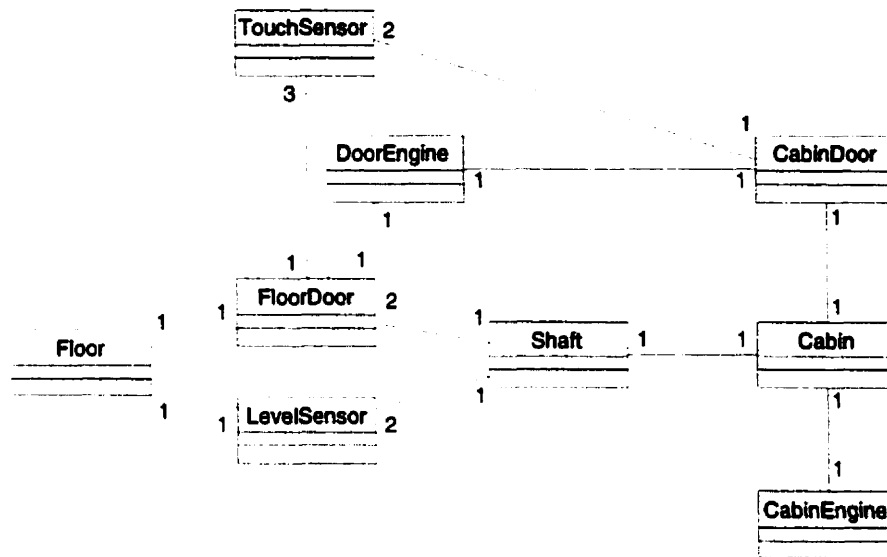


Figure 63. Generalized Object Diagram Represented as Class Diagram

an elevator system. The diagram depicts a very trivial realization of such a system where only one shaft with one cabin exists and the elevator can only go between two floors. For the elevator control, sensors are attached to floors, shafts, and doors, indicating the positioning of the cabin relative to the floors and the state of the door (open or closed).

Generalizing an object diagram is trivial because the semantics of boxes and lines are the same as in class diagrams. Figure 63 depicts the generalization of Figure 62. If an object does not indicate its type (its class), a dummy “class” with unknown type must be created. The generalization method additionally keeps track of how many instances of the same type were attached to the same instance of another type. For instance, *shaftA* has exactly one *cabinA*. Thus, there is a one-to-one relationship between *Shaft* and *Cabin*. Further, there are two instances of *FloorDoor* (called *ShaftDoor1* and *ShaftDoor2*). Each instance of *FloorDoor* has exactly one instance of *DoorEngine* and three instances of *TouchSensor* attached. Thus, there is a one-to-one relationship between *FloorDoor* and *DoorEngine* and a one-to-three relationship between *FloorDoor* and *TouchSensor*.

As with sequence diagrams, multiple object diagrams yield multiple class diagrams. Those class diagrams can then be merged together to yield a minimal class diagram. Unlike with statecharts, minimizing class diagrams has no risks attached and can be done by default since object and class

diagrams are more structural than sequence and state diagrams. Object view to class view generalization exhibits the following situations (patterns):

- (1) If there is a link between objects, then there must also be a link between their corresponding classes.
- (2) If there is more than one object of the same type attached to another object (of a different type), then this denotes cardinality. The lower and upper bounds of that cardinality are derived from the minimum and maximum numbers observed.
- (3) If objects of the same type are attached to one another, then there must be a circular link between their corresponding classes.

7.3.2.3 Generalization Rules and Automation

The generalization scenarios described in this section are very distinct. They have, however, two properties in common. First, they transform instances into types (e.g., calls into state transitions or objects into classes). Second, they combine information from multiple instances to yield more generic results. The information used for generalization are the interdependencies between classifiers (see Figure 64). Currently, we have adopted a third party tool called SCED [Koskimies et al. 1998] for sequence to state generalization. We have no tool support for object to class generalization.

7.3.3 Structuralization

Structuralization takes information about behavior to infer structure. For instance, a test scenario depicts interactions between objects. Since it is usually not possible to infer behavior out of structure,

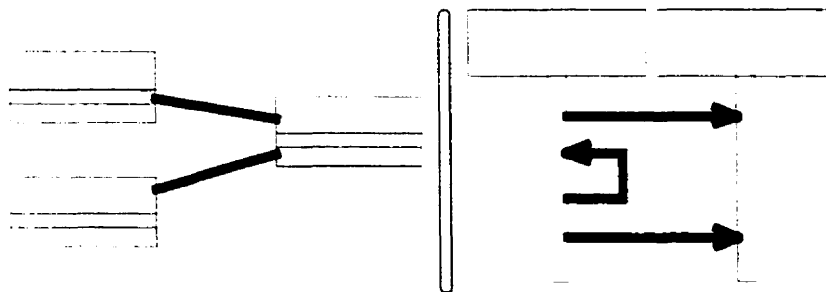


Figure 64. Generalization Patterns

structuralization is uni-directional from behavior to structure (ergo structuralization). UML uses sequence and statechart diagrams to describe behavior. To structuralize them, we create object and class diagrams. In particular, sequence diagrams get structuralized into object diagrams and statechart diagrams get structuralized into class diagrams (recall, our discussion in Sections 5.4 and 5.5 as to why we chose these types of views and transformations).

7.3.3.1 Sequence to Object Structuralization

A sequence diagram depicts the interactions between multiple objects. For structuralization, our primary interests are what objects interact and what the directions of their interactions are. For structuralization it does not matter when the interaction happens (as opposed to generalization). There is no distinction between whether interactions occur frequently over a short period of time or only once during a lifespan of an object. In both cases, the objects interact.

Figure 65 depicts a complex sequence diagram. The figure describes the interactions between a reservation clerk and a hotel reservation system and the scenario in particular depicts a regular reservation process: the reservation clerk initiates a *make_reservation* activity with the reservation application (*ReservationApp*). The latter causes service objects and user interface objects to be called. The details of the interaction are only of secondary importance. For sequence structuralization, only the existence and direction of interactions matter. For instance, only the object *ReservationApp* calls the object *ReservationHandler*. *ReservationHandler*, in turn, calls a number of other objects (instances of *Hotel*, *Guest*, *Reservation*, and *Transaction*). Note that only the type of objects (their classes) were specified in the diagram, but not their names.

Figure 66 depicts the structuralized view of Figure 65. The figure shows the same objects, their interactions, and the methods used. This particular example only represented one-to-one relationships. If objects of different types would interact with the same object then one could also record what methods are actually used by what types. For instance, if *Hotel* would also access *Guest* via the method *find*, then one could distinguish between methods used by *Hotel* and methods used by *Reservation Handler*. That information has no immediate use for view integration for UML; however, it could improve the

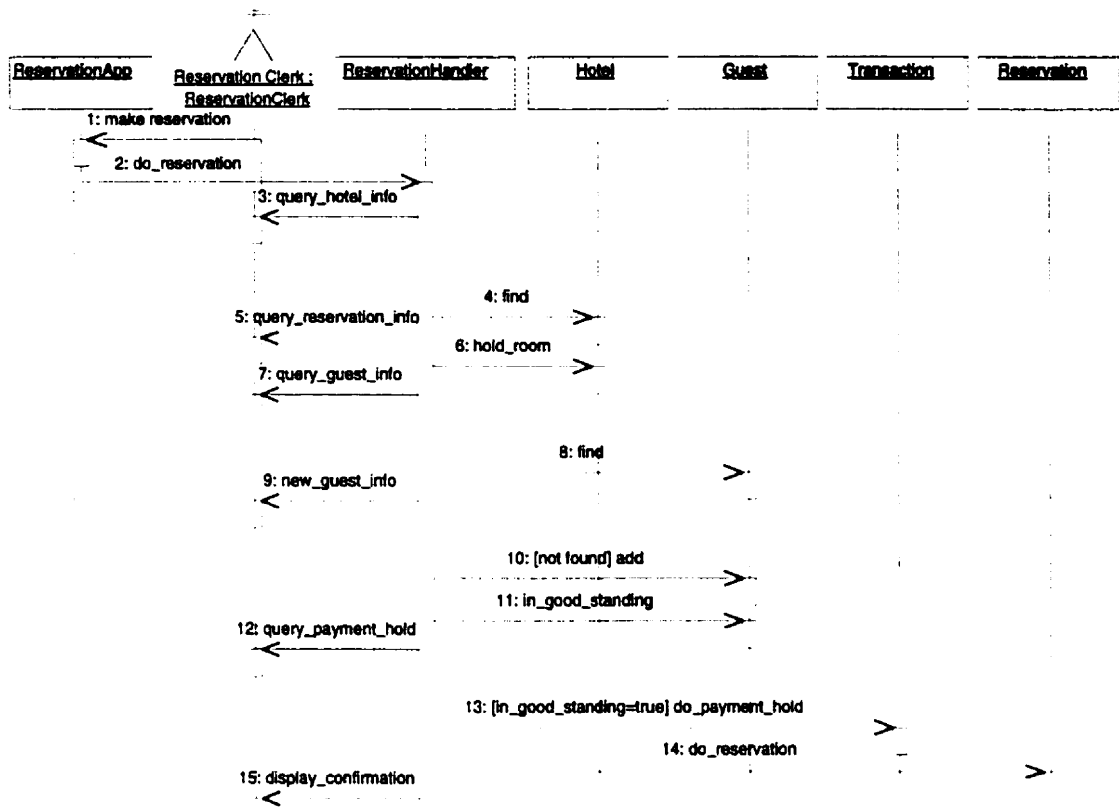


Figure 65. Sequence Diagram

understanding of objects, classes and their interrelationships. The sequence diagram therefore incorporates a series of structuralizable information. The following lists some common patterns:

- (1) If there is a link from one object's bar (vertical pole) to another object's bar, then this denotes an association between those two objects (direction of association is equal to direction of link).
- (2) If there is a link from one object's bar to another object (its box on top of the bar), then this denotes an association with the added information that the link method is a constructor (the method created another object).
- (3) If there is a link from one object's bar to another object's termination (cross at the end of an object's life), then this denotes an association with the added information that the link method is a destructor (the method destroyed another object).

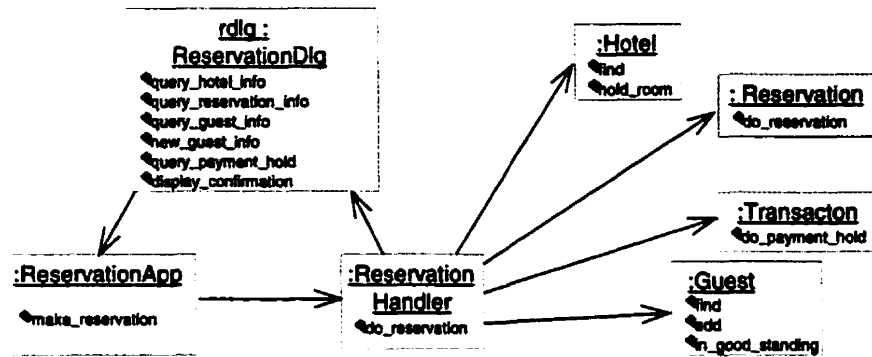


Figure 66. Sequence Diagram Structuralized into an Object Diagram

7.3.3.2 Statecharts to Class Structuralization

Statechart diagrams depict the generic behavior of classes. UML requires states to belong to single classes. For structuralization, our interest is how statechart diagrams of different classes interact. Figure 67 shows a class diagram with two classes called *Cabin* and *Door* as part of our elevator system. Each class has a statechart diagram attached. The statechart view for *Cabin* indicates that the cabin may be either stopped, moving down, or moving up. Similarly, the statechart view for *Door* indicates that a door may be opened or closed, or somewhere in between opening or closing. The example in Figure 67 also depicts one interesting relationship between the two statecharts. The statechart view of *Cabin* describes that a cabin can only start moving once the state of its *Door* is *closed*. The implication of this is that cabin has to be aware of door and, thus, cabin has to be able to access the state of door. For structuralization, we learn that there is a potential association from *Cabin* to *Door*.

This example shows an interesting transformation variation. So far, it was possible to transform views by transforming their boxes and arrows. Although both statechart and class views use boxes and arrows, it is not the boxes or arrows that describe relevant information for structuralization. Instead, here textual annotations in form of events, actions, and triggers are used. Those textual annotations are nevertheless defined model elements in UML. We can observe the following situations:

- (1) If a transition of one statechart is tied to a state of another, then this indicates an association between the classes to which those statecharts belong.

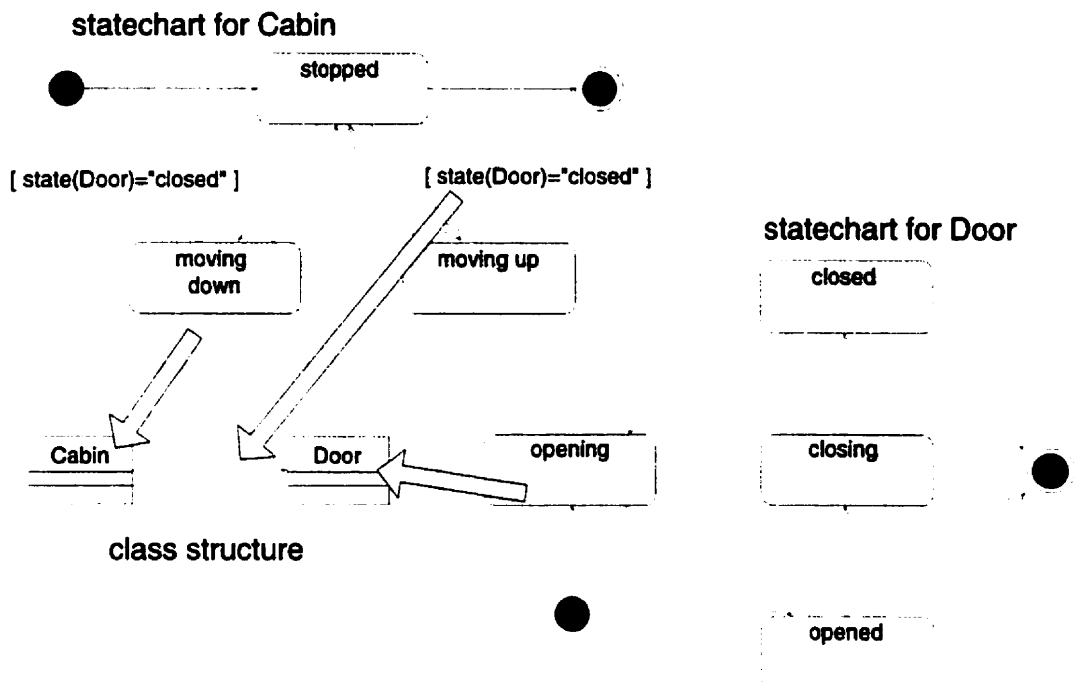


Figure 67. Structuralizing Statechart views into Class Views

- (2) If there is a transition in one statechart view that is triggered by a transition in another statechart view, then this indicates an association (in either direction) between the classes to which those statecharts belong.
- (3) If there is a state in one diagram that is tied to a transition in another diagram, then this may indicate an association between their corresponding classes.
- (4) If there is a state in one diagram that is tied to a state in another diagram, then this may indicate an association between their corresponding classes.

7.3.3.3 Structuralization Rules and Automation

Both structuralization examples exhibited similarities since in both cases it is knowledge about the interrelationships between two groups of data that helps infer structure. In case of sequence to object structuralization, it is the links between the T-like graphical items in sequence diagrams that denote interdependencies (see left side of Figure 68). Similarly, in the case of state to class structuralization, it is the links between statecharts belonging to separate classes that denote interdependencies (see right side of

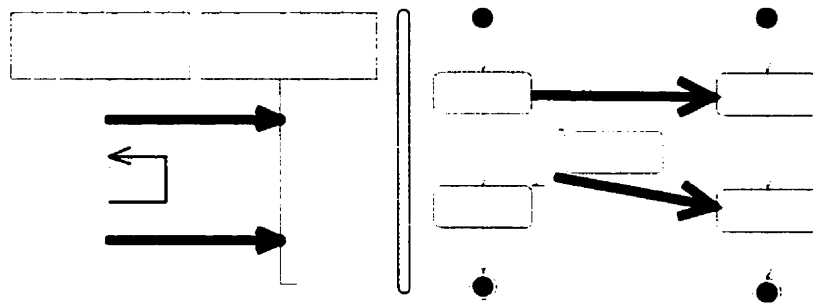


Figure 68. Structuralization Patterns

Figure 68). Note that in the latter case, there must not always be graphical elements relating to interdependencies. Instead, textual annotations could be used. The information used for structuralization are the interdependencies between elements belonging to different classifiers (see Figure 68).

7.3.4 Translation

Translation is the fourth transformation category discussed in this thesis. Translation handles conversion of modeling information without altering its level of abstraction, behaviorism, or generality. For instance, in UML there are two types of views to model specific behavior: sequence diagrams and collaboration diagrams. Instead of having to provide abstraction, structuralization, and generalization techniques for both sequence and collaboration diagrams, we could instead provide a translation from, e.g., collaboration diagrams to sequence diagrams and abstraction, generalization, and structuralization on sequence diagrams only. Thus, translation can minimize the effort required to realize additional transformation methods.

Translation also enables the switching of models to continue the transformation sequence. To revisit Figure 43, we see another example where transformation would be very helpful. In scenario d) we discussed the case of having to transform both views into a third common view to enable comparison. Of course, this third view could be any graphical view discussed before; however, it could also be some formal constraint language underneath the graphical notation. Applied to UML, we could choose the Object Constraint Language (OCL) [Warmer and Kleppe 1999] as an alternative view for describing and comparing modeling information. Thus, OCL could be used to represent the 'something like C' box in Figure 43 (d). Translation is then needed to transform modeling information from UML into OCL.

Another scenario in which translation becomes important is when it comes to replacing or substituting existing views with new types of views. For instance, in the case of UML, we might find the class view not always the ideal view for representing generic and abstract concepts. Instead, we might be tempted to choose a type of view outside our defined modeling environment (e.g., UML in our case). For example, if we wish to build a software system consisting of dynamic and concurrent components we could choose the architectural style C2 [Taylor et al. 1996]. In [Abi-Antoun and Medvidovic 1999] a translation technique is shown for converting C2 modeling information into UML. The issue of translation is not further explored in this thesis.

7.4 Complex Transformation

The comparison of different types of views is greatly simplified through the four types of transformations discussed before. However, on a grander scale, these techniques also have to be integrated with one another to ensure continuity and scalability. This section therefore addresses critical aspects of complex transformation—that of finding transformation paths. We refer to the integration and serial execution of simple transformations as complex transformations.

In simple terms, complex transformation is needed whenever no simple transformation exists. Figure 69 summarizes the complete list of simple transformation paths currently supported through our framework. Technically, the figure should depict a fully connected graph where each view (box) is connected to every other view (box). Since we are supporting nine categories of views (behavior versus structure; instance versus type; and concrete versus abstract), it follows that we would need 36 transformation methods. Currently we only support 14. Despite the partial coverage of needed transformation methods, we can still compare *all* views with one another. The solution is in the integration and serial execution of multiple transformation methods.

Figure 70 depicts examples of how a complex transformation bridges a concrete sequence view with an abstract class view. Having 14 simple transformation paths available (Figure 69), we can derive a number of complex transformation paths for bridging a concrete sequence diagram with an abstract class diagram. For instance, we could structuralize the sequence view to an object view, abstract the (still

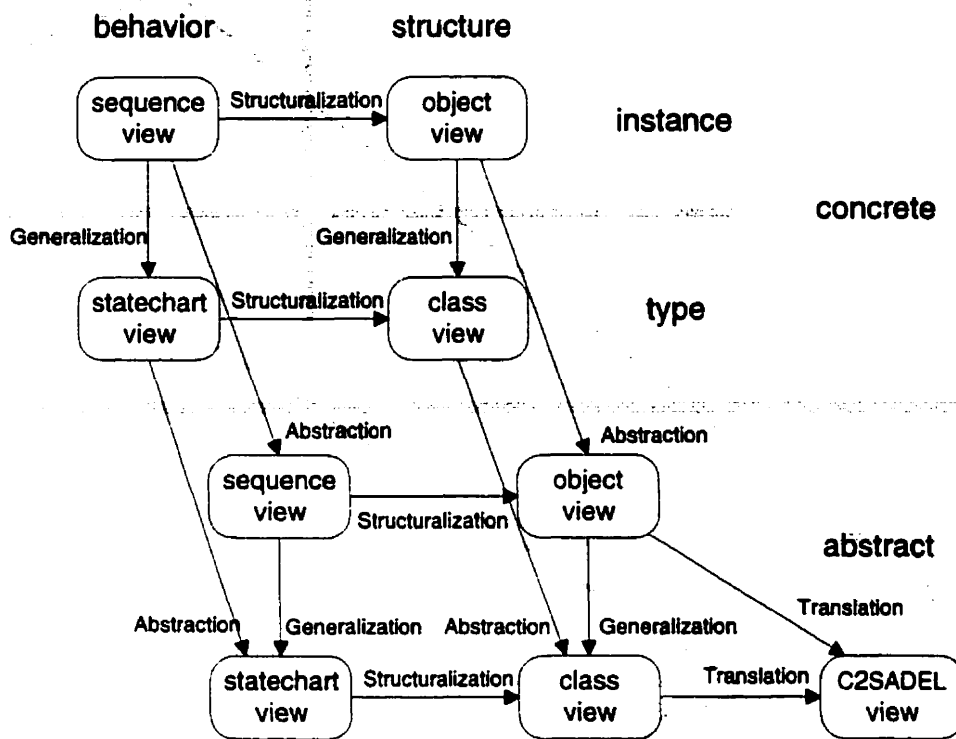


Figure 69. Transformation Methods and Paths

concrete) object view to an abstract object view, and finally generalize from the object view to the class view. This complex transformation scenario involves the serial execution of three transformation methods, resulting in a number of intermediate models. This particular example, depicted as “a)” in Figure 70 is, however, only one of many transformation options. Example “b)” shows that an abstract class view can also be derived by structuralizing the sequence view to an object view, generalizing the object view to a class view, and finally abstracting the class view.

Examples “c)” and “d)” show additional paths for deriving class views. Altogether, there are six paths but no transformation path is superior to the other unless differences in reliability allow their elimination (our reasoning in Section 7.3.1 on how to reuse and eliminate transformation results remains valid for complex transformation). Thus, all (or most) paths should be followed since together they may yield more comprehensive results. Improved comprehensive results are achieved because different intermediate models are used during complex transformation (such as object or state diagrams in the

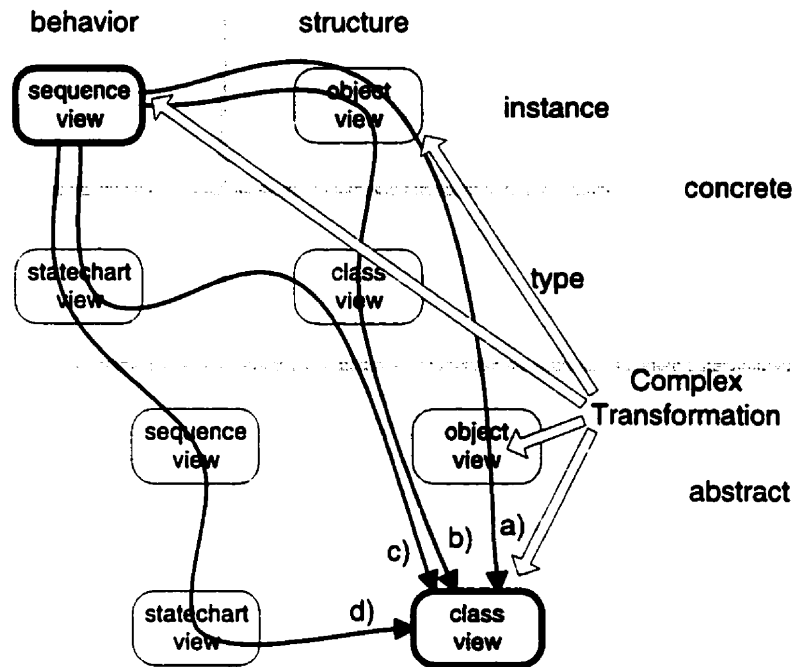


Figure 70. Complex Transformation Paths

above example). These intermediate models contain different model information and thus result in different interpretations. Subsequent transformation, therefore, builds on different intermediate models and interpretations, which, in turn, may build complementary results.

The advantage of having a transformation framework, such as the one in Figure 69, is that it limits the number of possible transformation paths. If bi-directional transformation were fully automatable, then we could potentially derive a much larger number of transformation paths between any two views. The uni-directional nature of transformation, combined with the fact that no circular transformation paths exist, results in a waterfall-like transformation framework. The background arrow in Figure 69 depicts this. The large gray arrow indicated in the figure, shows that views can only be compared by transforming views from the upper-left to the lower-right. Those transformation paths follow along the lines depicted in Figure 43 "c)" and "d)" in page 83 where one view is transformed so that it is more easily comparable in the context of the other. In case two views fall conceptually into the same view quadrant (e.g., both are concrete, specific, and behavioral as in the case of sequence and

1. Check for (transitive) traces from one to the other.
 If none found then stop.
2. Find common denominator
 If none found then stop.
3. Find all transformation paths between each view and its denominator
4. Serially execute remaining transformation paths
5. Combine execution results

Figure 71. Complex Transformation Algorithm

collaboration diagrams), translation can be used to transform one to the other and/or a direct comparison can be attempted (the latter case is supported through the technique depicted in Figure 43 “a”).

However, how can we compare views where neither one can be transformed to the other? For instance, how can we compare an object diagram with a state diagram? There is no simple or complex transformation path available. Our framework enables their comparison indirectly by finding a common denominator. In the case of object and state diagrams, a common denominator is a class diagram. The object diagram can be generalized to a class diagram and, similarly, the statechart diagram can be also structuralized to a class diagram. In the context of the class diagrams, the two can then be compared. This comparison scenario falls along the line depicted in Figure 43 “d).”

Figure 71 describes the complex transformation algorithm. The algorithm requires source and target views as an input. The first step checks, whether trace information between the source and target environment exist. The trace information guides the transformation process and, if none is found, indicates that fully automated transformation is not possible. Step two searches for a common denominator based on existing transformation methods. Again, if none is found, then transformation is not possible. Based on the denominator, all possible transformation paths can be identified and executed (steps 3 and 4). After execution, results can be combined or unreliable ones can be eliminated.

One remaining issue is the quality of intermediate models in handling complex transformations. The question is, are the intermediate models adequate for this task? For instance, if an intermediate view is used to compare a source and target view, but the intermediate model only captures a part of the redundancy between source and target views, then that view alone does not enable a comprehensive view transformation and subsequent comparison. Figure 72 depicts such a case in the context of sequence,

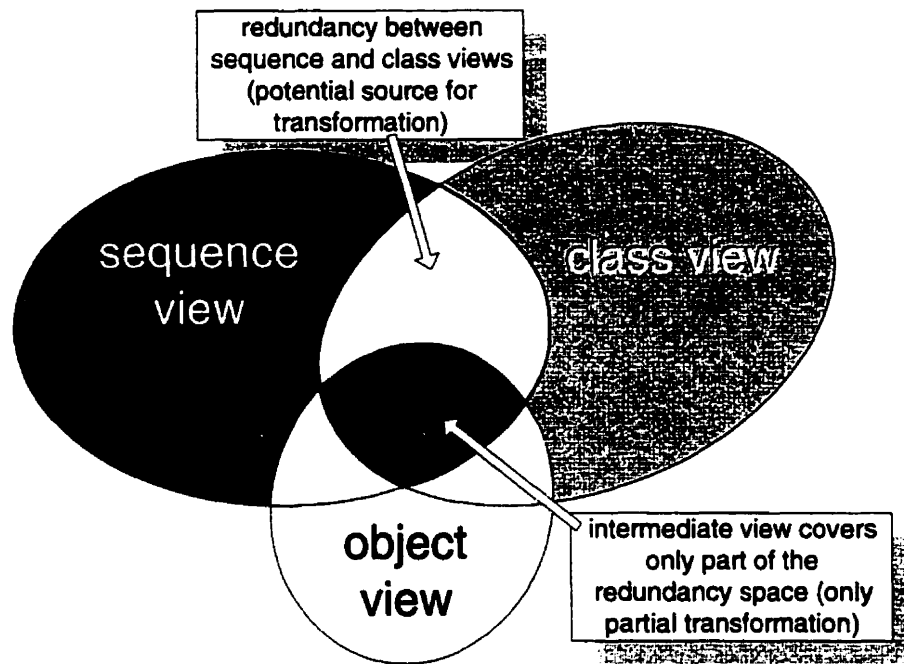


Figure 72. Lack of Intermediate Views in Covering Full Transformation

class, and object diagrams. Sequence and class diagrams share some modeling information, thus exhibiting some amount of redundancy. The figure depicts the redundancy as overlaps between the modeling spaces (ellipses) covered through sequence views and class views respectively. As we had discussed in Section 7, it is the view redundancy that enables automated analysis. Or in other words, it is the information that is shared in multiple views that might become inconsistent.

For transformation this entails two challenges: (1) view transformation should transform all redundant information; and (2) intermediate views need to capture all redundant information. As it was discussed previously, our framework does not support a direct transformation between sequence and class views. Instead, that transformation is broken up into two sequential and simpler transformations, e.g., using object views as intermediate views between them. Object views, however, only capture a part of the redundancy between sequence and class views (see Figure 72). Thus, this type of complex transformation is not fully effective and information is lost in the process. There are three potential options for addressing this issue:

- (1) **Introducing additional intermediate views:** For instance, if a single intermediate view does not adequately cover all redundant model information, then additional views could be introduced and/or the current views be extended. For instance, in our sequence to class transformation problem, we could use statechart views in addition to object views to transform a wider range of modeling information (recall our discussion that all complex transformation paths need to be explored).
- (2) **Building simple transformations:** Instead of using intermediate views to transform sequence to class views, a direct transformation method would be introduced. That step would not require an intermediate view and thus could handle all redundant information. Since our goal is to minimize transformations, we could choose to implement a direct sequence to class transformation covering only those modeling elements that are not covered through complex transformations. Thus, we would not have to build a complete additional transformation method.
- (3) **Extending the underlying model:** Instead of just transforming information that is needed by the derived view, additional information can be added. For instance, structuralization from sequence to object would also add class information. Since views cannot support that additional information, they would have to be annotated somehow. In Section 7.7 we briefly discuss the need of models to comprehensively cover multiple views. These model(s) are prime candidates to store that additional information. In [Egyed and Hilliard 2000], we discuss how model information can be made richer using a *decorative stance* [Hilliard 1999] in view integration.

In practice, any one or more of the options above could to be adopted to enable more comprehensive coverage of transformation needs. In the case of the sequence to class transformation, the introduction of the statechart view is sufficient. In other cases, however, other options have to be considered as well. Since intermediate views only cover a limited range of complex transformation needs, it also follows that there is a trade-off between the numbers of views needed. Basically, there are as many transformation methods and types of views needed as is necessary to ensure comprehensive coverage of all possible view interactions (and their redundancies). We have found that the following options limit transformation paths:

- Reliability numbers associated with (simple) transformation methods (note that we only discussed them in detail in Section 7.3.1 but they also apply to other transformation methods)
- Uni-directional transformation methods (bi-directional ones would result in a larger set of transformation paths)
- Remembering transformation results (instead of eliminating intermediate and final results, they can be stored for later use)
- Treating levels of abstraction separately (instead of transforming between the lowest level and the highest level, only adjacent levels need to be transformed)

7.4.1 Deferred Issues

Transformation cannot be seen as isolated in the context of our view integration framework. The following critical aspects will be addressed by other parts of our framework:

- Modeling Transformation Redundancy (see Section 7.7)
- Scalability Issues (see Section 7.7)
- Types of Traces Needed to Support Simple/Complex Transformations (see Section 7.6.2)
- Interpretation/Realization Relationships (see Section 7.6.2)

Above sections on simple and complex transformation discussed the problem of transformation in detail. We primarily focused on the problem of automated abstraction and only indicated informal solutions for generalization, structuralization, and translation. Although all four types of transformation are equally important, we explained in Section 4 why we have emphasized more strongly on one over the other. The methods we discussed to support abstraction, generalization, structuralization, and translation are dependent on the types of views we support in our thesis (Figure 46). For instance, structuralization only discussed sequence-to-object and statechart-to-class transformations since only these are needed to support our limited number of views. Recall Section 5.5, where we outlined that our limited set of views are sufficient in covering all three heterogeneous view dimensions. Naturally, if other views were needed, our framework would have to be extended by support by supporting additional transformation methods.

7.5 Automating Model Differentiation

Transformation converts modeling information between various view dimensions and makes that information more abstract, more generic, or more structural in the process. The converted information is referred to as derived information and, ideally, it reflects the redundancies between view dimensions (e.g., converting a sequence diagram to a class diagram implies taking all information from the sequence diagram that is potentially redundant with a class diagram). In Section 7.2, we discussed that it is the redundancy between views that poses constraints on views. Since derived information represents the redundancies between views, derived information can also be seen as constraints. The role of differentiation is to take derived model elements and to compare them with user-defined model elements, with the purpose of identifying differences (ergo differentiation). Since differences between user-defined information and derived information generally indicate constraint violations, those differences can be considered inconsistencies.

Based on traceability information (see mapping in Section 7.6.2) it can be specified what information needs to be compared. For instance, if there are two views with no mapping between them then this may imply the views are not related to one another. A direct comparison between two unrelated views is not necessary. Ideally, differentiation should be able to compare views directly without any additional overhead. However, as it was discussed in Section 7, it is frequently not straightforward to compare views such as sequence and state diagrams directly. Differentiation is thus complicated by syntactic and semantic differences between views. We, therefore, use transformation to convert model information to allow a direct comparison between derived elements and all user-defined elements that relate to it (e.g., transformation of a sequence diagram to a derived statechart diagram and the subsequent comparison between that derived statechart diagram and the corresponding user-defined statechart diagram). Different values in derived and user-defined model elements denote inconsistencies. Mapping and transformation support differentiation 1) by constraining *what* information has to be compared (through mapping) and 2) by defining *how* information has to be compared (through transformation).

Mapping and Transformation are therefore enabling technologies for more effective, less complex, and more scalable differentiation.

Differentiation is, however, more than just comparing model elements. Differentiation must also address what to do if inconsistencies are found. Transformation and mapping may simplify comparison but that does not imply that all of differentiation is simplified. For inconsistency checking, differentiation is responsible for three major tasks: 1) applying transformation methods; 2) comparing their results; and 3) reporting the findings to the user. The following sections will address all three aspects in more detail.

7.5.1 Comparing User-Defined and Derived Elements

Differentiation has to take derived elements and compare them with existing user defined elements. Depending on the types of views (as well as their transformations), the comparison may vary:

7.5.1.1 Comparison Modes

- **Equivalence comparison:** There are cases where transformation yields derived results that fully correspond to user-defined elements. For instance, if an abstraction process derives an association between two lower-level classes, then those same two classes should also have corresponding higher-level classes with an association between them (see Figure 73). The comparison between both values is to ensure equality.
- **Part-of comparison:** There are cases where transformation cannot generate a complete picture of the real situation. For

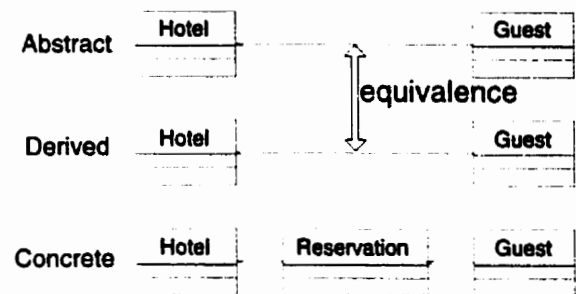


Figure 73. Examples of Equivalence Comparison

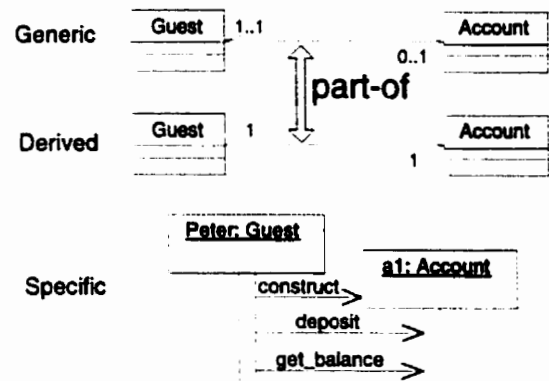


Figure 74. Examples of Part-of Comparison

instance, a sequence diagram likely only depicts a subset of an object's potential interactions. If a sequence diagram is generalized into a class diagram, then it follows that the derived class diagram only depicts a part (subset) of the existing user-defined class diagram. For instance, in Figure 74 the sequence diagram shows calls from the object *Peter* to the object *a1*. Since *Peter* is of type *Guest* and since *a1* is of type *Account*, it follows that at a derived but more generic level, the class *Guest* must depend on the class *Account*. That observation is confirmed by the real (user-defined) class diagram (top of Figure 74), however, that diagram additionally conveys that *Account* may also access *Guest*. The comparison between derived and user-defined elements, therefore, needs to ensure a part-of relationship.

Whether the comparison has be done in the “part-of-mode” or the “equivalence-mode” depends on the type of inconsistency (recall Section 6). Consistency checking between specific views and generic views is more likely to use part-of comparison. Consistency checking between abstract and concrete views is more likely to use equivalence comparison. There are, however, exceptions to both. Later we will see that consistency rules need to specify their comparison modes.

7.5.1.2 Multiple Interpretations and Realizations

Another complication of comparison is the issue of multiple derived interpretations and/or multiple user-defined realizations. For instance, there are transformation methods such a class abstraction that may yield multiple results. If such a case occurs, then comparing those multiple results with a single user-defined realization may become more challenging. The following describes those cases and discusses rules for handling them.

Figure 73 depicted the trivial case of a one-to-one relationship between a derived model element and a user-defined model element. The bottom of the figure showed a concrete class diagram and the top showed the corresponding abstract class diagram. By abstracting the concrete class diagram using our class abstraction mechanism, an intermediate class diagram is created (middle). The elements of the intermediate diagram are derived interpretations of the concrete diagram and correspond directly to the

abstract diagram on the top. Checking for consistency between them is therefore simply a comparison of the derived value with their corresponding user-defined values on a one-to-one basis.

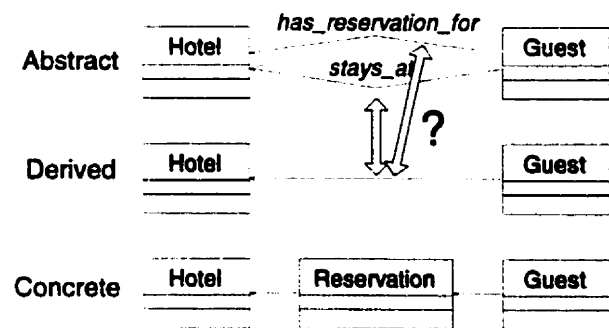


Figure 75. One-to-many Comparison

Figure 75 depicts a less trivial, though similar example. Here, the abstracted class diagram was slightly modified and depicts two (abstract) relationships between the classes. The abstraction of the concrete diagram does not change and again yields the same

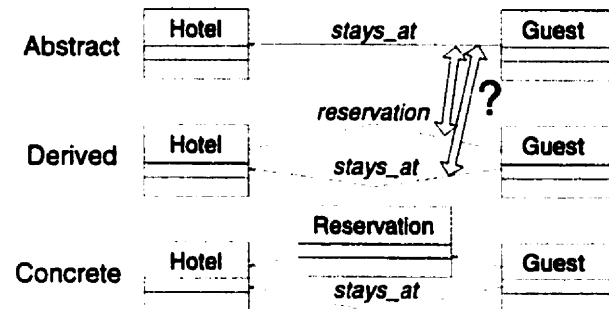


Figure 76. Many-to-One Comparison

intermediate class diagram. However, the relationship between the intermediate class diagram and the abstract class diagram is more complicated. The derived classes in the intermediate model still correspond one-to-one to the abstract classes, but the single intermediate relationship has two alternatives. The intermediate relationship could belong to either the abstract relation *has_reservation_for* or the abstract relation *stays_at*. Therefore, for comparison we also need to consider the possibility that a derived element may belong to one of several abstract relationships.

Figure 76 depicts a third scenario in the context of the same situation. Again a concrete class diagram is abstracted into an intermediate class diagram. In this scenario, the abstract class diagram has only a single association relationship, whereas the derived but abstract class diagram has two relationships. Comparison, therefore, also needs to consider the possibility that several derived elements may correspond to a single abstract element.

Figure 77 depicts a forth scenario in the context of the same situation. Again a concrete class diagram is abstracted into an intermediate class diagram. In this scenario, the abstract class diagram has

the proper classes but no relation between them. Comparison, therefore, also has to consider the lack of a comparable element.

To support the capture and evaluation of intermediate (derived) model elements together with their user-

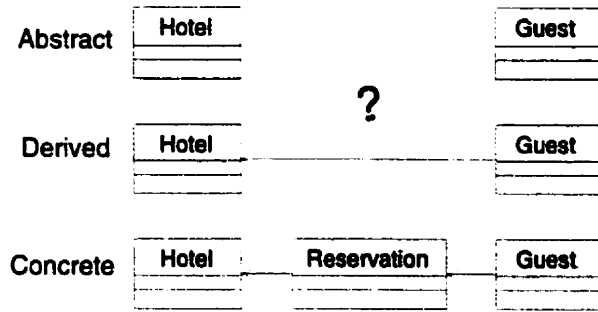


Figure 77. Zero-to-one Comparison

defined model elements, we require a more advanced form of repository. In Section 7.7 we will discuss the concept of a reduced redundancy model and its ability to integrate user-defined and derived information. Our repository currently supports zero or more derived elements to be associated with each user-defined element. In the case of the non-existent user element in Figure 77, our repository creates a dummy user-defined model element that is of type *unknown*. The reason for the creation of the dummy element is that this scenario can then be treated in the same manner as the others.

Figure 78 depicts all supported comparisons between user-defined and derived elements. Case (a) shows the one-to-one scenario where one user-defined model element is compared with only one derived model element at any given time. Note that this scenario does not disallow multiple derived interpretations as long as each interpretation is compared with exactly one user-defined model element.

Case (b) shows the scenario where two interpretations are compared with a single user-defined model element. In this scenario, both derived interpretations must be compared with the user-defined realization (note that we use the term realization to indicate user-defined elements that are comparable to derived interpretations).

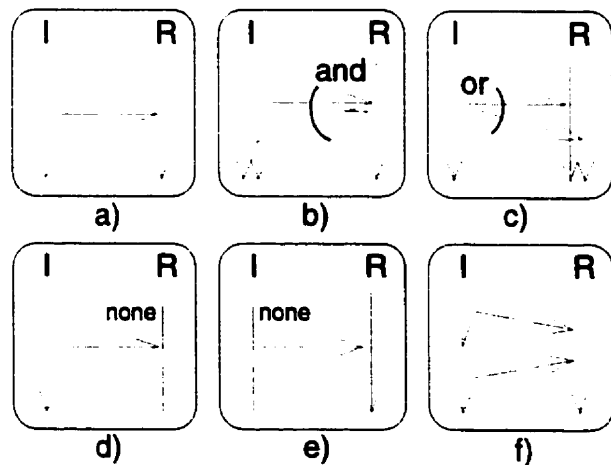


Figure 78. Variations in View Comparison

Case (c) shows the scenario of one interpretation for two user-defined model elements. In this scenario, the derived interpretation must be matched with at least one of the user-defined ones. A combination of cases (b) and (c) can also happen where two derived interpretations exist for two user-defined model elements. There, both derived interpretations are compared with both user-defined model elements. Case (d) shows the scenario where a derived interpretation is found but no user-defined model element is known. Case (e) is a counterpart to case (d) in that a user-defined element exists but no derived element is generated. Both latter cases are indications of incompleteness and, depending on the types of views, may also show inconsistencies.

Consistency rules, which will be discussed later, do not need to explicitly specify what to do in the case of multiple interpretations or realizations. Instead, our underlying view integration framework has to ensure that the above constraints are enforced.

7.5.1.3 Ambiguous Interpretations

A final complication of comparison is the issue of ambiguity. Figure 79 depicts the situations that may apply: (1) the interpretation is ambiguous and either the one or the other applies; or (2) the realization is ambiguous and, again, either one or the other applies. Figure 80 depicts an example that

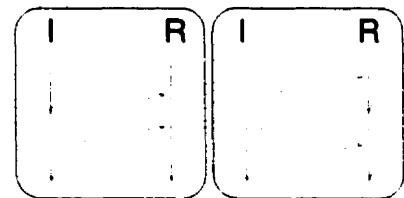


Figure 79. Ambiguous Comparison

shows some cases of variations (ambiguities) among interpretations. At the bottom, two input diagrams are shown. The left diagram is a sequence view describing the interactions between a guest and his account. Since the *create* method constructs a new object, we can infer that *create* is a <<constructor>>. A constructor causes a state change from a start state to a regular state (see Section 7.3.3). The state diagram at the top of Figure 80 indeed shows the *create* method in that form. The impact of the *deposit* method on the state diagram is, however, less clear. Since no additional information is available, we have to assume *deposit* to be either a query (does not change state) or an action (may change state).

The {action or query} tag indicates an ambiguity for this interpretation. However, it is still valuable to capture those values since they do constrain the situation somewhat (e.g., we exclude the

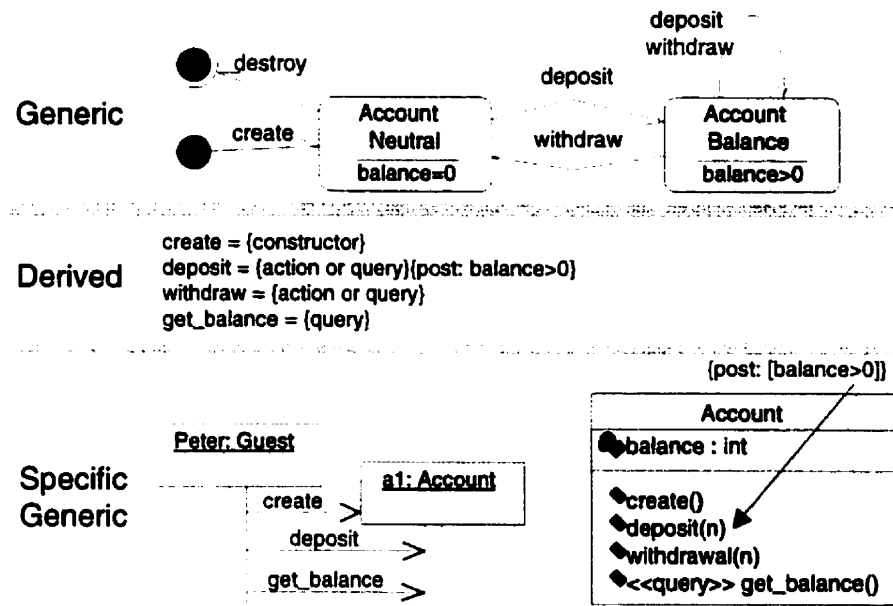


Figure 80. Variations (Ambiguities) in Transformation Results

possibility that it is an activity). For *deposit* we also have a second interpretation, which comes from the class diagram—the post condition must result in a balance greater than zero. That information allows *deposit* to be associated with the *Account Balance* state since that state has the same invariant. The state diagram depicts *deposit* to cause a state change from *Account Neutral* to *Account Balance*. Thus *deposit* is an action. This observation does not violate the {action or query} constraint since the derived interpretation allows *deposit* to be an action. The example in Figure 80 has no inconsistency; however, given the constraints the diagrams pose onto one another, a number of potential inconsistencies may occur: (1) if *create* is ever used as a message that does not create an object; (2) if *deposit* is used as a constructor; (3) if *deposit* would be declared a <<query>>; (4) if *deposit* has the post condition {balance=0} or even {balance>=0}; and so forth. Thus, ambiguous transformation results still support consistency checking and must be captured.

Inconsistencies are identified when derived interpretations and user-defined realization do not match. The comparison is done either in the “part-of” mode or the “equivalence” mode (as it was discussed above). The comparison rules and ambiguity issues discussed above must be supported implicitly and thus do not have to be specified explicitly as part of consistency rules. Comparison is

simplified by not having to compare between all views (or model elements). There is no value in comparing specific views with one another since they only exhibit usage scenarios. For instance, if one scenario claims A to be equal to 6 and the other claims A to be equal to 9, this does not always imply inconsistency. It only implies that A could either be 6 or 9 (an ambiguity). Comparisons between specific views and generic views are, however, meaningful. Similarly meaningful are comparisons among generic views as well as with their abstractions. In some cases, comparisons within single diagrams may also be meaningful. Those cases are, however, much simpler because direct comparisons without transformations often suffice.

7.5.2 Consistency Rules

7.5.2.1 List of Inconsistencies

In the course of evaluating UML we have identified about fifty types of inconsistencies. Section 6 discussed and illustrated them in detail. The following tables summarize those inconsistencies. Although we believe that all types of inconsistencies identified in this section can be detected automatically, we have not yet analyzed and automated them in sufficient detail to support that claim. The tables indicate the degree of tool support we have in place currently. *Full* support implies that both transformation and consistency checking can be done automatically. *Semi* support implies that only transformation has been automated and consistency checking still has to be done manually. We will discuss our tool and its automation in detail later in Section 8.

Table 5. List of Inconsistencies on the Abstract/Concrete Dimension

	Description	Views	Tool
1	Concrete relation has no corresponding abstraction	General	Full
2	Abstract relation has not been refined	General	Full
3	Concrete classifier has no corresponding abstraction	General	Full
4	Abstract classifier has not been refined	General	Full
5	Concrete relation is of different type than its corresponding abstraction	General	Full

6	Concrete classifier is of different type than its corresponding abstraction	General	Full
7	Concrete relation uses abstract classifier instead of its refinement	General	
8	Abstract relation uses concrete classifier instead of its abstraction	General	
9	Abstract classifier is replicated on concrete level although refinement exists	General	
10	Concrete classifier is assigned to multiple abstract classifiers	General	Full
11	Cardinality of refinement does not match its abstraction	Class	Full
12	Direction of concrete relation does not match its abstraction	General	Full
13	Concrete classifier does not replicate a method of its abstraction	Class	
14	Concrete classifier does not replicate an attribute of its abstraction	Class	
15	Concrete method is of different type than its corresponding abstraction	Class	
16	Concrete attribute is of different type than its corresponding abstraction	Class	
17	Abstract and public method is hidden in refinement	Class	
18	Abstract and public attribute is hidden in refinement	Class	
19	Abstract pre-conditions may not become stronger in refinement	General	
20	Abstract post-conditions may not become weaker in refinement	General	
21	Abstract invariant may not become weaker in refinement	General	

Table 6. List of Inconsistencies on the Generic/Specific Dimension

	Description	Views	Tools
1	Specific relation has no corresponding generalization	General	
2	Generic relation has never been instantiated	General	
3	Specific classifier has no corresponding generalization	General	
4	Generic classifier has never been instantiated	General	
5	Specific relation is of different type than its corresponding generalization	General	
6	Specific classifier is of different type than its corresponding generalization	General	

7	Cardinality of generic classifiers does not match specific scenarios	General	
8	Direction of specific relation does not match its generalization	General	
9	Generic method has never been instantiated	General	
10	Generic attribute has never been instantiated	General	
11	Specific method is of different type than its corresponding generalization	General	
12	Specific attribute is of different type than its corresponding generalization	General	
13	Specific view uses a method that is not defined in generic classifier	General	
14	Specific view uses an attribute that is not defined in generic classifier	General	
15	Specific relation has not been assigned to generic relation	General	
16	Specific classifier has not been assigned to generic classifier	General	
17	Generic pre-condition is violated in specific view	General	
18	Generic post-condition is violated in specific view	General	
19	Specific method used was declared private in generic view	General	
20	Specific attribute used was declared private in generic view	General	
21	State transition does not match method declaration	SC-S	
22	State description does not match method declaration	SC-S	
23	Method call order is violated	SC-S	

Table 7. List of Inconsistencies on the Structural/Behavioral Dimension

	Description	Views	Tools
1	Imported guard was not declared in structural view	SC-C	
2	Imported trigger was not declared in structural view	SC-C	
3	Structural view does not allow an interaction as required by guard	SC-C	
4	Structural view does not allow an interaction as required by trigger	SC-C	
5	Relationship between classes is not reflected in statechart	SC-C	

6	Method was declared “query” but is used for non-circular state transitions	SC-C	
7	Method was declared “action” but is used for state transitions	SC-C	
8	Method was declared “activity” but is used for non-circular state transitions	SC-C	
9	Guards leaving state are not mutually exclusive	SC-C	
10	Guard/trigger pre- or post condition does not match method condition	SC-C	

7.5.2.2 Simple Consistency Checking Example

In Section 7 above, we indicated that automated consistency checking requires rules for validating constraints (redundancies). In our framework, rule validation is reduced to a simple comparison of user-defined and derived model elements. Section 7.5.1 further complemented this by introducing and discussing comparison rules in the case of ambiguity or multiplicity. An issue that still remains is how and where to apply those comparison rules. To enable automated comparison, we need to specify conditions that indicate what consistency actually is. To that end, consistency rules must specify the groups of model elements they apply to as well as the conditions that must remain valid so that those groups of model elements can be considered consistent. The following will introduce consistency rules and will discuss them in the context of examples.

Figure 81 depicts an example of an inconsistency between two class diagrams at different levels of abstraction. The diagrams depict a simplified view of a hotel management system. The system is presented in two layers and has the constraint that each layer is supposed to present the system in a complete fashion although at different levels of abstraction. The first layer (top) shows the interactions between the classes *Hotel* and *Guest*. It is stated that a *Guest* may stay at a *Hotel* and that a *Guest* may have reservations for *Hotels*. The more concrete layer (bottom) shows refinements of the classes *Guest* and *Hotel* as well as a refinement of one of their

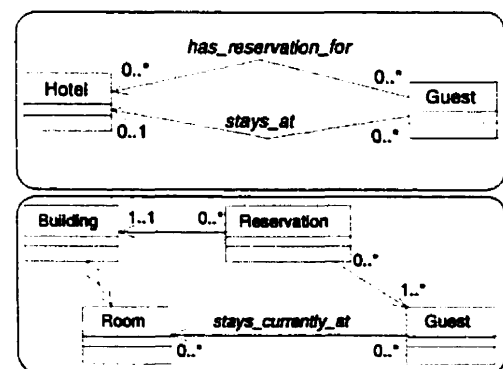


Figure 81. Refinement Inconsistency

relationships. It can be observed that *Hotel* was decomposed into having a *Building* which has *Rooms*. The class *Guest* still exists; however, its dependencies to the classes *Building* and *Room* are refined via the helper class *Reservation*. Since both diagrams (refinement and abstraction) depict the same part of the hotel management system, it follows that the information depicted within them must be consistent with one another. The issue of consistency is, however, hard to validate—even in a simple example as the one above—because:

- The class *Hotel* is not present in the lower-level design.
- The relationship between *Guest* and *Room* (the latter being part of *Hotel*) is obstructed by the helper class *Reservation*.

The example actually contains two inconsistencies. If we assume *Room* to be a surrogate of *Hotel* (which it partially is) then the cardinality between *Guest* and *Hotel* should be identical to the cardinality between *Guest* and *Room*. The higher-level design states that a guest may stay at most at one hotel at any given time, whereas the lower-level design states that a guest may stay at zero, one, or more hotels at any given time—an inconsistency. The second inconsistency in the example is in the direction of the relations. Whereas the higher-level diagram states that *Guest* may have reservations for *Hotel* (the uni-directional nature of the association implies that *Guest* may access methods of *Hotel*), the lower-level diagram depicts the class *Reservation* at the center of that interaction—another inconsistency.

In Section 7.2, we discussed that inconsistencies are based on redundancies between diagrams. Figure 81 illustrated this in the case of abstract information that poses constraints on refinements. For instance, the knowledge that *Guest* and *Hotel* interact at an abstract level is a constraint that such an interaction must also be implemented at a lower level (equivalence comparison). Similarly, the lack of an interaction at an abstract level is a constraint on the lower level not to interact.

In Section 7.3 we, therefore, discussed on how to use transformation to enable the comparison between different types of diagrams. In particular, for the example in Figure 81, we need abstraction. Figure 82 depicts the findings of that abstraction process. The bottom diagram shows the lower-level design from Figure 81. Our abstraction process is fully tool supported and uses abstraction rules (see Section 7.3.1.3.3) to automatically replace more complex class patterns with less complex (more abstract)

ones. The abstraction process described in Figure 81 involves two steps. The first step merges the classes *Room* and *Building* into a composite class *Hotel*. The composite class *Hotel* “inherits” the interfaces from *Building* and *Room* and, thus, has relations to *Reservation* and *Guest*. The second abstraction step eliminates the helper class *Reservation* since it obstructs our view onto the direct relationship between *Guest* and *Hotel*. The final result is a derived diagram where the direct relationships between *Guest* and *Hotel* are depicted (top of Figure 82). We refer to

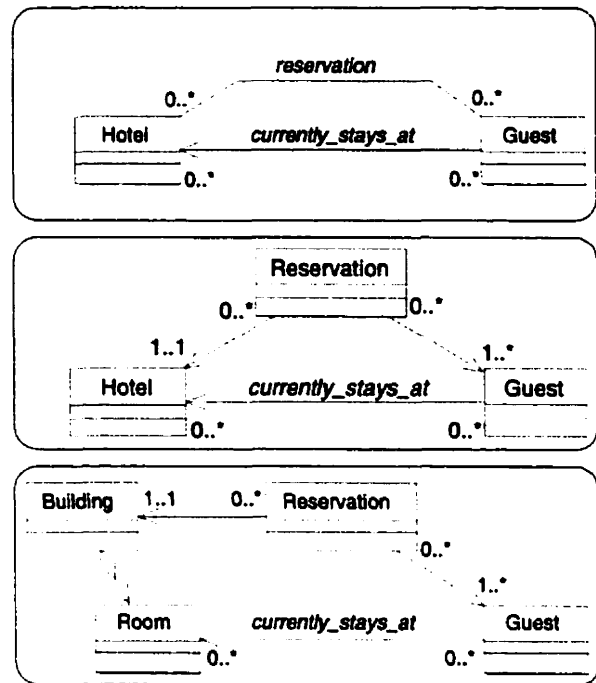


Figure 82. Abstraction Example

this diagram as the (abstract) interpretation of the lower-level diagram since it is directly comparable with the higher-level diagram in Figure 81.

To illustrate the process of comparison, consider Figure 83. The top and bottom rows depict the user-defined diagrams from Figure 81. The two rows in the middle are the derived diagrams we got after abstraction (see Figure 82). Since transformation and consistency checking relies on the existence of mapping information (see Section 7.6.2), Figure 83 also depicts trace information in form of vertical arrows that link elements between rows (trace mappings are shown as dashed arrows, e.g., between *Building* and *Hotel*). Additionally, transformation created interpretation relationships between some derived elements and higher-level elements to indicate that those elements are comparable. We refer to derived model elements that are comparable as *interpretations*. As a general rule, interpretations tend to be final transformation results as in above case were the most abstract model elements have become interpretations. Interpretation relationships are represented as solid vertical arrows with circles on both ends.

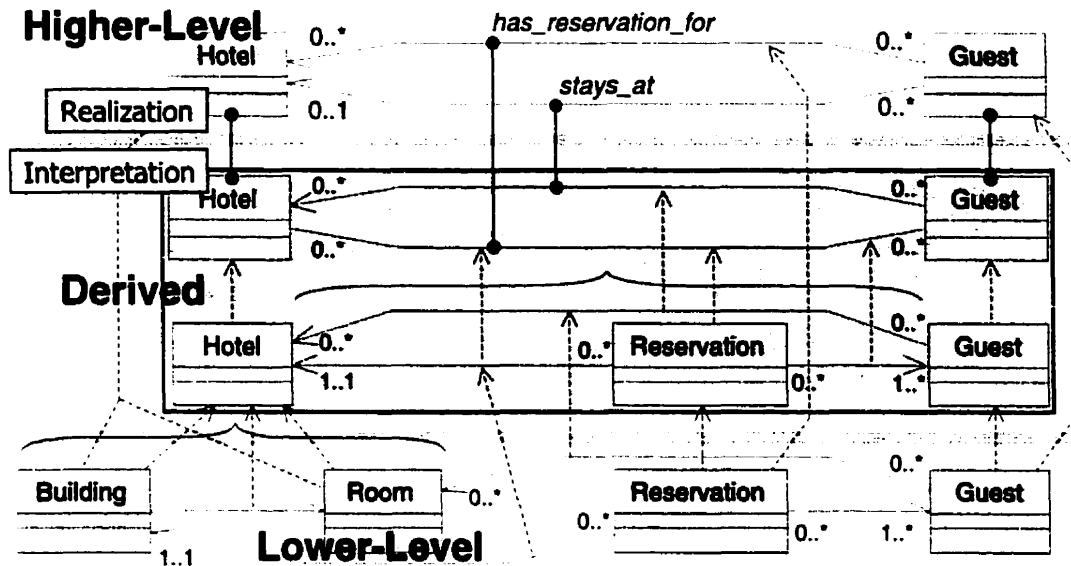


Figure 83. Example of Consistency Checking between Abstract and Concrete Elements

7.5.2.3 Consistency Rules Defined and Applied

Based on the relationships between user-defined and derived elements, consistency rules can be validated. The following shows a consistency rule that states that for each interpreted relationship there must be a *realization*. Note that a realization is the opposite of an interpretation. If interpretations depict derived elements that are comparable to user-defined elements then realizations depict the corresponding user-defined elements that are comparable to derived elements.

Concrete relation has no corresponding abstraction:

$$\forall r \in \text{relations}, \text{is_classmodel}(r) \wedge \text{is_abstraction}(r) \wedge \text{is_interpretation}(r) \Rightarrow \text{realization}(r) \neq \text{NULL}$$

Above rule applies to relations in the model. In Figure 83 relations are the horizontal arrows between classifiers (e.g., between *Guest* and *Hotel*). Above rule then qualifies what relations are actually meant. The following restrictions are defined: (1) only those relations that are part of class models (all in our case); (2) only those relations that are abstractions (all derived and high-level ones in our case); and (3) only those relations that are interpretations (only the two abstract relations in the second row from

top). This rule, therefore, only applies to abstracted relations of class diagrams that are comparable to user-defined ones. The rule then states that the realizations of those relations must also exist. In our example this rule is not violated since for both abstracted relations we also find at least one realization.

Direction of concrete relation does not match its abstraction:

$$\forall r \in \text{relations}, \text{is_classmodel}(r) \wedge \text{is_abstraction}(r) \wedge \\ \text{is_interpretation}(r) \wedge \text{realization}(r) \neq \text{NULL} \Rightarrow \\ \text{realization}(r \rightarrow \text{destination}) = (\text{realization}(r)) \rightarrow \text{destination}$$

This rule applies to the same relations as the previous rule with the additional constraint that realizations must exist. Since realizations exist for both interpreted relations, this rule is applied to both. The rule states that the relations in question must have the same destination. For instance, both relations in the higher-level diagram in Figure 83 go from *Guest* to *Hotel* (see arrow head). The “(realization(r))→destination” part of above rule, therefore, returns *{Hotel}* in both cases. Note that “realization(r)” yields the user-defined, higher-level relations *{has_reservation_for, stays_at}* and that “→destinations” yields the destination set of these relations (there may be multiple destinations since UML allows more than one destination per relation). On the other hand, the “realization(r→destination)” part of above rule returns *{Hotel}* for one relation and *{Hotel, Guest}* for the other (note that associations without arrows denote bi-directional associations). It follows that in one case, above consistency rule is valid whereas in the other case the rule is violated. The violation reveals that the interpreted result has a different direction of interaction than its realization.

Concrete classifier is of different type than its corresponding abstraction:

$$\forall r \in \text{relations}, \text{is_classmodel}(r) \wedge \text{is_abstraction}(r) \wedge \\ \text{is_interpretation}(r) \wedge \text{realization}(r) \neq \text{NULL} \Rightarrow \text{type}(r) = \\ \text{type}(\text{realization}(r))$$

This rule again applies to the same set of relations as above one and states that the interpreted relationship must be the same as the type as the user-defined one with which it is compared. The “type(r)” part results in an *association* in both cases. Since the realizations for both relations “type(realization(r))” are also associations, it follows that no inconsistency of this type exists.

Cardinality of refinement does not match its abstraction:

```


$$\forall r \in \text{relations}, \text{is\_classmodel}(r) \wedge \text{is\_abstraction}(r) \wedge$$


$$\text{is\_interpretation}(r) \wedge \text{realization}(r) \neq \text{NULL} \wedge \text{type}(r) = \text{"association"} \wedge$$


$$\text{type}(\text{realization}(r)) = \text{"association"} \Rightarrow \text{cardinality}(r) =$$


$$\text{cardinality}(\text{realization}(r))$$


```

This rule also applies to all relations as the previous one but further qualifies that those relations must be of type "association." It states that the cardinality of that association has to match the cardinality of its realization. Note that this rule is rather generic in that it could be refined to check for source and/or destination cardinalities. Also it could be refined to check for lower and upper bounds. Comparing cardinalities is also an example where one-to-one comparison cannot be easily implemented. For instance, a user could specify the same cardinality as [1..5] or as [1,2,3-5]. Comparing cardinality, therefore, requires the normalization of its contents (another transformation).

We analyzed the consistency cases between abstract and concrete diagrams and we identified 21 consistency rules (see Table 5). In Figure 83 most of those rules were not violated. Nevertheless, we would have to validate all rules against the model to ensure that. The consistency rules we discussed thus far followed along the same pattern and the rules were very similar. Rules can, nevertheless, get more complicated. Consider the following example:

Abstract relation has not been refined

```


$$\forall r \in \text{relations}, \text{is\_classmodel}(r) \wedge \text{is\_realization}(r) \wedge \text{is\_refineable}(r) \Rightarrow$$


$$\exists ir \in r \rightarrow \text{interpretations}, \text{is\_abstraction}(ir) \wedge \text{is\_classmodel}(ir \rightarrow \text{origin})$$


```

This rule uses a number of different constructs for validation. The "is_refineable(r)" part verifies whether or not other model elements of the same level have been refined. This is necessary since we do not want to accidentally declare the most refined class diagram has having relations that have not been refined. This rule is also different in that the "implies part" is more complicated. Since we are not searching for interpretations but for realizations ("is_realization(r)") and, as we discussed in Section 7.5.1.2, since more than one interpretation may exist per realization, it follows that each interpretation has to be searched. In above case, we want to ensure that a refinement exists for a refineable element.

Thus, we need to find at least one interpretation that is an abstraction from something else. That alone is not sufficient since we learnt in Section 7.4 that complex transformations are also possible. Thus, there could be a structuralized sequence diagram that was then abstracted. In that case, the abstracted information would qualify as an abstraction ("is_abstraction(ir)") although it was not derived from a direct refinement. We, therefore, need to additionally specify the origin of that derived element as being from a class model. We can use "is_classmodel" again for that purpose and provide as an input the elements from which the derived elements were built of ("ir->origin"). Origin traverses the classifier tree in Figure 83 downward until user-defined elements are found (e.g., *Building*, *Room*, and its relation).

Above examples only discussed relations. Handling abstract and concrete classes (classifiers) is very similar:

Concrete classifier has no corresponding abstraction:

$$\forall c \in \text{classifiers}, \text{is_classmodel}(c) \wedge \text{is_abstraction}(c) \wedge \text{is_interpretation}(c) \Rightarrow \text{realization}(c) \neq \text{NULL}$$

Concrete classifier is of different type than its corresponding abstraction

$$\forall c \in \text{classifiers}, \text{is_classmodel}(c) \wedge \text{is_abstraction}(c) \wedge \text{is_interpretation}(c) \wedge \text{realization}(c) \neq \text{NULL} \Rightarrow \text{realization}(c \rightarrow \text{destination}) = (\text{realization}(c)) \rightarrow \text{destination}$$

Abstract classifier has not been refined

$$\forall c \in \text{classifiers}, \text{is_classmodel}(c) \wedge \text{is_realization}(c) \wedge \text{is_refineable}(c) \Rightarrow \exists ic \in c \rightarrow \text{interpretations}, \text{is_abstraction}(ic) \wedge \text{is_classmodel}(ic \rightarrow \text{origin})$$

Above rules emphasized in equivalence comparison (recall Section 7.5.1.1). Indeed, equivalence comparison is the pre-dominant form of comparison on the abstract-concrete dimension but there are exceptions such as:

Abstract pre-conditions may not become stronger in refinement

$$\forall m \in \text{methods}, \text{is_classmodel}(m) \wedge \text{is_abstraction}(m) \wedge \text{is_interpretation}(m) \wedge \text{realization}(m) \neq \text{NULL} \wedge \text{precondition}(m) \neq \text{NULL} \Rightarrow \text{precondition}(m) \supseteq \text{precondition}(\text{realization}(m))$$

This rule validates methods of class diagrams. Methods are services (e.g., functions) that classes provide as an interface. For instance, the *Reservation* class in Figure 83 may have methods like *set_arrival_date* or *set_number_of_days*. Some of those methods may have preconditions. For instance, a method *set_arrival_date* may have the precondition that the arrival date cannot be the current date nor any past date. Thus, a reservation can only be made at least one day in advance. A refinement should ideally have the same pre-condition (equivalence), however, it is valid to weaken it a bit. For instance, if during refinement another method is created which has the relaxed pre-condition that no past arrival dates should be used, then this refinement does not contradict the abstraction. The refinement method still provides the same services as the original one only that it provides additional functionality. Note that this case should not be seen as allowing a requirements change. The validation that a reservation has to be done at least a day in advance still has to happen. Having a relaxed way of refinement supports concepts like class libraries or COTS (commercial-off-the-shelf) packages that frequently do much more than required. For instance, if the refined method is a part of a COTS package and it meets the same or weaker pre-conditions then it can be used as a substitute of that abstract element. Should above consistency rule not be applicable in other situations, it could be deleted, ignored, or replaced by a more adequate one. For instance, we could replace the “ \supseteq ” operator with a “=” if desired.

Defining consistency rules between specific and abstract elements is very similar to consistency rules for abstract and concrete elements. It was already indicated above that the types of inconsistencies are very similar. The main difference between consistency rules for abstraction as compared to those for generalization are the part-of relationships.

Specific relation has no corresponding generalization:

$$\forall r \in \text{relations}, \text{is_classmodel}(r) \wedge \text{is_generalization}(r) \wedge \\ \text{is_interpretation}(r) \Rightarrow \text{realization}(r) \neq \text{NULL}$$

Direction of specific relation does not match its generalization

$$\forall r \in \text{relations}, \text{is_classmodel}(r) \wedge \text{is_generalization}(r) \wedge \\ \text{is_interpretation}(r) \wedge \text{realization}(r) \neq \text{NULL} \Rightarrow \\ (r \rightarrow \text{destination}) \rightarrow \text{realization} \subseteq (\text{realization}(r)) \rightarrow \text{destination}$$

Cardinality of generic classifiers does not match specific scenarios

$$\forall r \in \text{relations}, \text{is_classmodel}(r) \wedge \text{is_generalization}(r) \wedge \text{is_interpretation}(r) \wedge \text{realization}(r) \neq \text{NULL} \wedge \text{type}(r) = \text{"association"} \wedge \text{type}(\text{realization}(r)) = \text{"association"} \Rightarrow \text{cardinality}(r) \subseteq \text{cardinality}(\text{realization}(r))$$

Above examples showed that consistency checking for generalization involves the same pattern of consistency rules for abstraction. The main differences are the use of the "is_generalization" construct and the more frequent use of the " \subseteq " operator instead of the "=" operator.

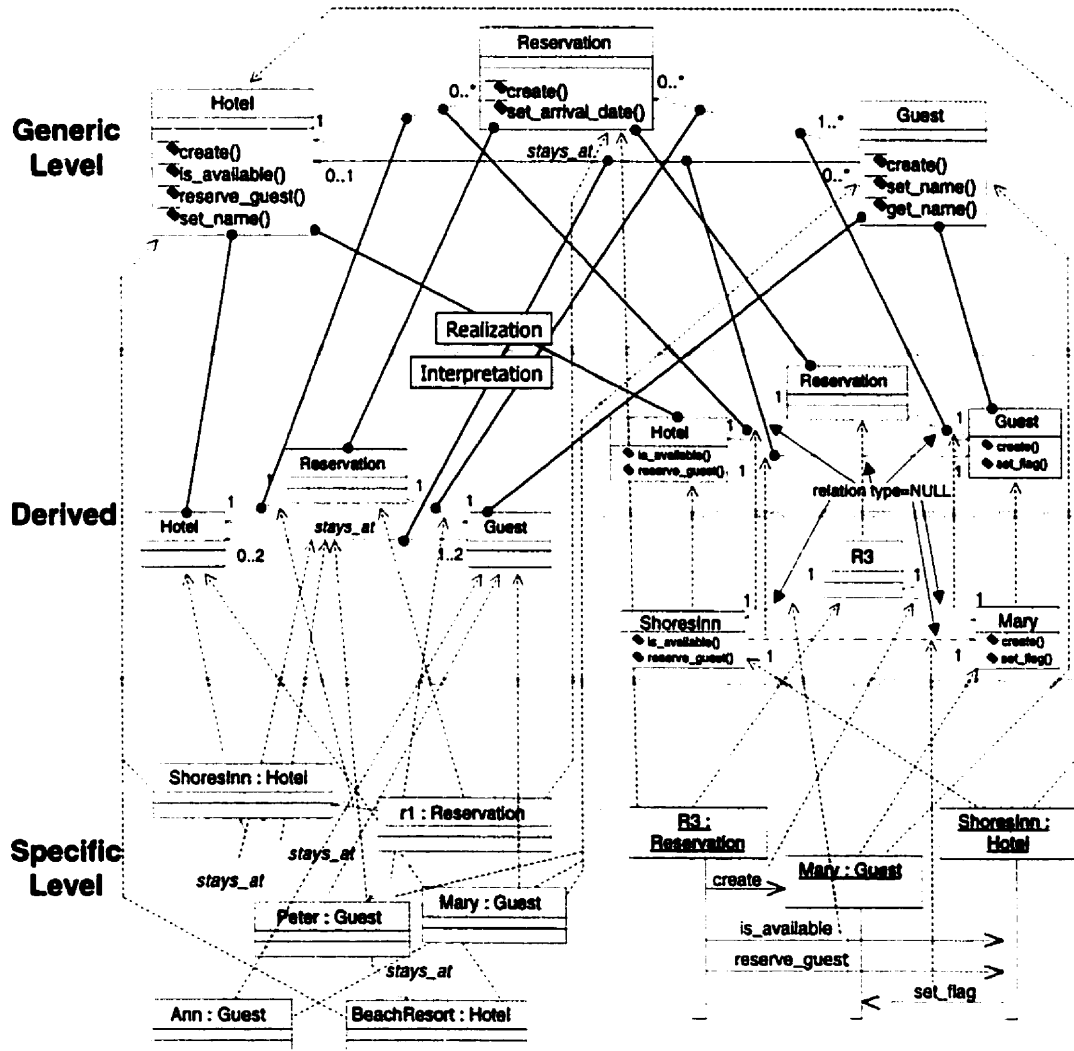


Figure 84. Example of Consistency Checking between Generic and Specific Elements

Figure 84 shows a comparison example between generic and specific diagrams. The figure is analogous to Figure 83. The top area and bottom areas again represent user-defined diagrams. On top, we find a class diagram depicting the generic relationship between *Hotel*, *Guest*, and *Reservation*. On the bottom, we find two specific diagrams depicting instances of the generic scenario. In particular, the bottom left shows an object diagram with the objects *Peter*, *Ann*, and *Mary* which are instances of *Guest*, the objects *ShoresInn* and *BeachResort* which are instances of *Hotel*, and the object *R3* which is an instance of *Reservation*. It is depicted that *Ann* stays currently at the *ShoresInn*, *Peter* currently stays at both hotels, and has a reservation for the *ShoresInn* (e.g., for some later date), and *Mary* has neither a reservation for any hotel nor does she stay at any one of the hotels.

The other specific diagram on the bottom right is a sequence diagram that depicts how a particular instance of *Reservation* (called *R3*) is used to make a reservation for *Mary* at the *ShoresInn*. We see that the reservation object instantiates the *Mary* object (*Mary* had not exist at that time) followed by verifying availability of space (*is_available* method) and reserving space (*reserve_guest* method) at the *ShoresInn*. The *ShoresInn* object calls the *Mary* object using its *set_flag* method (e.g., possibly to indicate that this person has some reservation for some place).

The middle area in Figure 84 depicts derivations of the specific views using some of the generalization techniques discussed in Section 7.3.2. For instance, the object diagram was generalized into a class diagram showing the three classes *Hotel*, *Reservation*, and *Guest*. Additionally, the generalization process was able to derive some cardinality information based in the specific object diagram. For instance, we can observe that the object diagram has the cases of zero, one, or two guests staying at hotels. This implies, on a generic level, that *Guest* may be related to zero-to-two *Hotel* objects. Other cardinality observations can be made in a similar manner. The sequence diagram was also generalized into a class diagram. However, as it was discussed in Section 7.4, no direct transformation between sequence and class diagrams exists. Therefore, our transformation process used complex transformation to generate first an object diagram out of the sequence diagram followed by a generalization of that object diagram into a class diagram.

The generalization of the sequence diagram was not able to reason about the types of relations between the classes *Hotel*, *Reservation*, or *Guest*. Those relations are therefore indicated as dashed arrows (type = NULL). Although the relations are of unknown type, we still were able to indicate the directions of the interactions. Also, the sequence diagram shows method calls that can be structuralized and then generalized into a class diagram. For instance, the observation that the *R3* object uses the *create* method to instantiate the *Mary* object implies that *Mary*'s type, which is *Guest*, must also have a method called *create*. Additional observations could be made about the sequence diagram; however, they are omitted here since they are not relevant for this example.

Since the generalization process requires mapping information, they are specified in form of *generalization* dependencies. Generalization dependencies are depicted as vertical dashed arrows. The generalization arrows between user-defined views and derived views were generated automatically. The generalization arrows between user-defined elements (e.g., between the generic and specific level) were defined manually. Like in Figure 83, interpretation arrows are used to denote elements that are directly comparable. The lines with circular arrows on both ends indicate such interpretations.

Checking for consistency between the generic and specific views is analogous to checking consistency between the abstract and concrete dimension. Consistency checking traverses the model and applies consistency rules whenever applicable. Thus, it traverses the model to find model elements for which the input condition holds and then validates the output condition. The example has a number of inconsistencies:

- Inconsistent cardinality between the class and the object diagram (e.g., between *Hotel* and *Guest*)
- Inconsistent direction of relations between the class and the object diagram (in particular between *Reservation* and *Guest*)
- Inconsistent direction of relations between the class and the sequence diagram (two cases between *Reservation* and *Guest* and between *Guest* and *Hotel*)
- Inconsistent method declaration and usage between the class and the sequence diagram (method *set_flag* is not declared in class diagram but used in sequence diagram)

Based on the examples shown, it can be observed that comparing generic and specific views is very similar to comparing abstract and concrete views. In both cases, diagrams are transformed to enable a direct comparison. In both cases the resulting comparison rules are therefore very much alike. The challenge of comparison is to identify interpretation relationships and comparison modes (e.g., part-of or equivalence). In both cases (generalization and abstraction) the actual comparison is complicated by partial, ambiguous, or multiple interpretations per comparable user-defined element. This section already discussed how to deal with those problems.

The examples of abstraction and generalization showed that consistency checking is greatly simplified through transformation. For instance, identifying the cardinality inconsistency between the generic and specific diagrams can only be done by investigating entire diagrams. This is one reason why simple one-to-one comparisons without prior transformations are often not very powerful (recall Section 7.2).

In the following, we will discuss the issue of how to trigger transformation as part of consistency checking. Since the comparison examples in Figure 83 and Figure 84 showed an awkward overhead in the use of model elements (especially trace arrows), Section 7.7 about repositories will discuss improvements. In particular, evolutionary aspects of view integration greatly depend on improvements made on the repository side.

7.5.3 Triggering Transformation

We discussed the workings of transformation methods in Section 7.3. This section further shows how transformation methods are triggered to enable comparison. Although, we see model transformation as the enabling technology to ease comparison, we still found that transformation remains fairly independent from comparison. Accordingly, we see the need to support two fundamental modes of transformations:

1. Transforming of the entire model prior to its comparison
2. Transforming on a need basis for specific comparison (localized transformation)

The reason for the independence of transformation from comparison is in the interface between them. We already discussed that transformation only generates derived information. In differentiation,

derived information is used to reason about consistency issues. The causal dependency between comparison and transformation is, therefore, only that transformation has to happen before comparison. It does not matter whether transformation was done immediately preceding comparison or whether it was done some longer period of time ago. Similarly, it does not matter whether only a partial transformation was carried out satisfying only a particular consistency checking need, or whether a more complete model transformation was performed. It has no negative influence onto consistency checking if more than required transformations were performed.

There are respective advantages and disadvantages in what transformation mode to choose. It has been our finding that in reality both transformation modes have to be supported. Initially, transformation on a need basis may be sufficient. Since transformation results should never be discarded unless they have become obsolete, a more completely transformed model becomes available over time by default.

There is also another scenario of not wanting to do transformation although their results might have impact onto comparison. For instance, if the model is changed, new inconsistencies could be introduced. Some of those inconsistencies are temporal since they only happen as part of the changing process. For instance, creating a new relation between two classes may require the deletion of the old relation followed by the creation of the new one. The first step of deleting the old relation, however, could cause an inconsistency. Since that inconsistency is immediately resolved once the new relationship is created, it is of little use to identify it. We refer to such an inconsistency with only a short lifetime as a temporal inconsistency. It must be noted that the life-time of a temporal inconsistency may be as short as a minute but could also be as long as days or weeks (e.g., changing larger parts of a model). It may, therefore, not be desirable to do consistency checking on those parts of a model that are in the process of revision. To that end it is not always desirable to *force* transformation prior to comparison. On the other hand, only if transformation is forced immediately prior to comparison are derived interpretation and comparison results up-to-date. The ability to separate evolutionary aspects of software modeling from consistency checking is, therefore, another advantage of using a separate transformation and consistency checking approach. Our view integration model supports three modes of operation:

- 1) Transformation only without comparison,
- 2) Comparison without prior transformation, and
- 3) Comparison with minimal (forced) prior transformation

The first case of transformation only implies the use of our transformation methods as synthesis methods. The results of transformation are not (yet) used for consistency checking. The second case of comparison without prior transformation implies the use of our comparison methods for analysis only. No transformations are performed prior or afterwards and no derived interpretations are generated. In the second case, comparison can only draw from previous transformations results. The third and final case of comparison forces a minimal amount of prior transformation. This last case shows the use of both synthesis and analysis methods together. The synthesis happens prior to analysis to ensure that the model is up to date and that analysis can detect the latest inconsistencies. It is the users choice the select the fitting consistency-checking mode.

The amount of transformation required for consistency checking depends on the selections made. For instance, if consistency needs to be validated between two diagrams then transformation can be limited to those two diagrams. If consistency needs to be validated given only a single diagram then all other diagrams that are transformable into that diagram need to be transformed. It is not very useful, although possible, to validate consistency between individual model elements since it has been our finding that often complete diagrams must be investigated to reason about single elements.

7.5.4 User Interaction

Consistency checking in itself may be seen disjoint from user activities but it is not separate. Consistency checking requires extensive feedback from users during all activities. The most manual activity of our view analysis framework is mapping. Nevertheless, transformation and differentiation cannot always be done fully automatically. Thus, the user at least needs to specify the extent and modes of transformation and comparison. In more advanced cases, the users may also have to address and resolve ambiguities (e.g., during transformation) to get more meaningful comparison results.

Besides consistency checking, view integration also needs to resolve inconsistencies. Two issues are important here: how to resolve inconsistencies and when to resolve inconsistencies. The issue on how

to resolve inconsistencies may be intuitive. The issue of when to resolve it is less so since one would assume that inconsistencies must be resolved right away. Instead, there are good reasons why inconsistencies cannot be resolved immediately. As an example we mentioned temporal inconsistencies previously where a model change can cause inconsistencies that will be resolved sooner or later. Other cases involve incomplete information where it is sometimes not possible to resolve an inconsistency due to the lack of more specific information. There has been extensive works on living with inconsistencies that handle issues like what inconsistencies to search for, how to present them, how to repair them, and when to repair them [Balzer 1991] [Finkelstein et al. 1991] [Nuseibeh 1996].

Like transformation, the differentiation activity is more challenging than just comparing model information. For one, we have to deal with similar issues as in transformation (e.g., reuse and redundancy of comparison results) but we also have to deal with more ergonomic types of decisions since view comparison causes more interactions with the users (humans). Our work does not address human computer interactions (recall Section 4) but the following lists some of the concerns that may arise:

- Ignoring inconsistencies: The user may be aware of some types of inconsistencies and may choose to ignore any feedback on them.
- Show all/show what has not been shown before: The user could choose to be presented with all inconsistencies every time an analysis is performed or the user could choose only to be presented with only the new ones.
- Passive feedback: Instead of requiring the user to acknowledge and/or immediately respond to inconsistencies, the feedback should be more passive, leaving it to the discretion of the user when and how to handle them.
- Prioritizing inconsistency feedback: This can be done by keeping track of the reliability of the transformation techniques used.
- Suppressing multiple feedbacks per modeling element: Since a single defect may be detected in different ways, a detection mechanism may also produce multiple defect reports about single defects which could be compressed.

- Inconsistency detection mechanism should err to the benefit of not indicating inconsistencies although there is one instead of indicating inconsistencies where there are not any. The rationale for that is that otherwise the amount of feedback to the user would exceed practical considerations.

7.5.5 Deferred Issues

Like transformation, differentiation cannot be seen in isolation. The following items will be addressed in later sections:

- Modeling Multiple Derived Results (see Section 7.7)
- Evolutionary Scalability (see Section 7.7)
- Types of traces needed to support comparison (see Section 7.6.2)
- Dummy element for derived interpretations that do not have user-defined counterparts (see Section 7.7)

7.6 Model Synthesis and Mapping

Science and engineering alike stress the importance of being able to produce and reproduce data. The production of a software system involves the creation of modeling information that either specifies the system itself or assists during the decision making process. A general technique for enabling reproduction is tracing ones steps from inception to conclusion. If done properly, tracing will outline every step along the way of how a problem was transformed into a solution, including intermediate results and findings. This section address model synthesis (the production) and model mapping (the reproduction).

7.6.1 Model Synthesis

Most model information for software can be categorized into two major categories: (1) Information that is relevant for the construction of a software system; and (2) information primarily needed for decision making along the way. This work primarily emphasizes the former—the specification of system relevant information. Creating model elements is mostly a manual activity performed by a single or by multiple users. The issue of how to develop software models using UML has been discussed in great detail in works like [Rumbaugh et al. 1999], [Booch et al. 1999], [Jacobson et al. 1999], [Fowler

1997], [Siegfried 1996], [Kruchten 1998], [Carmichael 1994], [Eliens 1995], [Magee and Kramer 1999], and [Wirfs-Brock et al. 1990]. It is outside the scope of this work to provide development recommendations.

Our tool, UML/Analyzer, uses Rational Rose® as a synthesis tool. Rational Rose is used to create, modify, and delete elements of the UML model. Our tool also uses Rational Rose to visualize transformation and consistency checking results (see Section 8).

7.6.2 Model Mapping

Mapping identifies relationships between modeling information of different views. Mapping therefore describes overlapping and often redundant pieces of information. We already discussed the importance of mapping for view integration and discovered that mapping frequently has to be done manually and can (if done properly) significantly improve scalability and reliability. Mapping can be supported via naming dictionaries or traceability matrices [Gieszl 1992] [Gotel and Finkelstein 1994].

7.6.2.1 Traceability Types

For automated transformation and differentiation we identified a number of tracability types that need to be supported. Only some of those are actually used by the users most others are represented by our consistency checking framework:

- Abstraction traces: to link abstract and concrete diagrams (e.g., high-level and low-level diagrams)
- Structuralization traces: to link structural and behavioral diagrams (e.g., statechart and class diagrams)
- Generalization traces: to link generic and specific diagrams (e.g., sequence and class diagrams)
- Interpretation/realization traces: to identify derived model elements that are directly comparable with user-defined model elements (for the most part transformed elements)
- Origin traces: to link derived model element to their original user-defined elements that served as input to the transformation process

7.6.2.2 Mapping Support

Mapping can be supported and partially automated and validated using analysis patterns, shared interfaces, and inter-view dependency traces. If done manually, mapping may result in an additional source of possible defects in that potentially two views are related that actually do not relate to one another or vice versa. Mapping also increases the manual overhead in applying integration techniques. The goal is therefore to have some automated mapping technology. As it was outlined in Section 4 that this work does not aim at discussing how to automate mapping. The reason for this limitation is that mapping is a very complex problem in its own rights. Not discussing automated mapping does, however, not mean that no automated support for traceability is available.

7.6.3 Deferred Issues

Mapping, like transformation, cannot be seen in isolation in context of our view integration framework. The following issues are deferred to the next section:

- Implementation of tracability types (see Section 7.7)
- Scalability Issues due to trace explosion (see Section 7.7)

7.7 Model Repository

The model repository is the central database for model elements. It provides both storage space and access points. In our view integration framework in Figure 42, the model repository was depicted outside of the view analysis component, and is, therefore, not discussed in every detail in this work. The main reason for that is that we chose a modeling language that is already predefined (e.g., UML). UML is supported by a meta-model that describes the definitions and interactions of model elements in detail (recall Section 5.3). For more information about UML's meta model, please refer to [OMG 1999]. The major reason why we do have a model repository section in this work is to define the interfaces between our analysis component and the repository as well as suggest improvements and extensions to allow a better and more scalable consistency handling.

7.7.1 Implementing View Integration Elements

For the most part, transformation does not require specific model elements. For instance, an abstraction of a class diagram yields another class diagram that in turn can be model in the same manner as the first one. To distinguish between user-defined and derived information, derived elements are annotated with the stereotype «derived».

UML 1.3 supports abstraction through the *dependency* relationship stereotyped as «abstraction». Similarly, UML supports some forms of generalizations. In UML, the supported generalization relationship, however, only applies to certain types of model elements and not to all (in theory every specific model element could be generalized). We, therefore, decided to create our own version of the generalization dependency which is modeled like abstraction in form of the *dependency* relation UML provides (annotated as «generalization»).

Origin traces link derived information to their user-defined ones. Since derived information may be based on intermediate derived information, the abstraction, generalization, and structuralization links are often not adequate in determining the original user-defined elements that enabled the transformation of a particular derived element. The origin trace, always a transitive link between derived and user-defined information, is also modeled via *dependency* relations. Figure 85 depicts origin traces in context

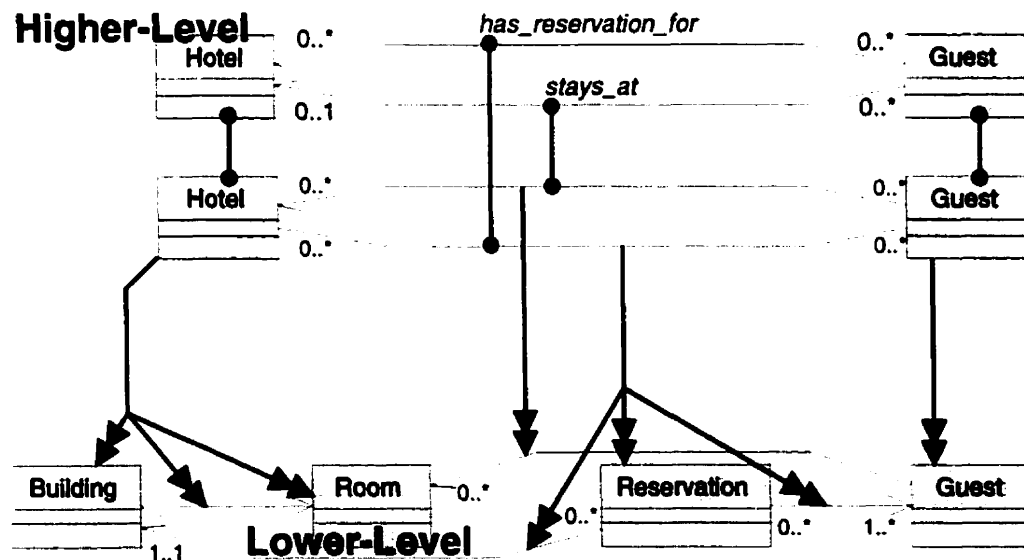


Figure 85. Origin Traces and Interpretations Traces

of the example from Figure 83 (note that we did not display origin traces in Figure 83 to reduce confusion). Storage wise, an origin trace is simply a shortcut.

7.7.2 Evolutionary Scalability Problem

Transformation improves reuse and has also a considerable positive impact onto scalability (reuse). Reusing transformation results also improves evolutionary consistency checking during later cycles. It must be noted that previous examples only described single consistency checking tasks at a single time (a brief moment during the software life-cycle). Since consistency checking cycles may to be repeated frequently during longer projects, the higher- and lower level class diagrams in Figure 81 might have to be validated multiple times (at least once per cycle). Having transformation results from past review cycles can, therefore, simplify the next review, again through reuse. For instance, if neither the lower-level diagram nor the higher-level diagrams were changed, then there would be no need for a re-evaluation. Remembering past consistency checking results, however, also implies keeping track of transformation results for a longer period of time. This latter aspect introduces a new scalability problem.

Assume that our hotel management system discussed before has grown to a large model and now contains 50 (user-defined) diagrams. Using transformation, we can easily come up with hundreds if not thousands derived views (intermediate views) using various transformation combinations. Assume that we now modify a model element in one of the user views. As a consequence, we would have to make sure this change is properly propagated to the other 49 user-defined views. Additionally, and this is the problem, we might also have to update *all* derived views (the intermediate views we had generated during transformation) since they might have become inconsistent as well. The latter would cause an enormous diseconomy of scale (recall the n^2 complexity challenge we had discussed previously). In case of our example in Figure 82, if we change the lower-most diagram (most concrete view) then both abstractions might get affected by that change and may be in need of updating. Figure 86 depicts this problem schematically. It shows that for a couple of user-defined views, a series of derived views may be generated. If a change is introduced, a large number of consistency checking activities have to be performed.

An easy solution to this problem is the creation of derived views (interpretations) on a need basis and only for individual review cycles. For instance, the abstractions in Figure 82 could be created to support the comparison with the higher-level diagram in Figure 81. After completion of that review cycle, both derived abstractions could be deleted.

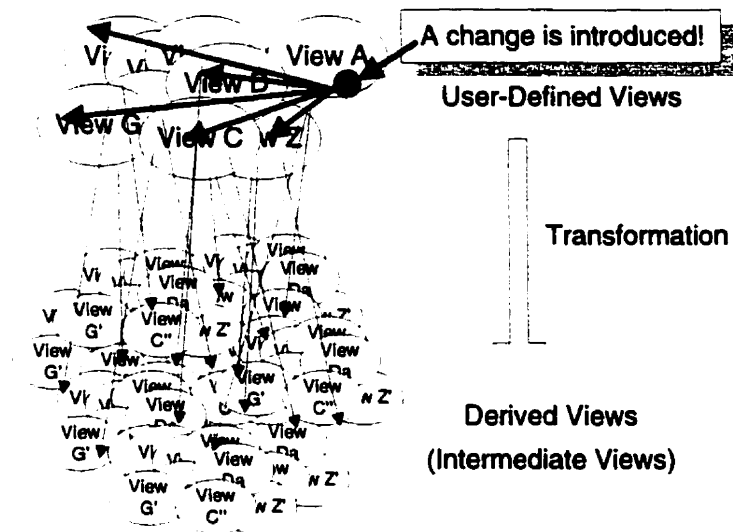


Figure 86. Evolutionary Complexity

Deletion would indeed solve the evolutionary scalability problem, however, require a lot of transformation overhead otherwise. Two reasons add to that overhead: (1) the same transformations must be repeated multiple times during evolution; and (2) user feedback as part of transformation results might get lost. Recall Section 7.5.4 where we discussed that there is a benefit in separating transformation from consistency checking. That benefit would be taken away if we would delete all derived model elements after usages.

A simple solution to above problem would be the localized deletion of modeling information. Instead of deleting derived information after every review cycle, that information is kept in the repository for future reuse. If a change in the model causes derived information to become inconsistent then a localized deletion is performed that eliminates obsolete elements. This solution improves the situation but is still not ideal since lots of changes cause the deletion of a large body of derived information. We will discuss this problem next.

7.7.3 Reduced Redundancy Model

The concept of a reduced redundancy model is a way of handling derived modeling information without having to worry about keeping them consistent. A reduced redundancy model is an internal representation of diagrams that tries to minimize redundancy. The problem we have with derived model

elements is that they exhibit a high degree of redundancy; or to be precise they are completely redundant since otherwise an automated transformation method would not be able to create them. Figure 82 shows substantial redundancies in that some model elements are replicated twice or even three times (e.g., *Guest*). In larger examples, the amount of replication is even more severe.

Previously, we discussed that redundancies are a major cause for inconsistencies. Due to the high-degree of redundancies in derived views, the number of inconsistencies that can occur rises significantly. Additionally, the high-degree of redundancy results in exponential increase of storage space. In a worst-case scenario, each type of model element may potentially be transformed into *all* other types, already causing a strong overhead. However, since those derived elements are again transformable into all other types (recall complex transformation in Section (1)), the storage overhead could explode. Instead of storing “n” elements, we might have to store $n*(n-1)*(n-2)*(n-3)...$ elements. This case is obviously a worst case scenario and the reality is by far not as bad, however, it is still a major concern.

A reduced redundancy model is a compression method for modeling information with the added benefit that inconsistencies are less likely to occur. The fact that space for storing modeling information is reduced in the process is another positive side effect. The reduced redundancy model is an important concept in improving scalability for UML consistency checking. It allows derived information to be stored without having to worry about their maintenance.

Figure 87 shows a reduced redundancy model of our example from Figure 82 using UML constructs. The figure depicts the same classes and relations as Figure 82 but uses less replication. Thus, instead of three instances of *Guest* (in Figure 82), Figure 87 only uses one instance. The savings are considerable: The class diagrams in Figure 82 used 9 classes, 9 relations, and 14 trace links (the latter not depicted in the figure). The more compressed class diagram in Figure 87 only uses 5 classes, 8 relations, and

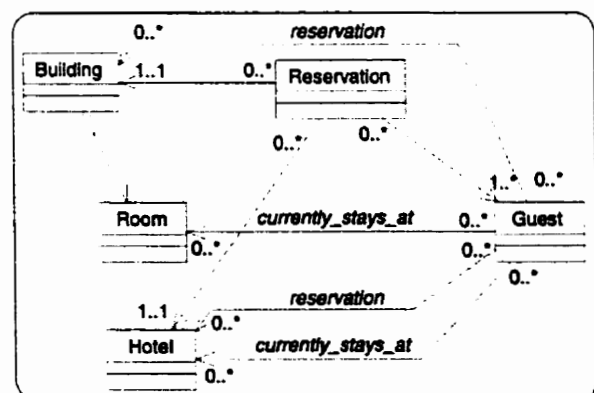


Figure 87. Reducing Model Redundancy using UML

8 trace links. Figure 87 is, therefore, a compression of Figure 82 in that it uses less model elements to describe the same interdependencies. Keep in mind that a reduced redundancy model improves the internal representation of modeling information. A user could still derive each diagram depicted in Figure 82 out of the reduced redundancy model (we will discuss that later).

As we discussed above, the advantage of a reduced redundancy model is reduced storage space and less consistency checking between user-defined and derived model elements. As an example of less consistency checking overhead, consider the following: If the name of the class *Guest* is changed to *Patron* in Figure 87, then this change causes no inconsistency in the process. On the other hand, a change like that in one of the class diagrams in Figure 82 would cause an immediate inconsistency with the other two diagrams. In Figure 87, this type of change does even require updates but instead the change is implicitly updated.

The reduced redundancy model is also an example where the UML definition is brought to its limits since the situation depicted in Figure 87 is not yet perfect. On the one side, there is still redundancy as in the case of the *currently_stays_at* and *reservation* relations. On the other side, the interfaces of classes become askew. For instance, the user-defined class *Building* has only two relations in Figure 82 but three relations in Figure 87. Those deficiencies cannot be addressed within the boundaries of the UML standard. Instead, we found that we had to introduce new concepts that go beyond UML.

Figure 88 depicts an even less redundant model of the problem above. This version improves the previous one since it only needs 5 classes, 5 relations, and 6 trace links. Its creation, however, requires the concept of a *bridge* that links model elements together (both classes and relations). Bridges are depicted as black circles where classes and relations are attached. Figure 88 completely eliminates both problems we identified with respect to Figure 87. First, the redundant relations are eliminated and, second, the classes only see their true

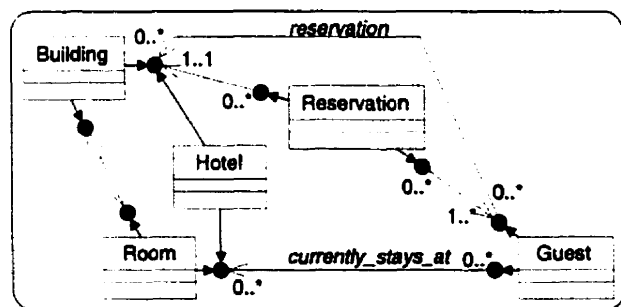


Figure 88. Reducing Model Redundancy outside UML

number of relations. For instance, the class *Building* now has exactly two bridges to relations. Also, building consistency rules for Figure 87 is much harder than building consistency rules for Figure 88. Additionally, the amount of computing effort required to navigate a more redundant model (as in Figure 87) is also much higher than navigating the less redundant model in Figure 88. The compactness of our reduced redundancy model, therefore, results in a higher accessibility of its model elements.

The only disadvantage of the reduced redundancy model in Figure 88 is that it cannot be supported in UML. UML's meta model has a clear definition on how model elements have to interrelate and the concept of a bridge is not supported. Bridges are located between model elements and each bridge may have two or more attachment placeholders for model elements (ports). At least two ports are needed since a bridge is used to link at least two model elements.

7.7.3.1 Reduced Redundancy Model for Class Diagrams

Since the emphasis of our work is geared towards class diagrams, this section discusses both their creation and access in context of the reduced redundancy model. The specification of a bridge is analogous to the specification of an *AssociationEnd* in its role to mitigate between *Association* and *Classifier* (see [OMG 1999]). The usage of bridges during model synthesis is also analogous to that of *AssociationEnd*. If a model element is derived (e.g., abstracted) then the derived element is attached to all external bridges it absorbs. Figure 88 shows this in the context of the *Hotel* classifier and the *reservation* and *currently_stays_at* relations. *Hotel* absorbs the classifiers *Building* and *Room* as well as their internal relation. Externally, two bridges remain to the relations that lead to *Guest* and *Reservation*. The major remaining issue is how to handle the access to a reduced redundancy model. In particular, is it possible to generate the original user-defined and derived views out of the reduced redundancy model? If yes, then this would imply that the original representations remained intact and there would be no negative side effect in using reduced redundancy models.

Figure 89 shows a generic example of three class diagrams on different levels of abstraction. The lower-most layer contains the original view (e.g., detailed structural/behavioral view) and depicts four classifiers (A-D) and four relations between them (α to δ). The middle layer groups the classifier B and C

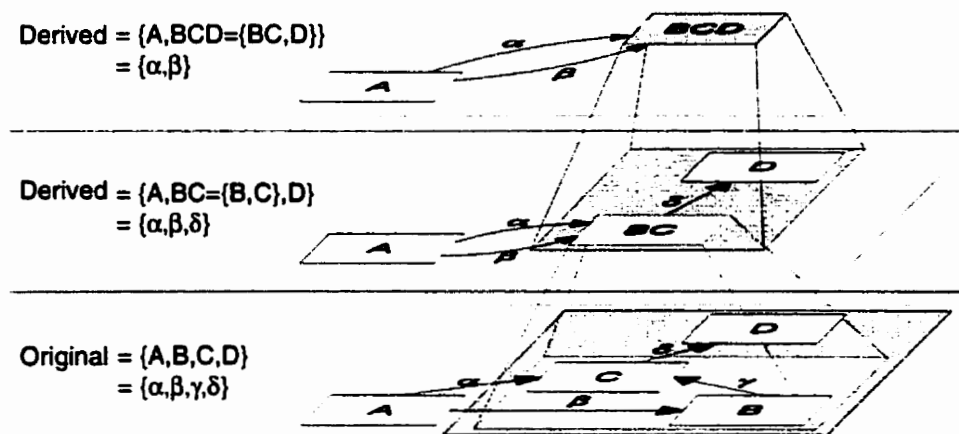


Figure 89: Generic Example of Classifier Abstraction

into a composite classifier named BC. The relation γ as well as the classifiers B and C are absorbed into the composite classifier BC and not visible any more. The third and top-most layer further abstracts by adding another composite classifier BCD which contains the composite classifier BC from the middle layer as well as the original classifier D from the lower-most layer.

The right hand-side of Figure 89 additionally depicts identifiers for accessing user-defined and derived views. Those identifiers are built out of classifiers or relations and are stored as part of view descriptors (a view descriptor describes information that is stored explicitly about views). If the user-defined view on the bottom needs to be reconstructed out of the reduced redundancy model then either the descriptor $\{A,B,C,D\}$ or the descriptor $\{\alpha,\beta,\gamma,\delta\}$ can be used. We refer to those descriptors as the identifiers of views and, as it can be seen, there are two types of identifiers that could be used for class diagrams—one consists of classifiers and the other consists of relations (hybrids are possible too but not explored here). Derived views can be accessed as well. For instance, the middle layer can be accessed using either one of the following three forms: $\{A,BC,D\}$, $\{A,\{B,C\},D\}$, $\{A,BC=\{B,C\},D\}$ where the last form is the most complete one.

Identifiers can also be based on relations. Using relations as identifiers may seem counter-intuitive at first glance since people often find classifiers to be the more dominant design features. In fact, for classifier abstraction it even turns out that identifiers based on relations are ambiguous. We found,

however, that during relation abstraction in the next section, it is the relations-based identifiers and not the classifier-based ones that yield unambiguous identifiers. Thus, both types are needed to allow seamless navigation between them.

Before we can discuss how to eliminate the ambiguity in identifiers, we will describe how to restore and access views in cases where no ambiguities are present. Figure 90 (upper-left) shows the reduced redundancy model for all the three layers in Figure 89 (Figure 90 is alike Figure 88). Besides the discussed scalability and storage improvements, the minimal redundancy model also has the added advantage that model information as well as their abstractions (and other possible transformations) are stored and accessed at a single location.

Using the classifier identifiers, the process of re-creating a view out of a reduced redundancy model is straightforward. We only need to take each classifier permutation and find all (direct) relations between them. For $\{A,B,C,D\}$, possible permutations are $\{A,B\}$, $\{A,C\}$, $\{A,D\}$, $\{B,C\}$, $\{B,D\}$, and $\{C,D\}$. Between $\{A\}$ and $\{B\}$ exactly one relation β exists. If this process is repeated, the remaining

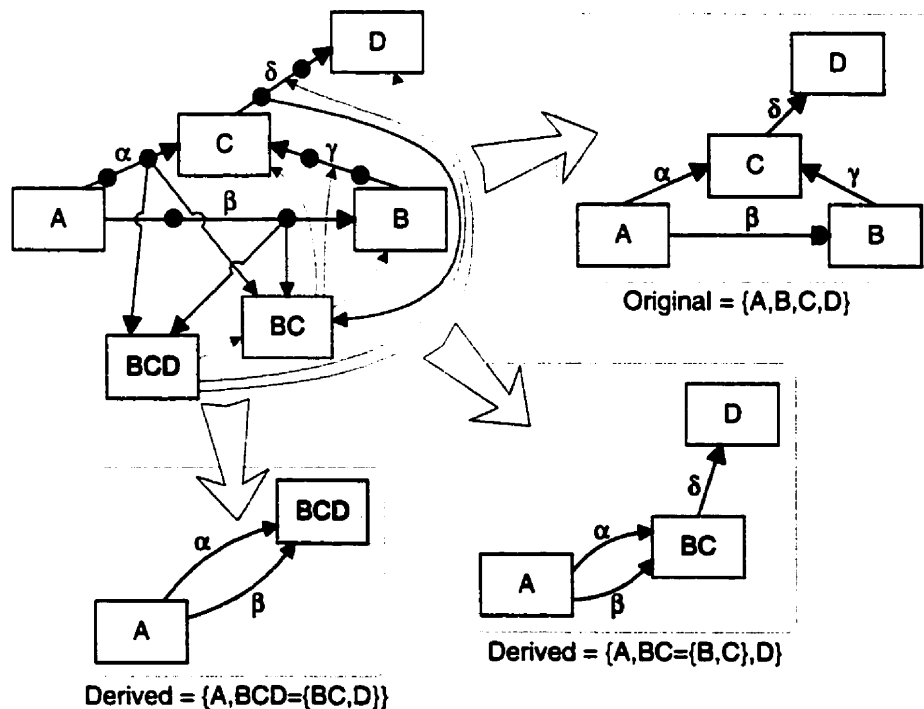


Figure 90: UML-A Model satisfying Classifiers for Multiple Abstractions

relations are found. The same process can be followed in case of identifier $\{A, BCD\}$ which yields the two relations α and β .

Since the reduced redundancy model plus the identifiers are sufficient in reconstructing all three diagrams in Figure 89, they can be deleted. After all, they can be re-constructed on a need bases at a later stage (e.g., if a user would like to modify one of them). In terms of computations, the re-construction only has a n^2 complexity where n represents the number of model elements in the identifier. Since the reduced redundancy model is only an internal representation, we refer to diagrams that are constructed out of it as *projections*. It must be noted that projections are not labeled *derived* in order to not confuse them with user-defined/derived model elements in the previous chapters.

We indicated above that identifiers based on classifiers are unambiguous, however, identifies based on relations are not. We investigated the issue and found a solution. The problem is depicted in Figure 91 which shows the ambiguity in using relations as identifiers for classifier abstractions (we use the same example as before). The figure shows that the identifier $\{\alpha, \beta, \delta\}$ of our example in Figure 90 will yield two possible projections. The left projection is based on the original view whereas the right one is based on the derived view. In order to find an unique identifier, we have to first decide what it is that we want to achieve. In view integration, when we query for information related to some modeling artifacts then one of the following two scenarios are most likely:

- Query is based on original modeling information for user-enabled manipulation
- Query is based on the minimal derived modeling information for consistency checking

Using the additional information about usage, the relation identifier becomes unique. The left diagram in Figure 91 shows the result of the query based on the original modeling information (scenario

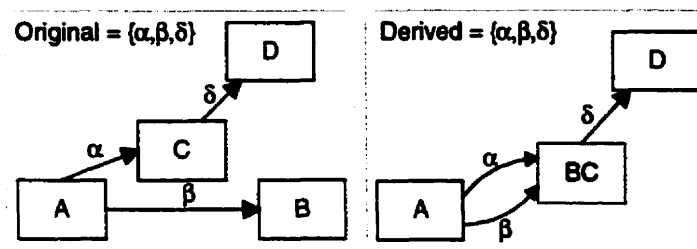


Figure 91: Ambiguity in Accessing Composite Classifiers via Relational Names

```

let relations = input
let classifier = {}
let paths = {}
(A)  $\forall$  from  $\in$  relations,  $\forall$  to  $\in$  relations and from  $\neq$  to
    paths  $\rightarrow$  put (from  $\rightarrow$  find-path(to))
(B)  $\forall$  p  $\in$  paths
    p = abstract(p)
    classifier  $\rightarrow$  union(p  $\rightarrow$  relations  $\rightarrow$  classifiers())
(C)  $\forall$  from  $\in$  classifier,  $\forall$  to  $\in$  classifier and from  $\neq$  to
    if (to  $\in$  from  $\rightarrow$  abstractions()) classifier  $\rightarrow$  remove(from)

```

Figure 92: Deriving correct Projection from Relation Identifier

(a)). Since the original view does not contain derived interpretations, the process of creating a user-defined view is simple (original approach applied to user-defined elements only). The second scenario is a bit more difficult to understand but yields a correct result as well. The procedure is as follows:

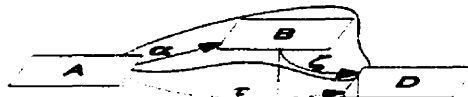
The *relations* variable in Figure 92 is the input to the procedure. If we follow our previous example, we need to set *relations* = { α , β , δ }. Part (A) then computes all possible paths between those relations which yields *paths* = {{ α , C, γ , B, β }, { α , C, δ }, { β , B, γ , C, δ }, { α , A, β }}. Note that the path { α , A, β , B, γ , C, δ } is not valid since this path uses β , one of the input relations.

Part (B) takes each individual path, searches for its abstraction(s) and stores classifiers found in the variable *classifiers*. The role of the *abstract* function is to take a path and collapse its structure in such a way that the final path is of the simple form {relation, classifier, relation}. In our example, the { α , C, δ } path is already of that form. What remains is to abstract the remaining three paths that are still more complex. The abstraction procedure takes each path (e.g., { α , C, γ , B, β }), queries whether there is already

Derived = {A,D}
= { ω ={ α , ζ }, β }



Derived = {A,B,D}
= { α , τ ={ β , δ }, ζ ={ γ , δ }}



Original = {A,B,C,D}
= { α , β , γ , δ }

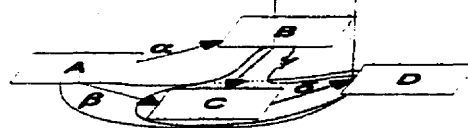


Figure 93. Generic Example of Relation Abstraction

an existing composite classifier (e.g., as it is for $\{C, \gamma, B\}$ —note that this query is based on $\{BC\}$ which is a unique identifier). If yes, then the existing classifier is returned, otherwise, the pattern $\{C, \gamma, B\}$ is collapsed into the composite classifier BC , and $\{\alpha, BC, \beta\}$ is returned. The same procedure is repeated for all other paths. Part (B) also gathers all (user-defined) classifiers involved in abstracted paths (now $\{\{\alpha, BC, \beta\}, \{\alpha, C, \delta\}, \{\beta, BC, \delta\}, \{\alpha, A, \beta\}\}$). Thus, the variable *classifiers* will end up containing $\{BC, A, B, C, D\}$ (duplicate classifiers are eliminated. Finally, Part (C) eliminates all those classifiers that already have abstractions of them in the list. In our example, B and C are refinements of BC and are removed. The remaining list of classifiers is $\{A, BC, D\}$. This result conforms to the right hand side of Figure 91 and can now be used as an unambiguous identifier.

The same process can be repeated for relation abstraction. To that end, Figure 93 shows an example of a simple view that is abstracted using again several layers of abstraction. The original view contains four classes A , B , C , and D as well as four relations α , β , γ , and δ . The $\beta \rightarrow C \rightarrow \delta$ pattern was abstracted in the middle layer by introducing a composite relation τ and, similarly, $\gamma \rightarrow C \rightarrow \delta$ is abstracted into ζ . The third layer takes α and B from the original layer as well as ζ from the middle layer and further abstracts that pattern in the composite relation ω . The left hand-side of Figure 93 shows the identifiers for composite relations.

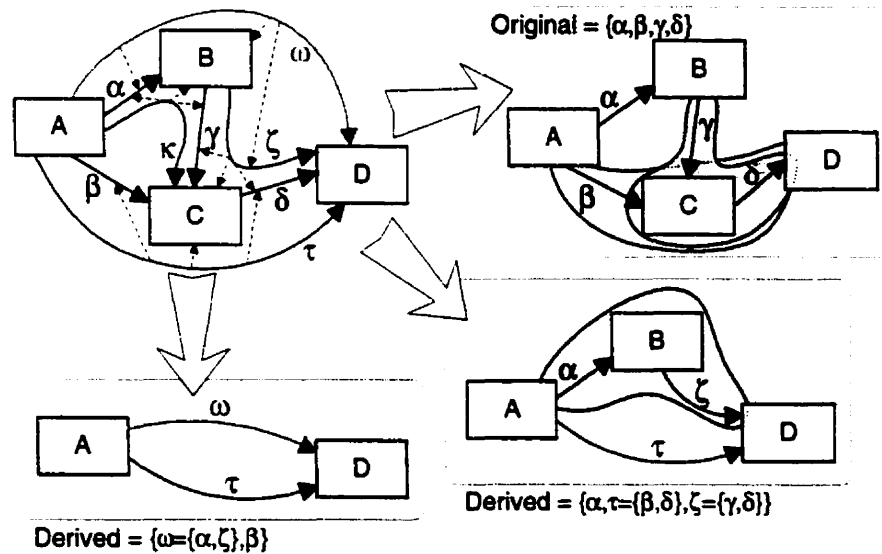


Figure 94: UML-A Model satisfying Relations for Multiple Abstractions

Figure 94 shows the reduced redundancy model for Figure 93. As discussed in classifier abstraction, the goal of the model is to exhibit as little redundancy as possible. This is again achieved by overlaying all model elements (original and derived ones) onto the same structure (see upper left part of Figure 94). Composite relations are linked to the original classifiers since they do not get modified in the process. This structure is analogous to the one given in classifier abstraction. The process of projecting views out of it is therefore also analogous.

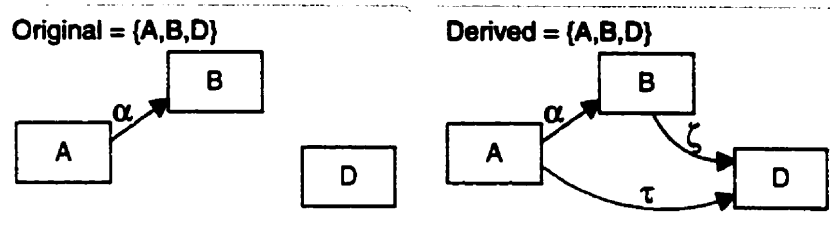


Figure 95: Ambiguity in Accessing Composite Relations via Classifier Names

The ambiguity that accompanies some forms of identifiers exists here too. However, in this case it are the classifier-based identifiers (e.g., {A,B,D}) which are ambiguous. Figure 95 shows examples of ambiguous results based on the identifier {A,B,D} which results in two potential projections. The solution to this problem is identical to the one given in classifier abstraction instead that the roles of relations and classifiers are reversed. The left hand-side of Figure 95 shows the projection of {A,B,D} onto the original model, the right hand-side shows the most abstracted projection that combines the original and derived modeling information. The latter projection is again derived by finding all paths between A, B, and D, collapsing those paths using the rule set defined below, and eliminating all refinements of selected composite relations.

The last step, the elimination of refinements (Part (C) of our algorithm in Figure 92), may seem however unnecessary. For composite classifiers, this step was necessary since a single classifier may have one, two, or more relations. On the other hand, a single relation was thus far only depicted as being exactly between two classifiers (or one in case of a circular relation). Although, relations are indeed used mostly in a dual fashion, there are exceptions. For instance, associations in UML class diagrams are allowed to link more than two classifiers. Figure 96 shows such a relation. Here the relation α links three

classifiers A, B, and C. If a composite relation δ is introduced that collapses $\alpha \rightarrow C \rightarrow \beta$. The search for a projection for {A,B} would yield α as well as δ where α is a refinement of δ . Thus, Part (C) of the procedure discussed in Figure 92 is still necessary to eliminate α from the result.

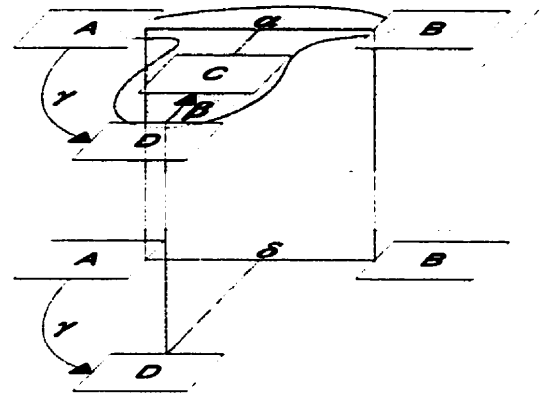


Figure 96: Special Case of Relations using Associations

7.7.3.2 Reduced Model Redundancy and UML

The reader may wonder why we do not extend the concept of a reduced redundancy model to all views UML has to offer including its meta-model. Primarily the reason is that reduced redundancy models have the strongest benefits in grossly redundant cases. Derived views are completely redundant with user-defined ones (we discussed this beforehand). Given the degree of redundancy, the noticeable impact of compressing the model is thus much stronger. Also, since UML does not specify how to store and maintain derived information, integrating those into the existing UML meta-model is more doable without major implications on the standard.

Nevertheless, providing a less redundant model base for all of UML would indeed be a very effective solution overall. In [Egyed and Hilliard 2000], we explore this option in context of architecture description languages. There we argue that in order to address view integration, we need to consider the classification of views first. Basically we distinguish between *Fixed Views* and *View-Independent Models*: Fixed views are stand-alone and support their interaction with other views through some explicitly defined form of data and/or control integration. Most (if not all) architectural definition languages as well as UML fall into the category of fixed views. View-independent models, on the other hand, incorporate all relevant information about multiple views under one common roof with the advantage that information must not be exchanged or updated explicitly.

The trade-off is between them is expressivity and consistency: If one accepts the idea that the drivers of architecting are stakeholders and their concerns, then view-independent models make the assumption that all stakeholders' concerns can be uniformly captured in a single representational scheme. If they can, then having a single model greatly simplifies the "integration problem." Although comprehensive models have emerged in the recent past, these models do not qualify as being view-independent. Take for instance UML which incorporates a number of views such as class diagrams, state diagrams, and sequence diagrams. Even though UML incorporates these views under one common meta-model, the actual storage does not exclude redundancy. It does not violate the UML notation to create two classes with a defined relationship between them and have their instances (objects) contain contradictory relationships. What the UML meta-model has achieved is not view integration but only view representation under a common roof [Egyed and Medvidovic 1999].

On the other hand, requiring a view-independent representation of having no (or only minimal) model redundancy is somewhat of a utopia, however a desirable one since it promises working with multiple views without having to deal with consistency issues. The classification from fixed views to view-independent models is not a discrete one but continuous and manifests itself through three major stances:

- **Constructive stance:** Development views of systems are individually constructed. To understand the model is to understand the sum of all its views and their relationships.
- **Projective stance:** Views are projected out of the model to allow its inspection and/or manipulation. The model itself is all-comprehensive and views are needed to extract the *essence* of particular concerns (like reduced redundancy model extended to all of UML).
- **Decorative stance:** Development views may be partially constructive and projective. Here a base representation exists that is annotated with additional information supporting a limited form of view projection.

Whereas the constructive and projective stances represent the extremes of view integration (none or full), the decorative stance takes the middle ground. Our work has shown that a decorative integration approach can be used even if the models and views were originally designed for a constructive stance (e.g., as in

case of most ADLs and UML). Furthermore, the decorative stance can support notions discussed in [Finkelstein et al. 1991] where it is argued that some amount of inconsistency cannot always be avoided.

The advantage of a (fully) reduced redundancy model for view integration is a simplification of integration work. A reduced redundancy model covering not only derived element but also user-defined ones could be used as a reference model. In an ideal case that reference model would exhibit no view redundancy and it would reduce the view integration complexity to a linear problem as depicted in Figure 97. With the existence of a reference model, the integration work could be reduced to translating each view so that it is fully (or sufficiently) represented in that reference model. Then we could define consistency and completeness rules based on the reference model. Thus, each view only needs to be translated once and all consistency and completeness rules needed only to be represented in one type of style (language, etc.) and not in a view-dependent form (as it is currently). A reduced redundancy model is a simple approximation of such a reference model, however, much more work is needed.

7.7.4 Purging

Using a reduced redundancy model, a change in a user-defined view is implicitly updated on all derived views. A fully reduced (thus minimal) redundancy model could therefore eliminate the consistency problem between user-defined and derived views altogether. However, it is very hard to realize such an integrated and minimal model. We found that it is possible to achieve partial view-independence but a complete one is often unrealistic. This is the reason why we referred to our model as a

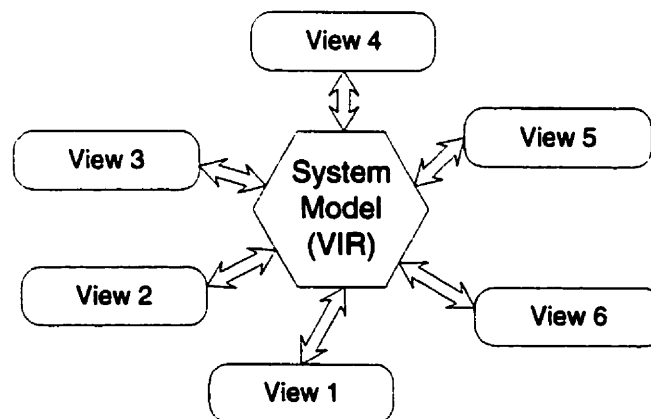


Figure 97: Linear Integration Work using an Integrated Repository

reduced redundancy model instead of a minimal one. In cases where view-independent representations fails, affected (and possibly inconsistent) derived modeling elements must be deleted (purged). The purging activity traverses the model and removes all derived elements that might have been affected by an initial change. This activity, although not trivial, results in only a *partial* deletion of derived modeling information and can be done without consistency checking. Here it comes to our advantage that transformation creates extensive trace links. Purging makes extensive use of those traces and principally works as follows:

```
Purging (input: changed model element)
```

```
1) Is any derived information affected by it?
```

```
    If yes then recursively call purging for those derivations
```

```
2) Delete changed model element
```

7.8 Summary

This section discussed the details of our view integration approach. First, we talked about transformation as an enabling technology for consistency checking. Transformation simplifies comparison (consistency checking) by converting model elements of different type and dimension in such a manner that they become easier comparable. We discussed simple transformation techniques and we discussed complex ones to extend the scope of the simple ones.

This section also discussed the problem of consistency checking after transformation. We first showed various situations that required different comparison modes. We then showed that the underlying consistency checking framework needed to address problems dealing with multiple derived interpretations and multiple realizations. Also we showed that transformation results must be considered with care in that they may contains ambiguities. Consistency checking was shown to be driven by comparison rules that specify input and output conditions. The input conditions defined where the rule is applicable (e.g., what model elements) and the output conditions defined what has to be satisfied for consistency to be ensured.

The problems of model synthesis and mapping were briefly discussed since they are important aspects of modeling and consistency checking. Both are however out of the scope of this work since they have to be done mostly manually by users. Model synthesis deals with the creation of architectural and design diagrams. Mapping further provides the inter-dependencies between those diagrams. The quality of model synthesis and mapping has a significant impact onto view analysis (e.g., reliabilities).

Finally, this section discussed issues on how to store and access modeling information in UML. As such, we showed that transformation reuse improves evolutionary consistency checking but we also showed that our technique causes another scalability problem. We therefore introduced a reduced redundancy model as a way of handling derived information without having to keep them consistent. To allow reduced redundancy models to be used analogous to regular models we showed how regular models could be projected out of reduced redundancy models.

8 Case Study

In this chapter we will apply our consistency-checking framework on a complex UML model. The problem model addresses the design and refinement of a *Hotel Management System* (HMS) dealing with reservation, counter, and accounting services. The services reflect the following needs:

- Reservation Services is used by clerks (employees) to make reservations for potential guests. Reservations may be made for any one of the participating hotels.
- Counter Services is used by clerks for check-in and check-out activities as well as for basic payment and expense handling (e.g., room fees).
- Accounting Service is used for handling and maintaining monetary issues related to guest activities in general. Accounting Services deals with issues like overdue fees or late charges.

8.1 Architecture Level

Figure 98 depicts the high-level architectural view of the HMS. The major components are the *ApplicationPackage*, *DialogPackage*, *ServicePackage*, *AccessPackage*, and *DataPackage*. Clerks, who are employees of the hotel chain, use the *ApplicationPackage* to access the three applications for reservation, counter, and accounting services. All three applications make use of the same *DialogPackage* to display information to the users (as well as for user input). The *ServicePackage* implements the business logic of the applications and offers services such as *make reservation*, *find guest*, or *make payment*. The *DataPackage* defines business objects (e.g., *Guest*, *Hotel*, *Transaction*, or *Room*) used by other packages and, finally, the *AccessPackage* is a front-end (wrapper) to a centralized database. All packages are executed on the local machines the clerks are using. The *AccessPackage*, therefore, is also a front-end for network services to a distant database.

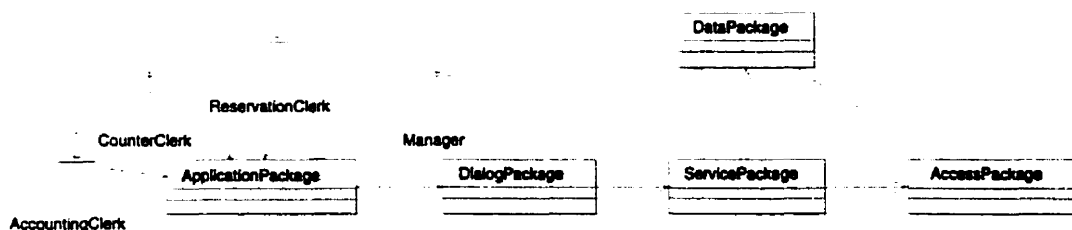


Figure 98. Architecture Overview of HMS

In the following, we will gradually reveal more details about the HMS refining it over two additional levels, the high-level design and the low-level design, using a variety of diagrammatic views from class diagrams via object and sequence diagrams to statechart diagrams. Since the model is very complex only a partial consistency checking study will be shown here. Our emphasis is the coverage of multiple consistency checking scenarios involving at least two examples for each transformation type.

8.2 Refinement to Higher-Level Design

In the course of constructing the HMS, the architectural view from section 8.1 was refined twice. The following discusses the first refinement called the *High-Level Design*, which contains a series of class, object, sequence, and statechart diagrams.

8.2.1 Overview

The class view of the *high-level design* is far more detailed than the equivalent one in the *architectural level*. To display it in a single figure would make it too complex. In the following, multiple figures are used to capture pieces of the architectural components (e.g., *DialogPackage*, *ServicePackage*) and their interactions.

Figure 99 depicts the basic data types of the HMS and their relationships. The figure shows that the HMS has to handle data like *Guest*, their *Security* deposits in case of reservations, or their *Payment* and *Expense* descriptions in case of their actual stays at a *Hotel*. It is also stated that a *Guest* may either stay at a *Hotel*

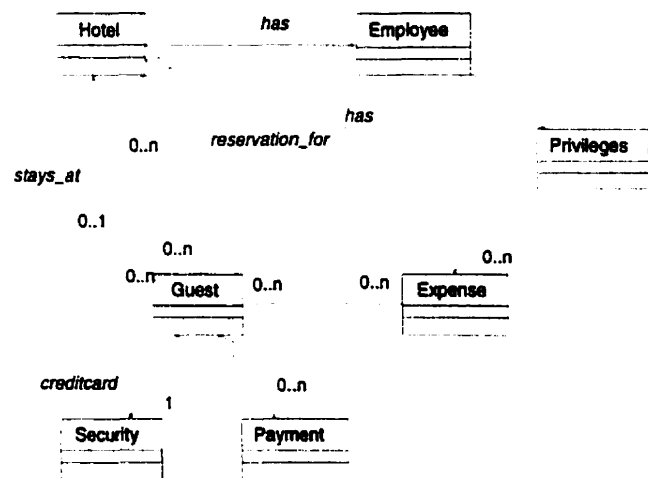


Figure 99. HMS Data Types

or may have a reservation for it. The diagram also captures some basic *Employee* information and their access *Privileges*. The latter information is required to know what guest-related data can be accessed and manipulated by what employee (e.g., *ReservationClerk*, *CounterClerk*, *ManagerClerk*). For instance, a

counter clerk has the privileges to check-in a guest or to add expense or payment charges. However, a counter clerk has no ability (or privilege) to make other types of monetary interactions (e.g., late reminders, service charges, etc.).

Figure 99 also depicts some cardinality information between the basic HMS data types. For instance, it is stated that a *Guest* may have reservations for zero, one, or more *Hotels* at any given time or that a *Guest* may stay at most at one *Hotel* at any given time. It is also stated that a *Guest* must always have a *Security* (e.g., in the form of a credit card), regardless of whether the guest stays at the hotel or has a reservation for it. This is done to ensure that late fees or cancellation fees can be charged at a later time.

The HMS provides three basic service packages corresponding to the needs of the three types of employees who have access to the system (Figure 100). The *ReservationService* is used by employees responsible for making reservations, *CounterService* is used by employees within hotels (mainly for check-in/check-out types of activities), and *AccountingService* is used by the financial group of employees who are maintaining guest accounts and their transactions. According to the service structure, the HMS provides three applications to access those services (*ReservationApp*, *CounterApp*, and *AccountingApp*). Additionally, a *ManagerApp* is provided to allow access to all three types of services. The access rights (privileges) are stored together with the employee information (see Figure 99).

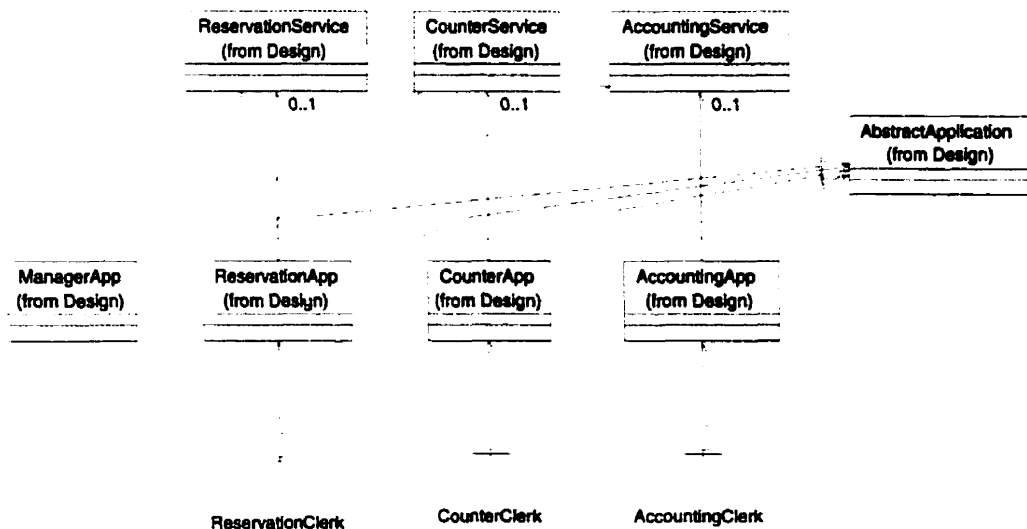


Figure 100. Employees Interacting with Applications using Services

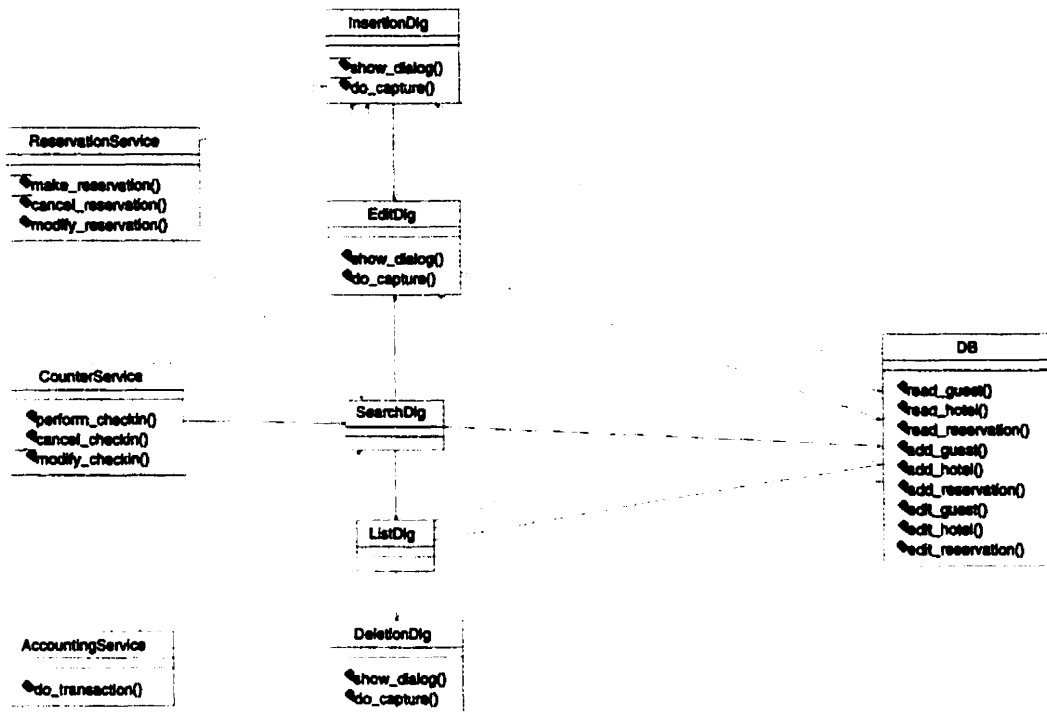


Figure 101. Services, their Dialogs, and the Database (DB)

Figure 101 depicts the kinds of dialogs (e.g., *InsertionDlg*, *EditDlg*, *SearchDlg*) that can be used by the services. Dialog classes are user interface classes and display data on computer screens and capture user inputs. For instance *EditDlg* is used to allow clerks to modify information of common HMS data types (e.g., hotel, guest, etc.). Other dialogs follow the same structure and provide user interfaces for inserting, deleting, searching, or listing of HMS data types. It can be observed in Figure 101 that not all dialogs are used by all services equally. For instance, the *ReservationService* requires access to *SearchDlg* and *InsertionDlg* (directly), and *EditDlg*, *DeletionDlg* or *ListDlg* (indirectly). This means that *ReservationService* may call *SearchDlg* to search for a reservation, which, in turn, may call dialogs like *EditDlg* to display a found reservation record. It can also be observed that the dialogs access the database.

Figure 102 refines what dialogs are using what containers. Containers are pieces of a user interface that handle self-contained elements of a dialog. For instance, a search container provides a user interface for searching data which is similar for all types (e.g., hotel, guest, transaction, etc.). The other containers (*CaptureContainer* and *ListContainer*) provide similar interfaces for modifying and listing.

The reason for separating dialogs from containers is to increase reuse and improve maintenance. For instance, the *EditDlg* for reservations incorporates a reservation capture section as well as a guest and hotel search section. Similarly, *SearchDlg* for hotel and guest respectively provide their own search functionalities.

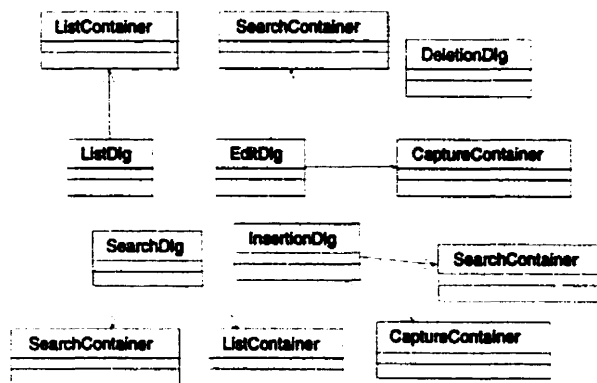


Figure 102. Containers used by Dialogs

Instead of programming guest and hotel searches twice, we only provide one as part of the *SearchContainer* which both can access. The *CaptureContainer* can also be reused by *EditDlg* and *InsertationDlg*, thus, requiring the implementation of data capture capabilities only once instead of twice. Containers also improve maintenance because if the specification of a data type changes, only the containers need to be updated (once) instead of requiring multiple similar updates in different dialogs.

Figure 103 depicts the relationships between services, dialogs, and their data types in more detail. The left side of the figure shows the three service types (*ReservationService*, *CounterService*, and *AccountingService*) and indicates what data types are used by them. For instance, *AccountingService* only needs to have access to *Guest*, *Payment*, and *Expense* data types but does not need access to the *Hotel* data type. Similarly, the different dialogs only require access to some data types. For instance, *SearchDlg* only requires knowledge of *Guest*, *Hotel*, *Payment*, and *Expense* (it cannot be used to search for security

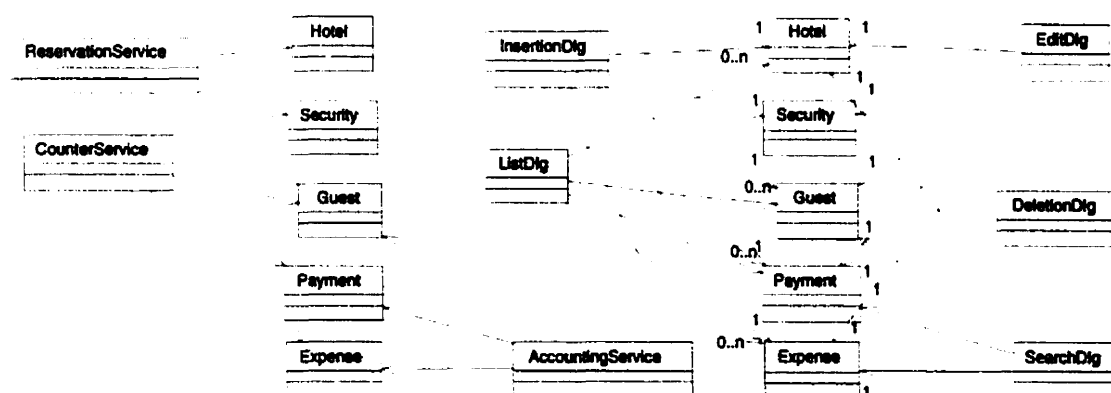


Figure 103. Data Types used by Services and Dialogs

deposits). The *DeletionDlg* does not requires access to any HMS data type since it comprised only of a simple question. The other dialogs, however, do need access to all HMS data types, however, at varying cardinalities. For instance, *EditDlg* displays only one guest at a time whereas *ListDlg* displays a potentially large number of guests.

Having discussed the breakdown of classes, we also need to show how those classes relate to the components in the architecture level. Figure 104 shows the mapping of some the high-level design classes to the architecture. The mappings are indicated through abstraction relationships. For instance, *ManagerApp*, *AccountingApp*, and *CounterApp* are refinements of the architecture component *ApplicationPackage*. Not all design classes are included in this mapping since some classes do not have direct counterparts. For instance, the container classes in the high-level design do not directly map to any architectural component. This is not a problem since our consistency checking approach can handle incomplete mapping information.

Before we discuss the consistency issue between the high-level design class diagrams and the architecture-level component diagram, we will introduce other types of diagrams. In the following, we will show pieces of object, sequence, and statechart diagrams supporting above class structure.

Figure 105 shows an instantiation of some of the data types in Figure 99. Figure 105 depicts a collection of guests and hotels as well as their relationships. For instance, it can be seen that *Peter* stays at the *ShoresInn* hotel for which he also has a reservation. *Ann* currently stays both at the *ShoresInn* and at

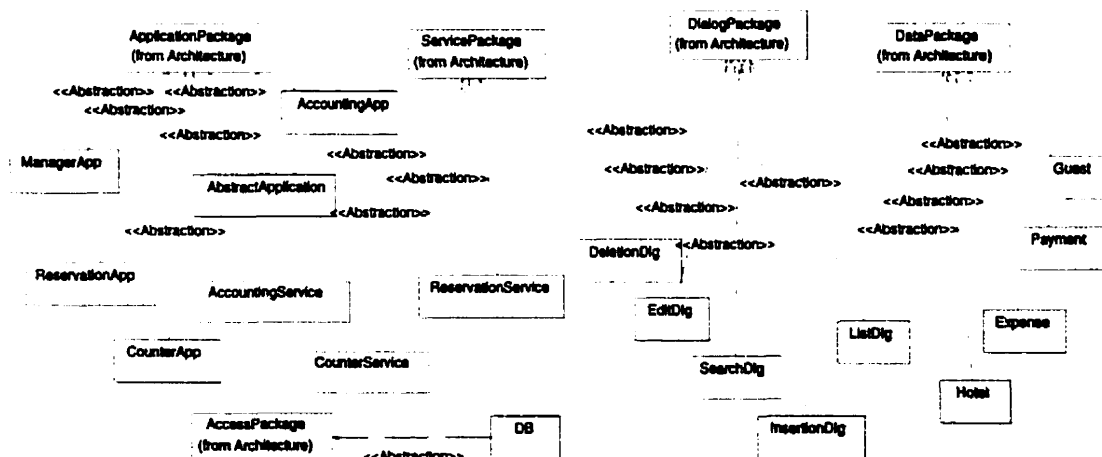


Figure 104. Mapping from Design Classes to Architecture Components

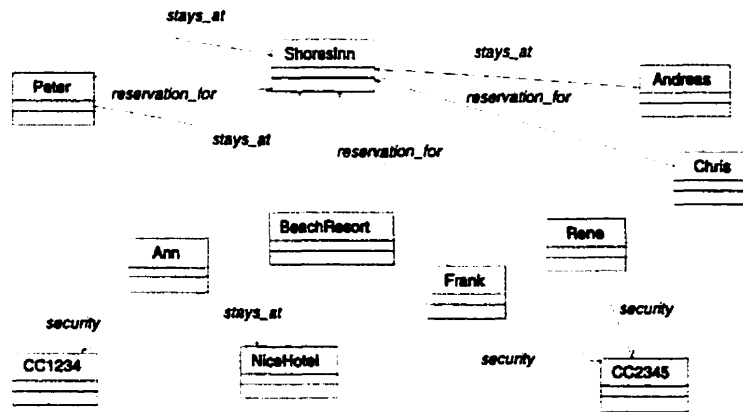


Figure 105. Object Diagram Depicting the Relationships between Guests and Hotels

the *NiceHotel*. She also had to give a credit card (CC1234) as a security. It can also be seen that the *BeachResort* hotel currently has no guest or any reservations. Also, the guest *Rene* has neither a reservation for a hotel nor does he currently stay at any hotel.

Figure 106 depicts a statechart diagram for the class *EditDlg*. Recall Figure 101 where we introduced *EditDlg* and explained that it is used to capture and display HMS data. We also discussed previously that *EditDlg* uses multiple containers to capture HMS data. The statechart diagram in Figure 106 states that *EditDlg* transitions from state *idle* to state *valid* if and only if all inputs (all containers) have a *valid* input. It is, therefore, left to the state information of other classes (e.g., *CaptureContainer*) to determine whether *ReservationDlg* transitions from *idle* to *valid*.

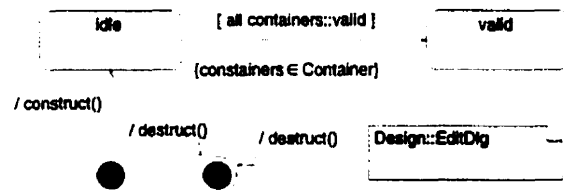


Figure 106. Statechart for *EditDlg*

Figure 107 depicts the state diagram of one such container. Indeed, the capture container offers the state *valid* which indicates that the information entered currently constitutes a valid input. A valid input is determined by investigating the *isValid()* method of an object called *element*. *Element* is an object pointing to a basic data type

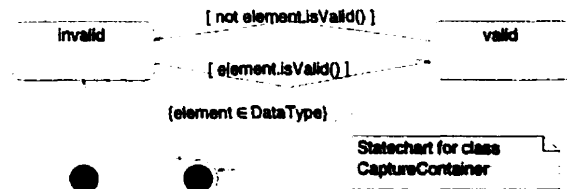


Figure 107. Statechart for *CaptureContainer*

(e.g., *Guest* or *Hotel*). For instance, if the container handles guests records (e.g., *GuestCaptureContainer*), then element is of type *Guest*. It is left to the object *Guest* to know whether it is valid or not. Being valid means that the information captured currently is sufficient to describe a guest. Normally, name, address, and security (e.g., credit card) information are required for a valid guest entry.

Figure 108 shows the statechart diagram for the class *ReservationService*. Recall from Figure 101 that *ReservationService* implements some of the business logic for HMS applications (*ReservationService* in particular implements the business logic involving reservations). After instantiation, a *ReservationService* object is in the *idle* state. It is left unspecified how to transition from *idle* to the state *capture*, however, it can be observed that

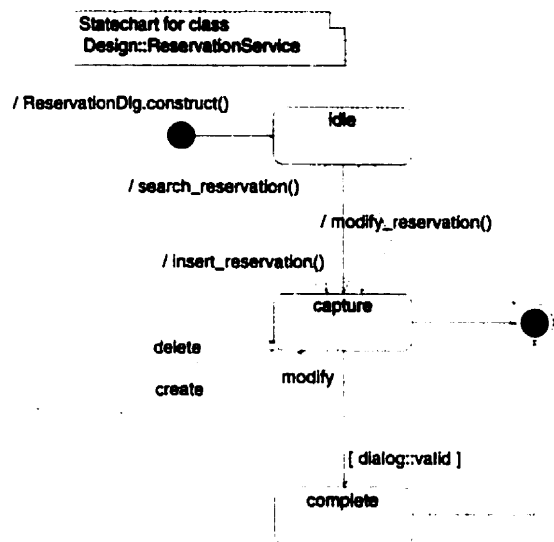


Figure 108. Statechart for *ReservationService*

such a transition is possible. If all required information about reservations has been captured (done via dialogs), the state of *ReservationService* transitions to *complete*. From this state it is possible to actually *create*, *modify*, or *delete* a reservation entry in the database.

Figure 109 depicts a sequence diagram showing the timing of certain activities during the process of modifying a reservation. First the *modify_reservation()* method is called in *ReservationService*, which, in turn, reads the reservation (via method *read_reservation()*) and then opens a dialog window (via method *show_dialog()*). The dialog window waits until all reservation information has been captured (note that interactions with containers become relevant here but were omitted). Thereafter, the dialog queries the database to find the new hotel information, which is then instantiated as a new *Hotel* object. At the end, the reservation is updated by calling the DB method *edit_reservation()*.

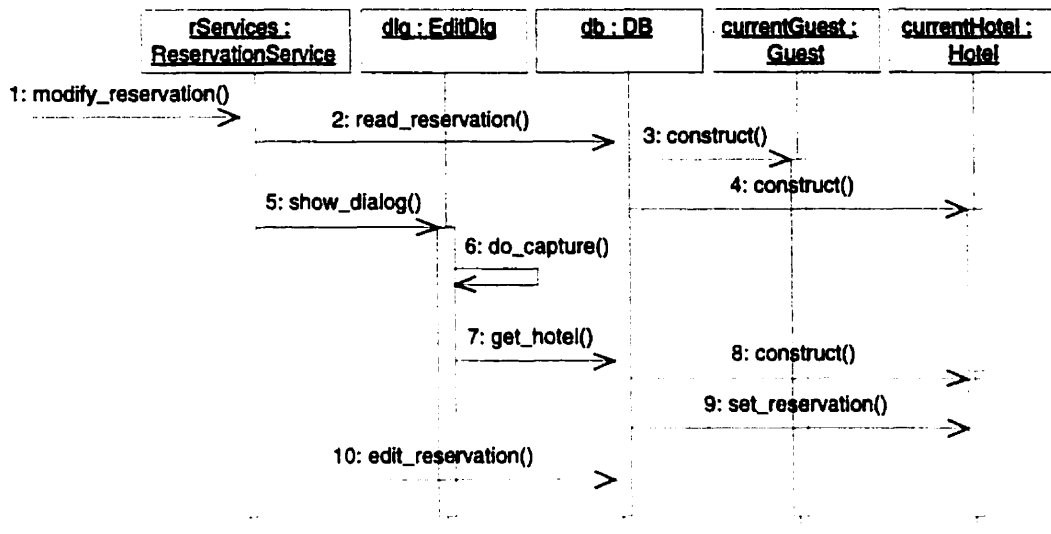


Figure 109. Sequence Diagram depicting a Search for a Reservation

8.2.2 Transformations

The refinement of the architecture level into the high-level design increased the information content of the model by an order of magnitude. Since the architecture only provided one view, consistency checking within the architecture level is not necessary. However, with the added design information, we are now confronted with about a dozen diagrams part of different subsystems and levels of abstractions. Figure 110 shows the basic structure how those diagrams inter-relate. The top level

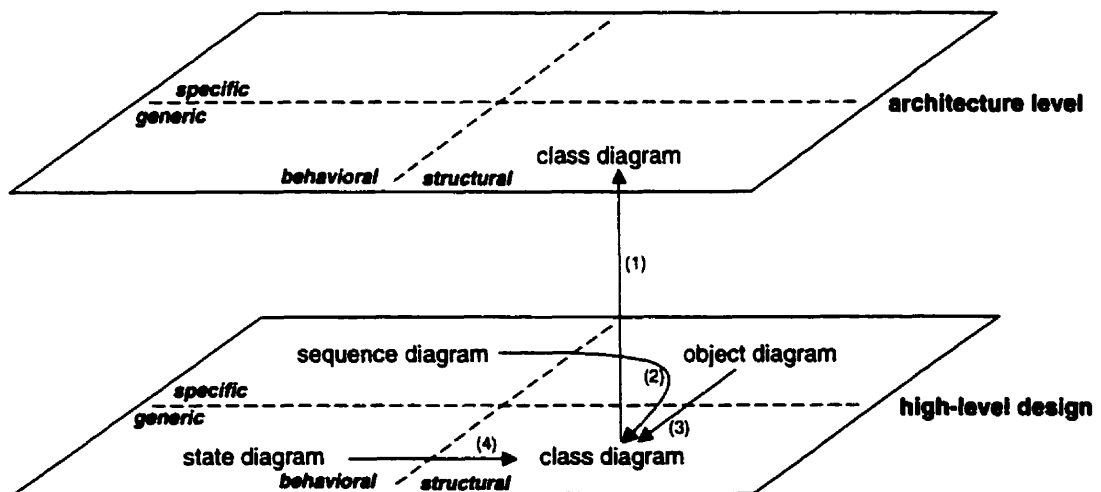


Figure 110. Transformations to support Consistency Checking of Architecture and Design

depicts the architecture level which currently holds one diagram (Figure 98). The architecture was refined into a series of class diagrams depicted at the lower right area of the high-level design in Figure 110.

Additionally, the design level provided an object diagram (Figure 105), a sequence diagram (Figure 109), and several statechart diagrams. From Chapter 5.5 we know that the object diagram is structural and specific, the sequence diagram is behavioral and specific, the statechart diagrams are behavioral and generic, and the class diagrams are structural and generic. Those diagrams are accordingly represented in the specific and/or behavioral sections of the high-level design. Given the current state of the model, the following four transformations are necessary to enable consistency checking between the currently existing diagrams:

1. Abstraction between the high-level-design class diagrams and the architecture-level class diagram,
2. Structuralization between the high-level-design sequence and object diagrams,
3. Generalization between the high-level-design object and class diagrams, and
4. Structuralization between the high-level-design statechart and class diagrams.

8.2.3 Consistency Checking

Figure 111 depicts the result of abstracting the high-level design classes into a form that is suitable for comparison with the architecture-level diagram from Figure 98. Figure 111 was generated with our UML/Analyzer tool. It can be observed that the basic structure of the abstraction is similar to Figure 98. However, on closer inspection, a series of inconsistencies can be observed:

1. The actor *ManagerClerk* is not connected to *Application*
2. The roles of *ServicePackage* and *DialogPackage* are reversed which causes the direction of the relationship between them to be reversed

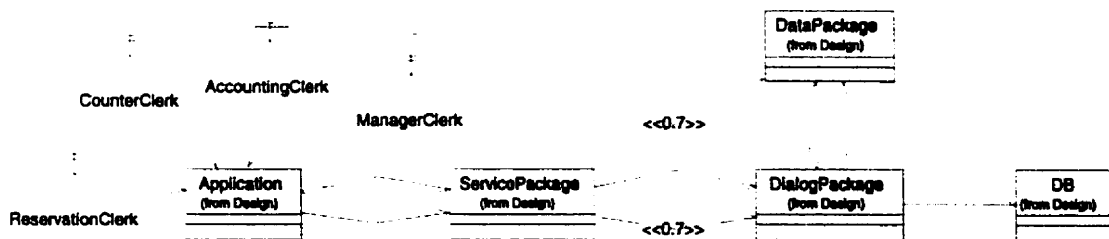


Figure 111. Abstracted Design-Level Class Diagram

3. Concrete classes like *SearchContainer* and *Employee* have not been assigned to abstract classes
4. Concrete relationships like the association between *ReservationApp* and *ReservationService* have not been realized in the abstraction
5. The database (*DB*) class is not connected to *DataPackage*

Figure 112 depicts the result of structuralization and subsequent generalization of the sequence diagram shown in Figure 109. The result is an interpreted class diagram that should be consistent with the high-level class diagrams. Since the sequence diagram is a specific view, the consistency checking has to follow the *part-of* mode in that it is not expected that it completely represent the

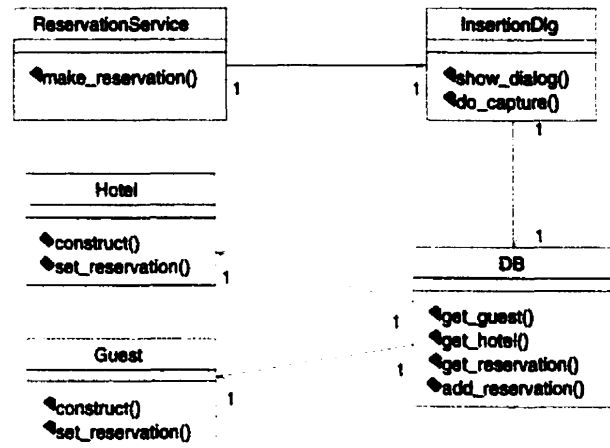


Figure 112. Structuralized and Generalized Sequence Diagram

generic class diagrams. Thus, validating the consistency involves the validation that the sequence diagram does not contradict the class diagrams. The following inconsistencies can be observed:

1. Methods like *DB::get_hotel()* or *Guest::construct()* are not defined in generic view (see Figure 101)
2. The relationship between *ReservationService* and *InsertionDlg*, *DB* and *Guest*, and *DB* and *Hotel* are not defined in generic view

The third required transformation is a generalization from an object diagram (Figure 105). Again, a class diagram (Figure 113) is the result of that transformation which needs to be consistent with the high-level class diagrams. Note that consistency checking between the sequence diagram and object

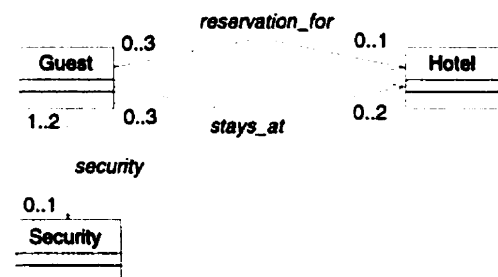


Figure 113. Generalized Object Diagram

diagram is not necessary since their validation is done implicitly by using the design-level class diagrams as a common denominator. Since the object diagram is also a specific view, consistency checking follows the part-of mode. The following inconsistencies can be observed:

1. The relationship between *Guest* and *Security* is different (note: Figure 99 defines that relationship as aggregation which is impossible in Figure 113 because of the zero-to-one cardinality)
2. The cardinality between *Guest* and *Security* as well as between *Guest* and *Hotel* (*stays_at* relationship) are different.

Figure 114 is the forth and final transformation required for consistency checking between the architecture and high level design.

Figure 114 depicts the result of the structuralization of the three statechart diagrams in Figure 106, Figure 107, and Figure 108. The knowledge of

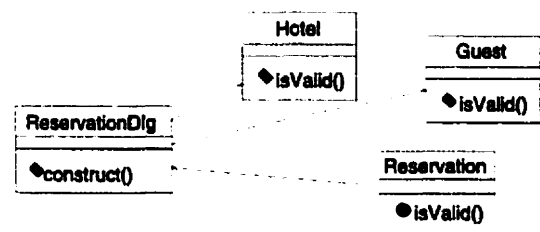


Figure 114. Structuralized Statechart Diagrams

statechart ownership and references are used here to infer class-relevant information. It can be observed

that methods like *isValid()* and *construct()* have not been declared in the class views. Otherwise, no inconsistencies exist.

8.3 Refinement to Lower-Level Design

As it can be seen in above diagrams, having transformation methods simplifies consistency checking enormously. However, thus far, we only showed consistency checking within class diagrams and between class diagrams and other views. This was primarily caused because the architecture-level only provided a single class diagram. This section will refine the higher-level design diagrams and show other types of transformation and consistency checking.

8.3.1 Overview

Since the lower-level design view contains as large amount of model elements, we will focus on some pieces only in this section. Figure 115 shows the refined version of the data types we previously discussed in Figure 99. As can be seen, the basic data types are still present (e.g., *Guest*, *Hotel*, etc.),

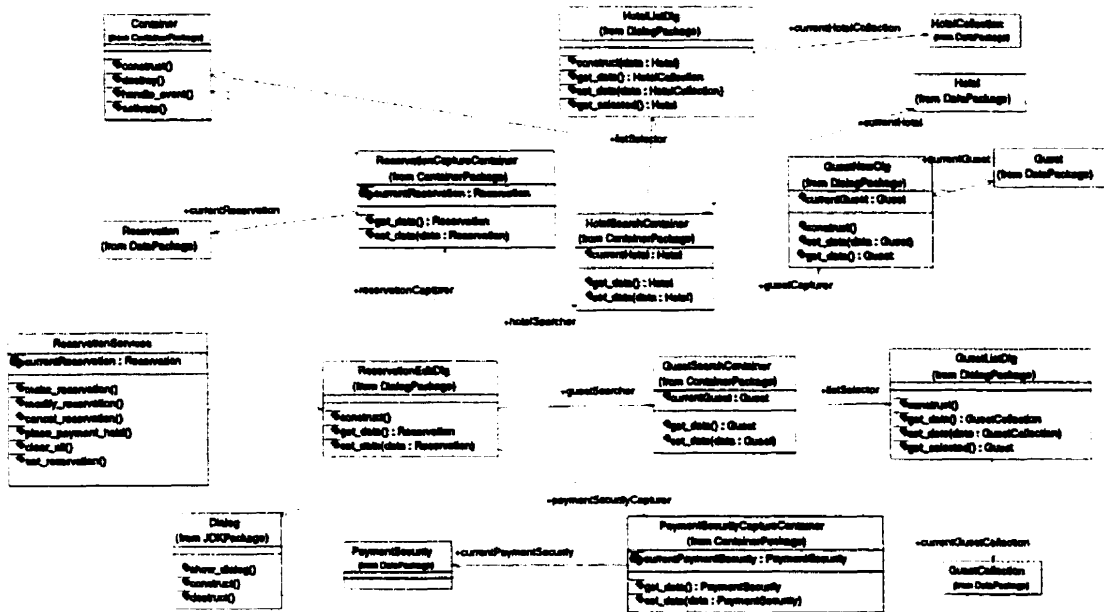


Figure 116. ReservationServices, ReservationDlg, and its Containers

number) or they can be used to make a rough search followed by listing the findings in a separate dialog (*GuestListDlg*).

As a refinement of the *CaptureContainer* in Figure 107, Figure 117 depicts the *ReservationCaptureContainer*. *ReservationCaptureContainer* displays reservation data on the screen and validates provided input. It is specified that its state can only transition from *ReservationCapture* to *ValidReservationCapture* if and only if *currentReservation.validate()* returns true. From Figure 116, we know that *currentReservation* linked from *ReservationCaptureContainer* points to the HMS data type *Reservation* (note the label *+currentReservation* originating from *ReservationCaptureContainer*), indicating that the *validate()* method of the class *Reservation* is meant.

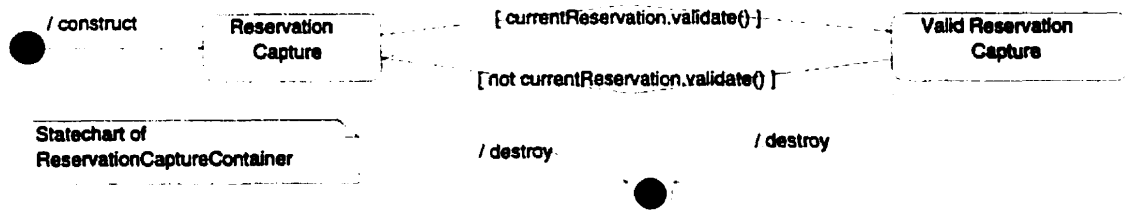


Figure 117. Statechart diagram for ReservationCaptureContainer

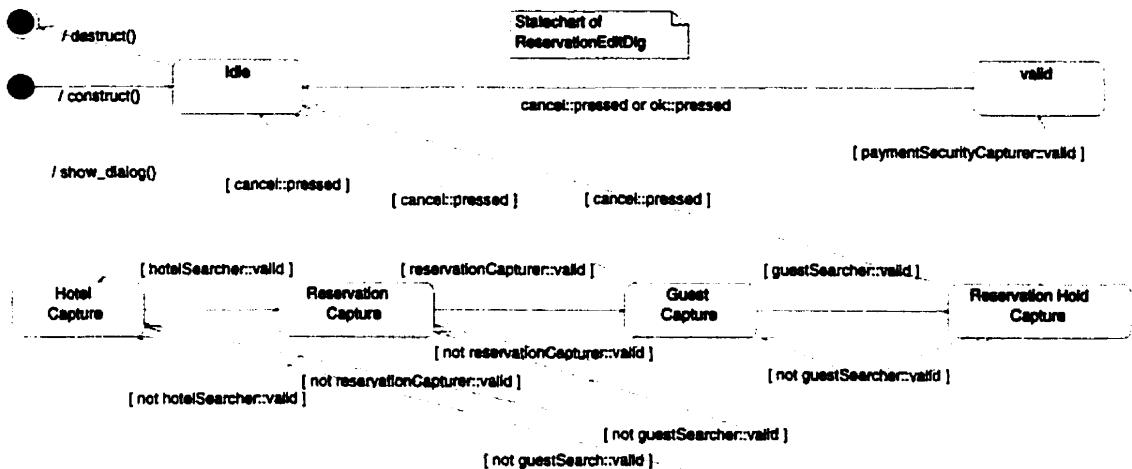


Figure 118. Startchart Diagram for ReservationEditDlg

Figure 118 shows the statechart diagram for *ReservationEditDlg*, which is a refinement of *EditDlg* from Figure 106. *ReservationEditDlg* is fairly complex since it must capture the interactions between the dialog and all its containers. After construction, *ReservationEditDlg* is in the *idle* state. Once *show_dialog()* is invoked, the state of *ReservationEditDlg* transitions to *HotelCapture*. Once hotel information has been completely (validly) captured, *ReservationEditDlg* transitions to *ReservationCapture*. After the successful capture of reservation, guest, and security information, *ReservationEditDlg* reaches the *valid* state. Now the user (clerk) has the option of pressing the OK button on the screen to make modifications to the reservation. The cancel button could have been used throughout the process.

Figure 119 depicts a sequence diagram showing how classes interact during modifying a reservation. The links in the sequence diagram reflect the ordering of method calls. It can be seen that the modification of a reservation involves the *ReservationService* making a call to *ReservationEditDlg* which in turn instantiates needed data types and containers. Once the entered data is valid, the database is called to make the actual modification (*edit_reservation()*).

At the end, the sequence diagram in Figure 119 proceeds in displaying the same record again by calling the *show_dialog()* method. The previous construct methods are not needed this time since the objects still exist.



8.3.2 Transformations

The refinement of the high-level design into the low-level design again increased the information content of the model. Since the high-level design and architecture were already checked for consistency, this step only needs to validate the consistency between the high-level design and low-level design. Note that we could also validate the consistency between the architecture and the low-level design but we would benefit only little through it. To understand this, assume for a moment that all three levels of abstraction are completely consistent. In that case the comparison between the architecture and high-level design would reveal no inconsistencies and neither would the comparison between the high-level design and low-level design. If we would now also compare the low-level design with the architecture, we would naturally also not find any inconsistency since we indirectly proved this already. It follows that we only need to validate the consistency between the two design levels.

Like before, validating consistency involves the transformation of diagrams in such a manner that they become directly comparable to the remaining diagrams. Like Figure 110 previously, Figure 120 again shows the relationships between the lower-level design diagrams and the higher-level ones. Given the current state of the model, the following five transformations are necessary to enable consistency checking between the currently existing diagrams:

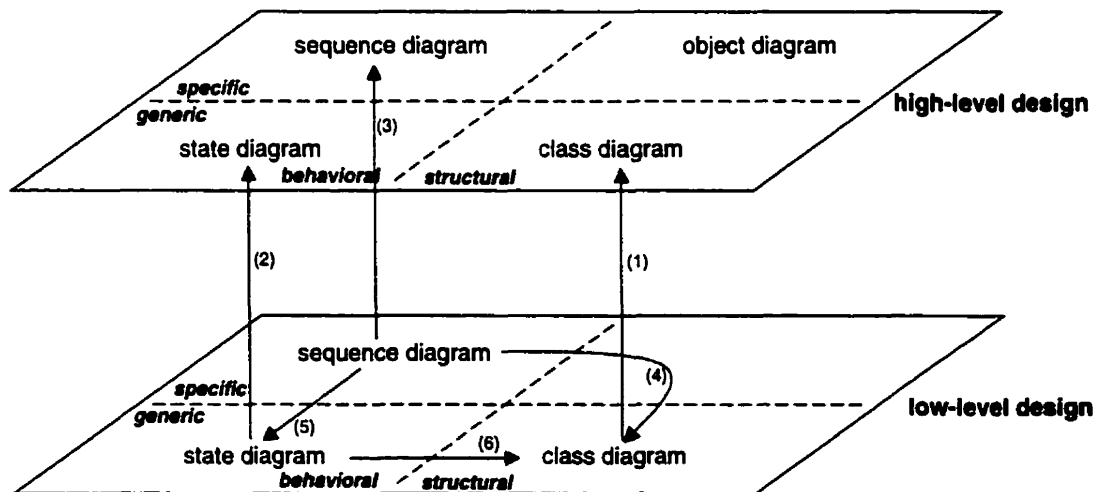


Figure 120. Transformations to support Consistency Checking between Designs

1. Abstraction between the low-level class diagrams and the high-level class diagrams,
2. Abstraction between the low-level statechart diagrams and the high-level statechart diagrams,
3. Abstraction between the low-level sequence diagrams and the high-level sequence diagrams,
4. Structuralization followed by generalization between the low-level sequence and class diagrams,
5. Generalization between the low-level sequence and statechart diagrams, and
6. Structuralization between the low-level statechart and class diagrams.

To enable consistency checking, Figure 121 depicts the mapping between some of the lower-level classes and higher-level classes. For instance, it can be observed that *CreditCard*, *Cash*, and *Check* are refinements of *Payment* or that *GuestSearchContainer* and *HotelSearchContainer* realize *SearchContainer*. This type of trace information is again needed to ensure automated transformation and consistency checking.

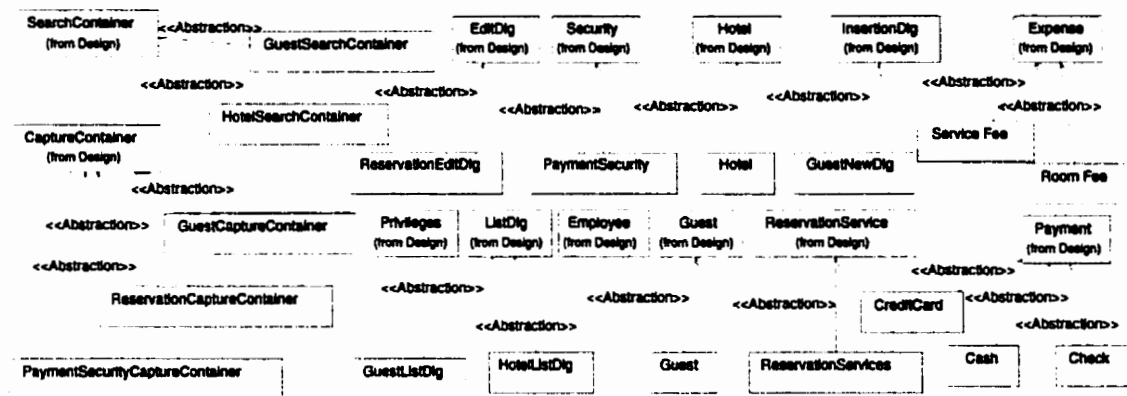


Figure 121, Mapping from Low-Level Design Classes to High-Level Design Classes

8.3.3 Consistency Checking

Figure 122 depicts the result of abstracting the low-level design classes into a form that is suitable for comparison with the high-level class diagram from Figure 115 and Figure 116. Figure 122 was again generated with our UML/Analyzer tool. Since we did not depict the complete lower-level class diagram, the abstraction only reflects a part of the high-level diagram. The following inconsistencies can be observed:

1. There is only one association between *Hotel* and *Guest*, and this association has the wrong direction
2. Classes *Employee* and *Privileges* were not refined
3. Cardinality between *Guest* and *Expense* is wrong (note that association type and cardinality between *Guest* and *Payment* is correct since the diamond head in Figure 99 corresponds to a cardinality “1”)
4. Relationship between *Hotel* and *PaymentSecurity* is illegal
5. Relationships from *SearchContainer* and *CaptureContainer* to *InsertionDlg* and *ListDlg* are not allowed by abstraction
6. Likewise, relationships from *Containers* to *Guest* and *Hotel* are illegal

Figure 122. Abstracted Low-Level Design Class Diagram

Since the class *ReservationCaptureContainer* is a refinement of the high-level class *CaptureContainer*, it follows that the statechart diagrams attached to them relate in the same manner. The statechart diagram in Figure 117 thus refines the statechart diagram in Figure 107. Since the basic structure between both statechart diagrams is identical (box and arrows relate in a one-to-one fashion), the abstraction process leaves Figure 117 unmodified. Comparing Figure 117 and Figure 107, we find that all interconnectivities are consistent. Figure 107 states that *element.isValid()* must be true for a transition to happen and Figure 107 also defines that *element* must be a *DataType* (e.g., *Guest*, *Hotel*, etc.). We can see from Figure 116 that *currentReservation* (as defined in Figure 117) is a link to the *Reservation* data type as it was defined in Figure 115. Currently, however, we did not specify the traceability (mapping) that *Reservation* implements one of the data types of the high-level design—we

found an inconsistency. The absence of the *construct()* and *destruct()* methods (from Figure 117) is not an inconsistency since those methods were not yet defined in *CaptureContainer* in the high-level design.

The class *ReservationEditDlg* is also a refinement of the class *EditDlg*. The statechart diagram corresponding to *ReservationEditDlg* (Figure 118) is, however, more complex than the one corresponding to *EditDlg* (Figure 106). If we assume that the *idle* and *valid* states correspond to one another, then we get an abstraction of *ReservationEditDlg* that looks like Figure 123. It must be noted, that all information that is not known on a higher-level is omitted and transitive relationships were created. For instance, there is only one path from *idle* to *valid*. This path uses intermediate states like *HotelCapture*. We also know from the lower-level class diagrams that attributes like *hotelSearcher* link to containers (see Figure 116). With that we can infer that the methods *show_dialog()* and the *valid* states of several *SearchContainer* and *CaptureContainer* are needed for Figure 123 to transition from *idle* to *valid*. Circular links within *idle* are also possible but not easily abstractable since *cancel::pressed* is referring to a class that is not defined (nor known) at the higher-level. A blank circular link is therefore depicted. Likewise, the backward transition from *valid* to *idle* is blank, indicating that it is possible to do it but it cannot be inferred how. We can observe the following inconsistencies:

1. Circular transition not allowed in abstraction
2. Backward transition not allowed in abstraction
3. Method call *guestSearch* as used in Figure 118 is not defined
4. Method call *show_dialog()* is used in lower level to transition between *idle* and *valid* but is not used at higher level.

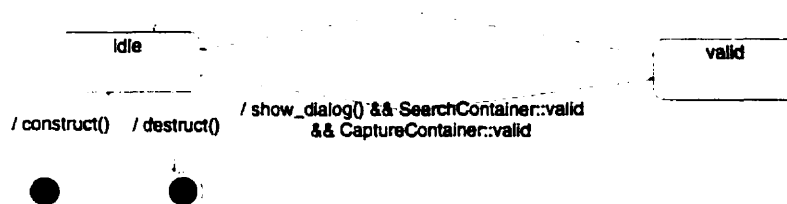


Figure 123. Abstracted Statechart Diagram for EditDlg

Under normal circumstances, consistency checking between specific views is not very meaningful since specific views represent usage scenarios and scenarios may vary. For instance, if one specific view defines $A=9$ and the other defined $A=5$, then this denotes no inconsistency. However, based on trace information (mapping), we may choose to force comparisons between specific views. In our model, we make the claim that the sequence diagram in Figure 119 is a refinement of the sequence diagram in Figure 109 since both depict the same scenario on how to modify a reservation by changing the hotel. To allow consistency checking between the two sequence diagrams, Figure 119 must be abstracted. Abstracting a sequence diagram is similar to abstracting a class diagram. It involves the grouping of classes and the derivation of transitive relationships. Like with statechart diagrams before, method calls that were not defined in the abstraction may be omitted. Figure 124 depicts the abstraction of Figure 119 and it can be observed that the abstraction looks very similar to the high-level sequence diagram. Note that we did not explicitly state a trace from *handle_event* (lower-level) to *do_capture* (higher-level). The following inconsistencies can be observed:

1. The method call *get_hotel()* was not refined as well as its subsequent method call *set_reservation()*
2. The method call *do_capture()* was called three time although only one was expected

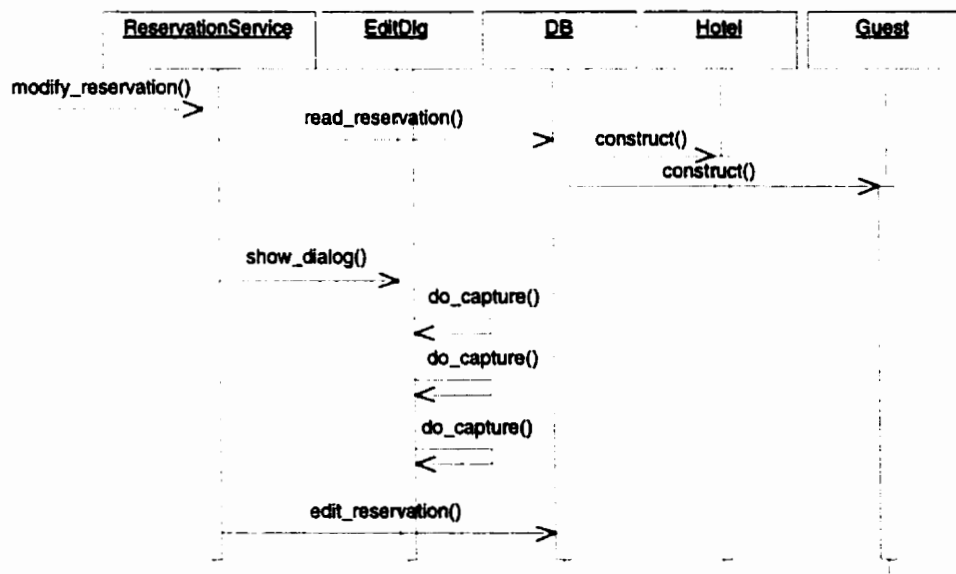


Figure 124. Abstracted Sequence Diagram for *modify_reservation()*

The forth type of transformation we need to perform is a structuralization followed by a generalization of the low-level sequence diagram. The result of that operation is a low-level class diagram that should be consistent with our current class diagrams. We find the following inconsistencies:

1. The calling dependencies between *ReservationEditDlg*, *GuestCaptureContainer* and *HotelCaptureContainer* are not allowed
2. The class *Button* does not exist

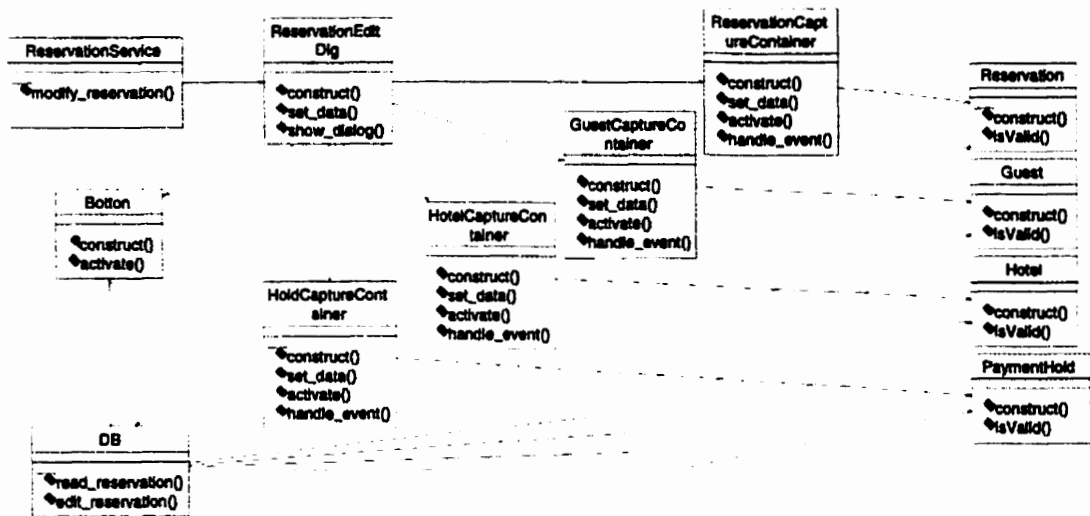


Figure 125. Structuralization and Generalization of Sequence Diagram

Note that the methods *handle_event()* or *activate()* and the relationships to *DB* are undefined in Figure 115 and Figure 116, however, they do exist in the lower-level class definition. For brevity, their definitions were omitted previously.

For the fifth transformation, the generalization of the sequence diagram in Figure 119 to a statechart diagram like Figure 118, we again need additional traceability (mapping) information to proceed. A sequence diagram (like the one in Figure 119) can be used to create pieces of multiple statechart diagrams belonging to multiple component. Here we will demonstrate it on the case of *ReservationEditDlg*. It can be observed that *ReservationEditDlg* receives and sends out information during the course of modifying a reservation. We define *construct()* as a *constructor* and the methods *show_dialog()* and *activate()* as actions. Using that information, the sequence diagram can be transformed into a statechart diagram which looks like Figure 126. It can be seen that the *construct()* method

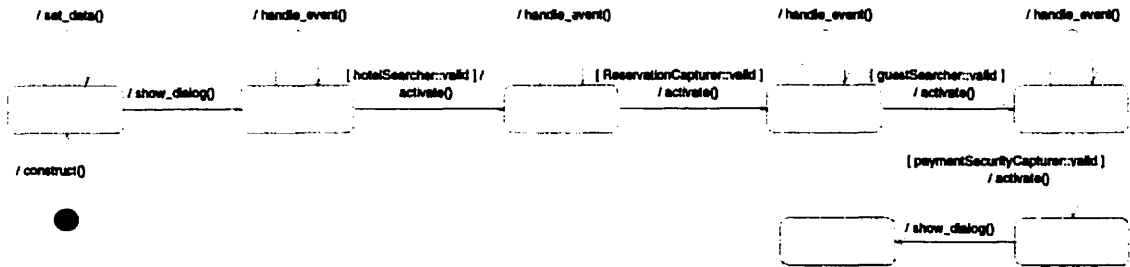


Figure 126. Generalized Sequence Diagram to Statechart Diagram

transitions form a start state to a regular state, the methods *set_data()* and *handle_event()* are circular state transitions indicating that they are queries (or undefined), and the methods *show_dialog()* and *activate()* are regular state transitions. No information can be inferred about the state names, however, some of the conditions that enable state transitions can be taken over. It can be observed that the two statechart diagrams are almost consistent, except for the last *show_dialog()* method. The original statechart diagram in Figure 118 requires the occurrence of another state transition [*cancel::pressed*] or [*ok::pressed*] before *show_dialog()* may be called again.

The sixth, and final, transformation is a structuralization between the statechart and class diagrams. Figure 127 shows the result of transforming the two statechart diagrams in Figure 117 and Figure 118 into a class diagram that becomes comparable to Figure 115 and Figure 116. The following inconsistencies can be observed:

1. Class *Button* is not defined
2. Class *Reservation* does not have a *validate()* method
3. Classes *ReservationCaptureContainer* and *ReservationEditDlg* do not have *destroy()* methods

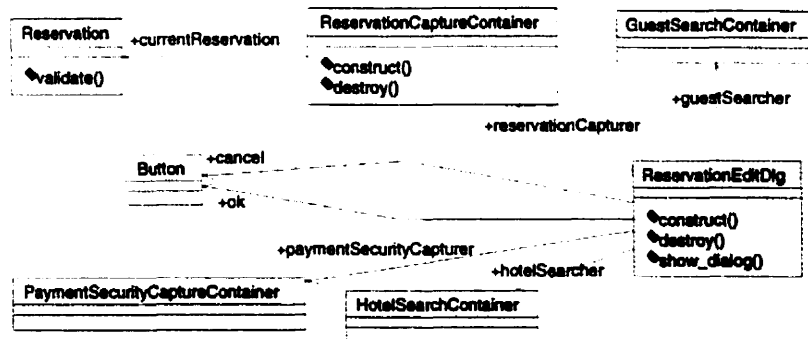


Figure 127. Structuralized Statechart Diagrams into Class Diagrams

8.4 Scalability

Without any scalability measures and without our view integration framework, consistency checking of 19 user-defined diagrams would require up to 171 transformations and/or consistency checks (not counting model elements). Additionally, such an approach would also require up to 22 transformation types. In context of the HMS, such an unscalable approach would actually have to perform 118 transformations and 17 transformation methods.

Our consistency checking approach is geared towards scalable consistency checking supporting extensive reuse. In context of the HMS, our approach only requires 17 transformations and 7 transformation methods. It must however, be noted that the worst case scenario of our approach could require up to 342 transformations and 8 transformation methods for 19 diagrams. This scenario would happen if 18 concrete sequence diagrams were to be transformed to 1 abstract class diagram. Our experience at looking at dozens of UML models shows that such a worst case scenario is unlikely. Similarly unlikely are other cases that yield bad performances. Our approaches' performance improves the more types of diagrams are used. Above worst case showed the use of only two types of diagrams that are located in the most extreme "corners" of our transformation framework.

Even if such a worst case scenario would occur, the number of transformations would only be a factor of two higher, not significantly worsening the unscalable approach, however, our approach would, in a worst case scenario, still only require 8 transformation methods (versus 22). We therefore see our approach as both an improvement in the number of transformations to be performed under "normal" usages as well as an improvement in the number of transformation methods to be implemented to support consistency checking. Our approach therefore also lowers the entry barrier to enable large scale and scalable consistency checking.

Finally, our approach also scales well if additional types of diagrams are introduced. For instance, if collaboration diagrams need to be supported by our approach only a single translation method between sequence and collaboration diagrams needs to be added. We already added C2 ADL (C2SADEL) and also only required a single translation method. Thus, these two additional diagram types

only add two transformation methods. An unscalable approach would require the additional of 44 transformation methods on top of the 22 already existing ones. In such a scenario, our approach improves the number of methods required by an order of magnitude.

8.5 Summary

This section presented a non-trivial case study of a hotel management system and demonstrated our approach in context of 19 diagrams. We showed the transformations required to perform consistency checking and we listed the inconsistencies the 19 diagrams currently exhibit. We concluded the discussion by comparing some scalability numbers between our approach versus a non-scalable one.

9 UML/Analyzer–A Tool

The UML/Analyzer tool implements our view transformation framework in the context of object, class, and C2SADEL diagrams. Figure 128 depicts some screen snapshots of the tool. UML/Analyzer is integrated with Rational Rose™ for the purpose of using it to create and modify views (synthesis). Rational Rose models are converted through an automated process into a system model called UML-A where they are analyzed via UML/Analyzer (UML-A is an adaptation of UML to support advanced consistency checking concepts like reduced redundancy models). Generated modeling information as well as identified model inconsistencies can be fed back into Rational Rose for visualization. Figure 128 shows Rational Rose™ in the lower right as well as the UML/Analyzer main window to the upper-left. The tool uses transformation rules (upper-right) to convert class and object models. Models loaded into UML/Analyzer can then be transformed and analyzed with respect to their consistency (lower-left).

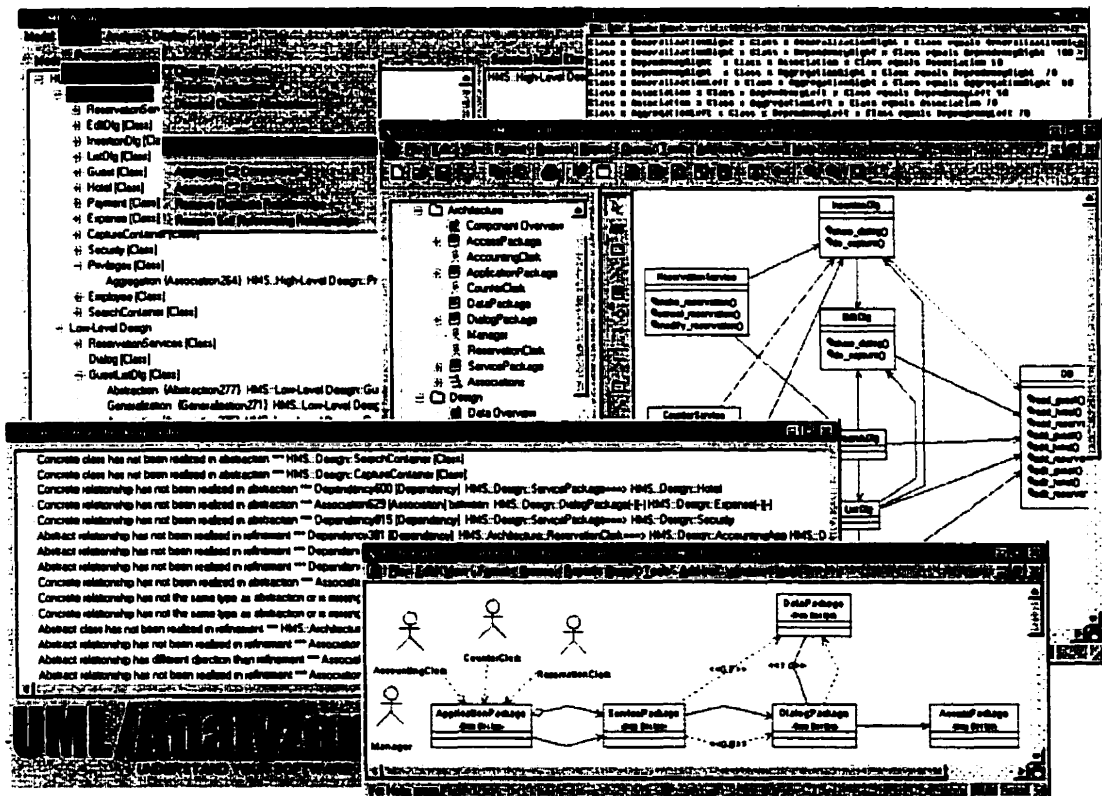


Figure 128. UML-Analyzer Tool Supporting View Integration

At the current state, the tool supports class and object diagram abstraction and consistency checking. Other transformation techniques are still being implemented. The tool also supports the scalability measures discussed in this work to enable their evaluation. Some industrial companies have participated in its creation and/or evaluation. For instance, we have collaborated with Rational Software on our relation abstraction technique [Egyed and Kruchten 1999]. Rational Software also implemented that technique in a tool called Rose/Architect™.

Future plans are to integrate a model constraint parser and checker component (depends on availability of OCL parser and checker) as well as to integrate additional transformation techniques. However, even at its current state by only supporting partial automated transformation we have already observed an enormous benefit in using it. Figure 129 shows a populated UML model of our hotel management system containing both user-defined and derived modeling elements. Figure 129 is analogous to our reduced redundancy model we discussed in Chapter 7.7.3. As it can be seen, the task of

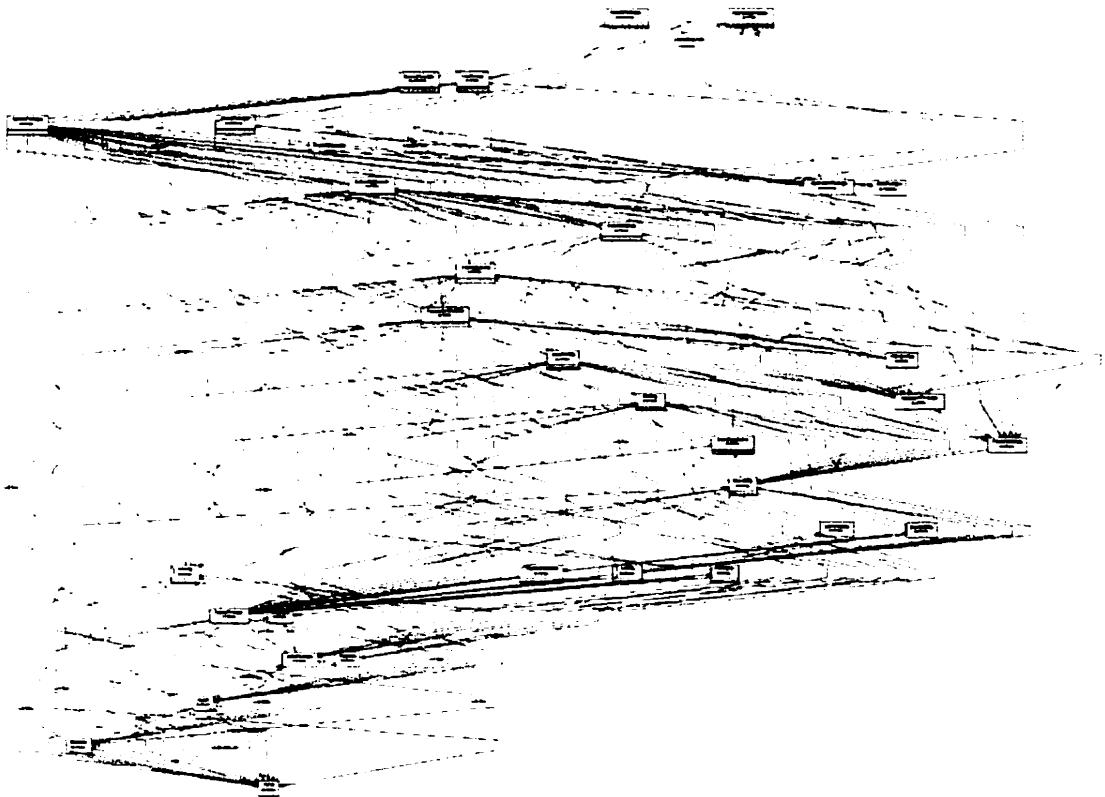


Figure 129. Complexity in Class Abstraction

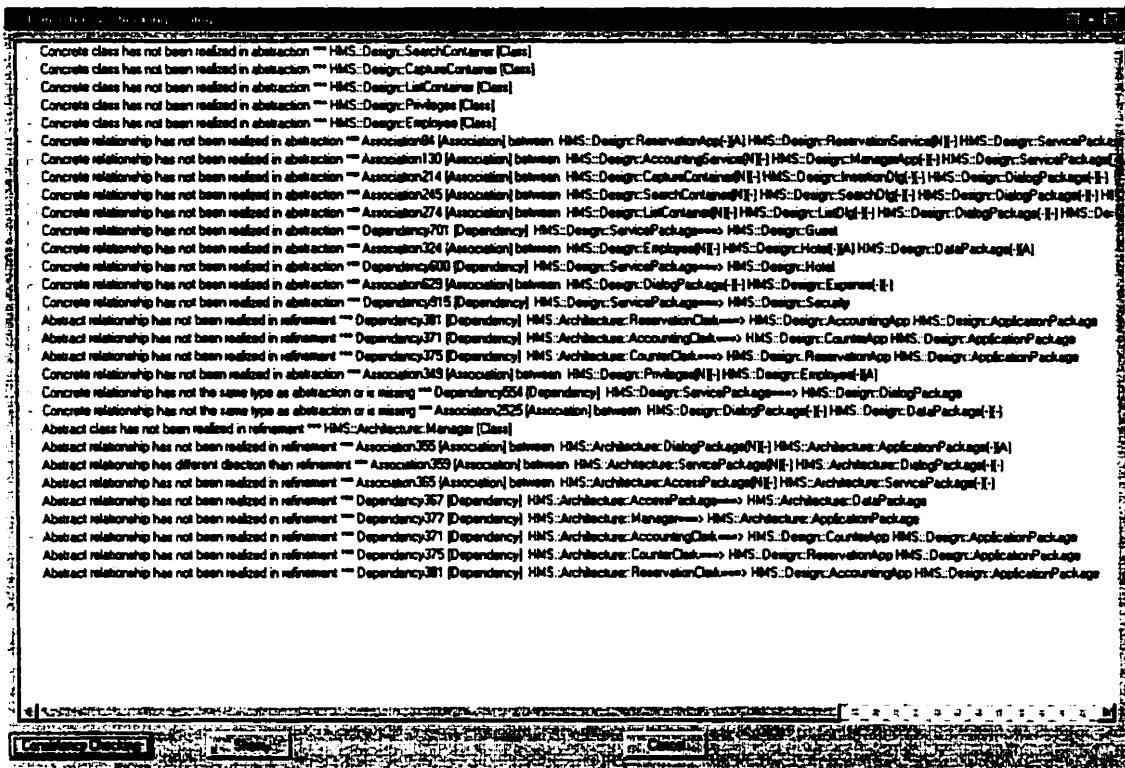


Figure 130. Inconsistencies between HMS Architecture and High-Level Design

abstracting a class model can be considerably complex and time consuming if done manually. Proper class abstraction requires the exploration of all possible path combinations followed by the application of proper class abstraction rules. The tool reduces this task to mere fractions of a second.

Figure 130 depicts the list of inconsistencies between the architecture and design level of the HMS system (see Chapter 8) as generated by UML/Analyzer. Each entry first describes the nature of the inconsistency followed by the list of involved model elements. For instance, the first entry in Figure 131 states that the concrete class *HMS::High-Level Design::SearchContainer* has not been realized in abstraction (recall the lack of a container package in the architecture level). In case of inconsistencies among relationships, usually multiple involved model elements are listed. For instance, the inconsistency “abstract relationship has different direction than refinement” indicates a problem between the two packages *HMS::Architecture::ServicePackage* and *HMS::Architecture::DialogPackage*, saying, that this



Figure 131. Inconsistencies between HMS High- and Low-Level Designs

relationship exists in the high-level design but has a different direction. Figure 131 also lists the inconsistencies among the high-level and low-level designs.

Figure 132 depicts some statistics gathered by the UML/Analyzer tool during the process of abstracting the high-level design into the architecture. During download, roughly 80 relationships are created. Those relationships are part of the design and architecture diagrams of the HMS. During the course of abstracting the relationships among *Clerks*, *ServicePackage*, *DialogPackage*, *ApplicationPackage*, *AccessPackage*, and *DataPackage*, derived relationships are added to the model. Those derived relationships represent the more abstract interdependencies which are created by grouping concrete relationships. Our abstraction process also supports some scalability measures like reuse and elimination of duplicate but similar abstracted relationships. It can be observed that those two scalability measures eliminate the bulk of the derived elements. We have observed a similar pattern while abstracting over a dozen different and non-trivial class diagrams. We have also observed that the amount

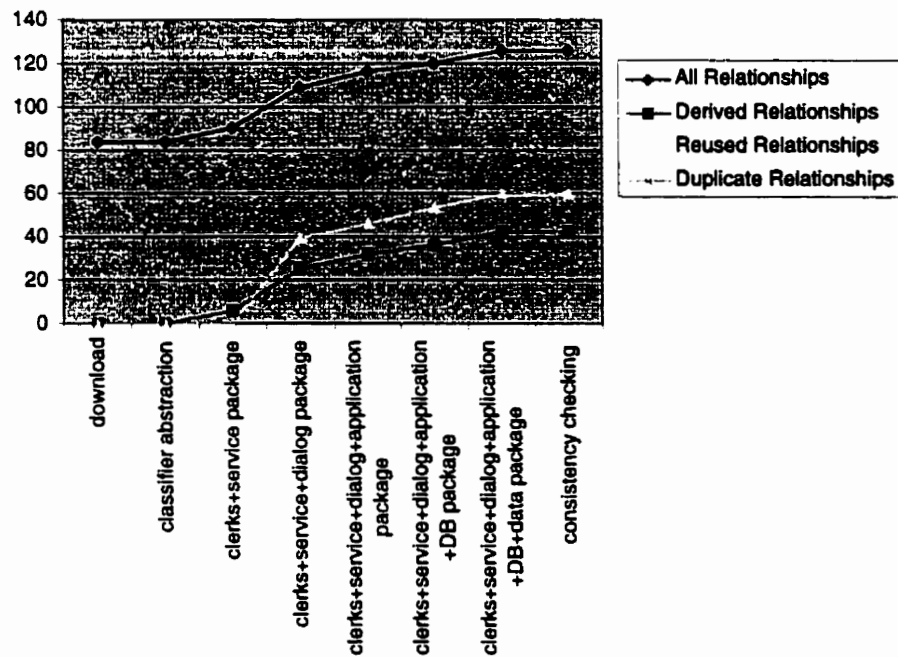


Figure 132. Reuse and Duplication Elimination during Abstraction

of reuse increases over time as the repository is extended. The amount of reuse may peak up to 100% within reuse cycle, however, due to purging and evolutionary changes in the model, the degree of reuse may vary. We observed an average of 40-80% reuse within review cycles.

10 Related Work

In one form or another, the view integration problem has been worked on by numerous researchers. This section discusses their works and also discusses in what ways their works differ from ours. It is important to note that other view integration approaches are not independent or in any way orthogonal to ours. [Sage and Lynch 1998] wrote that “unfortunately, there appear to be no detailed definitions that distinguish between various types of integration, and this may appear to make the subject disjoint. ... [However] integration is generally always being performed, but it is not clear as to where it is performed or how to accomplish it successfully.”

This section presents and discusses twelve related view integration approaches. Instead of discussing them individually, we found criteria on how to enable their comparison in a more meaningful fashion. Those criteria are partially based on our framework and could be considered subjective. However, even in case of subjectivity, these criteria enable reasoning about how their approaches relate to ours. The set of criteria may not be complete, but they cover a number of important architectural considerations, and can serve as a baseline for future work.

10.1 Overview

View integration is part of every aspect of the development life cycle and, thus, our work, and the related work presented in this section, fit somehow into the greater scheme of the view integration problem. Because of the depths of the integration problem it is far out of the scope of this work to present a complete survey. We start this overview with the works of Sage and Lynch [Sage and Lynch 1998] because they are one of the few people who have attempted to summarize all key aspects of integration even across the boundary of software. Their work on *Systems Integration and Architecting* covers integration aspects, principles, and practices on the system level going far down into details of systems and software development. Their recent summary is an excellent work of 50 pages and we could not possibly provide a better one here. In that work, they talk about the need for integration on the systems engineering level and present the results and findings of numerous researchers. Systems engineering

Table 8. Comparison of View Integration Approaches

		Egyed Dissertation	AAA	Belhouche-Lemus	Delugach	Engels et al.	JViews (MViews)	Keller et al.	SADL	SAAGE	SCED	ViewPoints	VisualSpecs
Integration	Ergonomics												
	Analysis												
	Synthesis												
	Automation												
Modes	Identification												
	Resolution												
Media	Formal Notation												
	Graphical Notation												
	Document												
View Dimensions	Abstract/Concrete												
	Generic/Specific												
	Structural/Behavioral												
Life-Cycle	Requirements												
	Architecture												
	Design												
	Coding												
Flow	Forward												
	Reverse												
	Change Management												
Scalability	Basic												
	Within Cycle												
	Evolutionary												
Other Models	Process Support												
	Property Support												

differs from software engineering in that it tries to cover all that is offered by the latter but more. It also covers hardware aspects and how software is integrated with it. With that in mind they address the need of model integration, as it is addressed in our work, although the scope of their work did not permit more than an overview.

Since consistency-checking approaches are abundant, we decided to place the focus of this section onto automatable approaches. With that we exclude all manual validation and verification techniques such as inspection [NASA 1993], review boards [AT&T 1993] and others. Boehm's paper on verification and validation techniques [Boehm 1989] provides an excellent overview of such techniques.

10.2 Comparison of View Integration Approaches

In the following subsections we discuss automated or automatable view integration approaches. In our evaluation we considered twelve cases. Although, those cases do not all address consistency checking, they do solve some significant portions of it. For instance, we decided to also include approaches that only automate transformations (e.g., SCED [Koskimies et al. 1998]). The types of approaches we considered are depicted as columns in Table 8. The first column represents our approach; the remaining columns represent related approaches.

As we indicated previously, we defined a set of criteria in dealing with view integration issues (and consistency checking issues in particular). The eight criteria were then refined into 2 to 5 characteristics each. Table 8 shows the ratings of all approaches. The following ratings were given: *none*, *weak*, *strong*. The rating *none* implies that that criterion is not satisfied by that approach. A *none* rating is also given when the criteria does not apply or is insignificant. The *weak* and *strong* ratings are given when some support or extensive support is available. We believe that no view integration approach is complete and, therefore, decided not to give any stronger ratings. The following approaches were evaluated (listed in alphabetical order)

- **AAA (Architect's Automated Assistant):** Abd-Allah and Gacek address the problem of component integration [Abd-Allah 1996] [Gacek 1998]. That task involves the interpretation of components and their characteristics followed by reasoning about potential inconsistencies

among them. Their approach is very unique since they investigate inconsistencies on a very high-level. For instance, if a software product is composed of a number of components—some of which may be COTS (Commercial-of-the-Shelf)—then based on certain properties of those components potential inconsistencies could occur.

- **Belhouche-Lemus:** Belhouche and Lemus take a more formal approach to view integration in the context of statecharts and dataflow diagrams [Belhouche and Lemus 1996]. Their work reflects the opinion that views are independently created and analyzed, however, formal transformations should be used to support consistency checking (we have taken a similar stance). Their approach seems automatable, although, they do not provide tool support at this point.
- **Delugach:** Delugach took two types of diagrams, data-flow diagrams (DFD) and class diagrams, and transformed them onto a conceptual graph [Delugach 1996]. What makes his work unique is that he then verbally describes the relationships of components in that conceptual graph. That process can be automated and a modeler can then read those descriptions (which are in plain English) and reason about their validity.
- **Engels et al:** Ehrig, Engels, Heckel and Taentzer worked on the problem on how to merge object diagrams into more generic types of diagrams [Ehrig et al. 1997]. Object diagrams are combined stepwise into a more generic model by merging two views at a time until all of them are merged. In doing so, they explore different generalization paths along the way. Their work is very useful for the generalization of specific views into generic ones. Besides, their approach can also be used for showing how a method changes an object diagram over time. For instance, if a method (function) adds, removes, modifies a current object model then this kind of knowledge can be used for consistency checking.
- **JViews (MViews):** Grundy, Hosking, Mugridge, and Warwick have created several modeling environments—most notably JViews (a successor of MViews) [Grundy et al. 1996]. They primarily focus on code and lower-level designs, however, were able to address a series of view integration problems in that domain.

- **Keller et al.:** Schönberger, Keller, and Khriiss concentrated on scenario diagrams and how they can be transformed and merged into statechart diagrams [Schönberger et al. 1999] [Khriiss et al. 1998]. They do this by matching method calls to state changes. Although they not provide tool support, they present detailed algorithms that also address concurrency issues. Even though they did not have view integration in mind, their work is fundamental when it comes to view integration.
- **SADL:** His approach is different to the previous one in that he provides a formal language and transformation laws with which transformations are guaranteed to remain consistent [Riemenschneider 1999]. In particular, his work is geared towards consistent refinement of abstract descriptions to a level of detail that allows its direct implementation.
- **SAAGE:** Robbins, Medvidovic, Redmiles, and Rosenblum used the UML notation and modeled their architecture description language (ADL) in UML [Medvidovic and Rosenblum 1999]. Abi-Antoun and Medvidovic then took that translation specification and automated it in their SAAGE tool [Abi-Antoun and Medvidovic 1999]. Like SADL's work, their work guarantees consistent refinement (at least initially).
- **SCED:** The work of Koskimies, Systä, Tuomi, and Männistö is very similar to Keller et al. since it also involved sequence to statechart transformation [Koskimies et al. 1998]. Although, the techniques of Keller's group are more in-depth, Koskimies's group was able to provide tool support (SCED).
- **ViewPoints:** The work of Easterbrook, Finkelstein, Hunter, Kramer, and Nuseibeh on ViewPoints [Easterbrook and Nuseibeh 1995] is also close to ours in that it presents some views and corresponding rules to identify inconsistencies within and between them [Finkelstein et al. 1991] [Nuseibeh et al. 1994]. Their strongest contribution is a framework with which they provide mechanisms for detecting, classifying and resolving inconsistencies. Their work emphasizes the early parts of the life cycle strongest—requirements and architecture—and they provide several tools in the process.

- **VisualSpecs:** The work of Bourdeau, Cheng, and Wang, acknowledges and addresses a deficiency of modeling languages, like UML or OMT, which has to do with the lack of precision and formalism [Cheng et al. 1995] [Wang and Cheng 1998] [Wang et al. 1997]. They propose ways on how to eliminate that problem by integrating formal methods into OMT [Rumbaugh et al. 1991]. So they substitute object models with algebraic specifications, various OMT semantics with algebraic semantics and instance diagrams with algebras. Reasoning about diagrams can then be shifted onto the more precise formal specifications.

The following sections will discuss the above approaches in context of our evaluation criteria. We defer the evaluation of our approach to Section 11.1.7.

10.3 Integration Criteria

The *Integration Criteria* addresses the extent of how view integration has been addressed. In particular we are interested in the ergonomics, analysis, synthesis, and automation of individual approaches.

Ergonomics addresses human-computer interface issues. For instance, how is information communicated to the users and how are users able to influence the integration process? JViews (a successor of MViews) [Grundy and Hosking 1996] [Grundy et al. 1996] handles many facets of ergonomic issues related to user interactions [Grundy et al. 1998]. As such, they also deal with issues on how to present inconsistencies and how to act in their presence. Similarly strong, from an ergonomic point of view, is the work on ViewPoints [Finkelstein et al. 1994] which addresses the life cycles of inconsistencies. AAA only handle simple user-interaction in defining and analyzing consistency issues and thus received a weak rating. All other approaches do not address ergonomic issues in context of consistency checking.

Analysis addresses an approach's ability to reason in the presence of a model description. AAA defines and then analyzes conceptual features of components. Based on the values of those features, potential inconsistencies are identified. Belhouche-Lemus's approach transforms diagrammatic

descriptions like dataflow diagrams and statecharts into a formal representation. In the context of that representation, diagrams can be analyzed and inconsistencies identified. JViews takes a different approach where diagrammatic and textual description (e.g., source code) are captured in a well-defined model repository (the base model). It is in that model where inconsistencies are identified based on the semantic relationship between model elements of that model. ViewPoints takes multiple approaches to consistency detection and handling, although those mostly involve the description of a problem in some formal language and its detection and resolution in context of that language. VisualSpecs' approach is analogous to Belhouche-Lemus and ViewPoints. Diagrammatic types of views are transformed into algebraic specifications, diagram instances are transformed into algebras, and the semantics that hold between diagrams and their instances are transformed into algebraic semantics. VisualSpecs then uses those algebraic semantics to reason about the consistency between algebras. SADL's approach is very different in that no explicit analysis component exists but instead synthesis implicitly also covers analysis. His approach builds on well-defined semantics and uses proof-carrying transformation steps that guarantee that transformations remain consistent. SADL therefore merges synthesis and analysis.

The *synthesis* criteria addresses whether consistency checking only involves the direct comparison of elements or whether transformation is part of that process. We found that most integration approaches use some form of transformation. In many cases, transformation involves the translation of some given model into a more rigorously defined formal notation (e.g., Belhouche-Lemus, SADL, ViewPoints, VisualSpecs). Reasoning is then done in context of that formal representation. In many other cases, transformation is used to provide a common repository (e.g., Engels et al., Keller et al., SAAGE, SCED). In some cases, no explicit transformation is used. Instead model information is annotated (made richer) to allow extended reasoning. For instance, AAA describes components in form of conceptual features that are added to component definitions; JViews provides a common repository and base model to enable a uniform reasoning environment; and Delugach creates descriptions of models in plain English (a generic reasoning environment for manual analysis).

Most integration approaches support automation and have tool support. We gave a strong rating when tool support is available for modeling, analysis and, inconsistencies detection. We gave a weak

rating when only partial tool support is available (e.g., SCED only supporting synthesis but not analysis) or when no tool support is available but algorithms are defined (e.g., Keller et al.).

10.4 Modes Criteria

The *modes criteria* indicate the usefulness of an approach in dealing with the entire life cycle of inconsistencies. Most importantly are the detection and resolution of inconsistencies.

All approaches are useful for inconsistency *identification*. Some approaches received a strong rating if they defined analysis and synthesis rules (e.g., consistency rules and transformation rules) in such detail that inconsistencies can be identified in a fully automatable process (e.g., AAA, Belhouche-Lemus, JViews, ViewPoints, VisualSpecs). The weak ratings are given if only partially consistency checking is automatable (or automated) (e.g., Delugach, Engels et al., Keller et al., SAAGE, SCED). For instance, in case of SCED only transformation is automated which simplifies comparison considerably. Nevertheless, the actual comparison still has to be done manually. A very exceptional case is SADL's approach. His approach gives no direct support for automated consistency checking since he uses a process that guarantees consistent refinement. However, since his synthesis steps guarantee consistency, he received a strong rating.

Inconsistency *resolution* is the natural extension to inconsistency detection. Resolution is, however, also very hard to accomplish. Only SADL's approach enables the automatic resolution of inconsistencies (strong rating), although, this rating may be misleading since his approach avoids having to deal with inconsistencies. Nevertheless, his approach supports a form of view integration that does not require any inconsistency handling. All others have to handle inconsistencies somehow. JViews and ViewPoints provide some support (mostly user interface and process type) to deal with inconsistencies. That support helps but does not provide full automation (partial rating). All other approaches do not support inconsistency resolution.

10.5 Media Criteria

The *media criteria* addresses the types of models required to support view integration. Most integration approaches support *graphical* modeling languages such as the UML. Some other approaches are at least partially based on *formal* notations. Even in cases where synthesis and analysis requires a formal notation (e.g., Belhouche-Lemus, ViewPoints, VisualSpecs), graphical counterparts exist (with the exception of SADL). Only two approaches could be useful for informal types of media like *documents*. Delugach's approach produces such media for analysis and ViewPoints uses some structured form of requirements evaluation. Pure document-driven consistency checking approaches are not handled in this work. Please refer to [Boehm 1989] for an overview.

10.6 View Dimensions Criteria

The *view dimension criteria* covers the types of diagrams that could be supported by various approaches. In our works, we distinguished inconsistencies between abstract and concrete views, generic and specific views, structural and behavioral views, and within a single type of views.

Whereas most integration approaches showed strong similarities for some of the previous criteria, when it comes to the applicability we find strong deviations. Only one approach handles consistent refinement (*abstraction*) in a strong manner—SADL. Two other approaches have partial support for it. We gave partial support when the consistency checking between refinement and abstraction assumes one-to-one traces. For instance, JViews can validate consistency between a source code type of view and a class diagram type of view (note that there are differences to the UML definition of class diagrams). However, JViews assumes the mapping from source code to class diagram to be one-to-one. Thus, a source code class corresponds directly to a class diagram class and so forth. We believe that this case constitutes a proper abstraction/refinement relationship since the source code level shows information not present in the class diagram level, although, their approach only deals with the most trivial form of abstraction (weak rating). SAAGE also got a weak rating since it transforms C2 architectural diagrams into UML object diagrams. That transformation could be considered a refinement since C2 implicit information is explicitly stated in UML. Nevertheless, like in the case of JViews, the

distinction between abstract and concrete is very weak. Accordingly weak are the findings in relevance to consistent refinement.

We found that view integration approaches tend to be most useful for consistency checking between *generic and specific* views. Engels et al's approach merges object diagrams (instances) together using some type of reference model for naming and consistency issues. Although this work does not actually address consistency checking, their work can be fundamental in generalizing specific views into more general ones. Their approach uses a step-wise generalization, creating intermediate models in the process. The result of the generalization process is an integrated object diagram with more generic properties. Their approach is also unique in that they relate object diagrams to one another and define what kind of methods cause their differences. For instance, consider a constructor method in context of object diagrams. If the input to a method is an empty object diagram (no objects), and the output of the method is an object diagram with one object then one can infer that that constructor method created an object. That additional information about the relations between object diagrams can then be used to further argue about generic consistency issues.

Keller et al's approach and SCED's approach are very similar. Both take (multiple) sequence diagrams (which are specific views) and merge them into more generic statechart diagrams. That generalization process interprets message calls between objects as triggers of state changes. Given a collection of sequence diagrams, repetitive patterns between and within them can be used to cross-reference and merge their transformations into statechart diagrams. When we adopted their approaches, however, we found that their approaches have a number of deficiencies. For instance, the assumption that a method call triggers a state change is true in many cases but there are exceptions: A state change in a class may also be triggered by a state change in another class without them having to communicate. There are also hidden complexities with concurrency issues. For instance, if two objects run concurrently and they exchange messages then it is important to know at what time that state changes occurred. The SCED approach adopted a rather simplistic solution whereas the Keller et al. approach tried to address some concurrency issues.

VisualSpec is the fourth approach to receive a strong rating. As it was already discussed, it transforms informal diagrammatic schemas into formal algebraic specifications and algebras. Types are transformed into algebraic specifications and instances are transformed into algebras. The type/instance relationship is similar to our generic/specific relationship. VisualSpec then defines diagrammatic semantics in form of algebraic semantics. Consistency checking involves the validation of algebras in context of those algebraic semantics.

Delugach's approach is the only case that received a partial rating. The remaining approaches did not pass the threshold. Delugach's approach interprets diagrammatic views (e.g., class diagrams) and converts their contents into plain English. The produced documents can then be read sequentially and in that context it can be reasoned about inconsistencies. His approach, like some others, do not actually provide automated consistency checking, however, through the transformation process it is claimed that some types of inconsistencies are more easily identifiable. We believe that his approach is only useful in context of generic and specific views since, on that level, the semantic relationships are still understandable enough to enable a direct comparison. In case of abstraction and structuralization, we find that a simple one-to-one comparison is of little use. Transforming a diagrammatic view into plain English does not simplify that.

Consistency checking between *structural and behavioral* views is among the weakest supported forms we observed. We only gave the approaches of Belhouche-Lemus and ViewPoint a strong rating, not because they explicitly investigated consistency between structural and behavioral views but because the formal notation they support allows behavioral reasoning.

10.7 Life-Cycle Criteria

We also evaluated related approaches in context of their usefulness during the software life cycle. The *life-cycle criteria* is therefore split into requirements, architecture, design, and coding. Note that it is not our intention to favor a particular process model. The sequencing of specifications should not be interpreted as waterfall-like. Instead, we chose those criteria to denote an approach's usefulness in dealing with the different types of semantic models present during those development stages.

The *requirements* criterion indicate the usefulness of integration approaches in dealing with high-level types of information. Requirements are typically captured less formally (often in plain English) and typically cover a wider range of development issues even outside the boundary of the actual product. Almost all approaches with the exception of ViewPoints cannot be used on that level. Even ViewPoint resorts to a formal specification of requirements and evaluates them in that context.

The *architecture* captures high-level components, their relations as well as their roles and responsibilities. Architectural descriptions are still abstract enough so that their components are intuitive parts of software systems. Only one of the approaches we evaluated could be considered purely architectural. The AAA approach evaluates architectural components that could be legacy systems, in-house developed components, or COTS (commercial-off-the-shelf) components. AAA analyzes consistency between them by analyzing conceptual dependencies of their characteristics. For instance, if a component is concurrent and it is composed with a component that is sequential, the concurrent component may potentially use the sequential one in a concurrent fashion causing faults.

The ViewPoints' approach also seems more architectural although that was not explicitly stated as such by the authors and we gave them a weak rating here. Both SADL and SAAGE come from the architecture community. The emphasis of their works is strongly motivated in the context of architecture description languages. Nevertheless, both claim that automated refinement even through the code level is possible. However, it is our opinion that their languages get rather awkward if used in the small. Both approaches only support one type of modeling language and, at this point, they are not sufficiently equipped to deal with lower-level concerns (e.g., design patterns [Gamma et al. 1994], etc.).

Most approaches are at least partially useful for *design* modeling (except for AAA) since they tend to use some design-level constructs. As such Belhouche-Lemus is based on dataflow diagrams and statecharts; Delugach evaluates class-like diagrams; Engels et al. is based on object diagrams; JViews also uses class-like diagrams; Keller et al. and SCED use sequence and statechart diagrams; and VisualSpecs is based on OMT (OMT is similar to UML). ViewPoint got a weak rating since we saw only little support for design type diagrams although they state that some exists. SAAGE and SADL do not support design notations, however, both claim their notations to be useful for designs (weak ratings).

10.8 Flow Criteria

The direction of development flow and how changes are propagated are two important aspects of view integration. We find that all techniques support *forward engineering* (more waterfall-like scenario). Also, most approaches could be used for *reverse engineering*. The exceptions are the four architectural approaches AAA, SADL, SAAGE, and ViewPoints. In case of AAA, reverse engineering does not make sense since it is needed for component composition. In case of SADL, SAAGE, and ViewPoints, reverse engineering is probably possible but because on their reliance on formal notations without common design notations it might be harder to realize (it is hard to relate implementation construct to architectural constructs).

Change management, the last flow criterion, denotes the ability to make changes at some later point. For instance, what happens if one of VisualSpecs diagrams is changed after an algebraic specification was generated and it had been instantiated? Naturally, the simple solution would be a re-generation of the algebras based on the new diagram, however, that approach would require a complete re-evaluation of that diagram for consistency checking. If only a small part of the diagram changed than not all should be re-evaluated. Change management, therefore, requires a smart and well-defined way of dealing with evolutionary changes. We did not find strong support for change management among the dozen approaches we evaluated. Two approaches, JViews and ViewPoint, defined that notion and have incorporated some solution, however, that problem remains largely unsolved.

10.9 Scalability Criteria

The scalability criteria are meant to evaluate solutions with respect to the single largest challenge of view integration—that of scalability. View integration does not scale well and in this work we discussed a number of reasons why. Evaluating related works we find that scalability has not been addressed in detail. The most *basic scalability* problem of view integration is the fact that all model elements (and all views) have to be compared with all other model elements (and views). Only JViews provides the start of a partial solution by introducing a base model that compactly handles modeling information. We found

that extensive reuse during consistency checking is the key for improvements. No other view integration approach addresses scalability in a similar strong manner.

10.10 Other Models Criteria

The final criteria with which we evaluated related works are their usefulness towards other areas of software development. Note that our emphasis thus far was about product models that describe software systems. Other types of models include process models and property models.

Process models describe development activities. Examples of process models are the waterfall model [Royce 1970] and the spiral model [Boehm 1988]. Integration could, for example, validate the conformance of a development process to the actual process. We found that only JViews (integrated with Serendipity II [Grundy and Hosking 1996]) provides process coverage.

Property models, additionally, evaluate non-functional aspects of software systems. For instance, software properties could be performance, reliability, security, etc. AAA, SADL, and SAAGE have some support for those although they do not model them explicitly.

The works presented above do not cover the complete picture of what is going on in view integration. However, it gives an overview of the major approaches. The diversity of the work above is another reason why this work thrives not to just add another technique. Many approaches described in this section are excellent and, thus, our work tries to also take the best of what already exists.

11 Evaluation, Future Work, and Summary

11.1 Evaluation

A critical aspect of a new approach is its evaluation. In this section, we will discuss the kinds of validations we have performed and the observations we have made. Evaluating a body of work like this embodies a number of complexities, especially since we did not set off with a well-bounded problem nor did we end up with a well-bounded solution. Thus, a mathematical proof of correctness cannot be cast across the entirety of our work. Nevertheless, validation is possible and we chose to follow a series of dimensions:

- Evaluating Transformation Techniques
- Evaluating Comparison Methods
- Evaluating Effectiveness, Efficiency, and Reliability
- Evaluating Scalability
- Evaluating Applicability outside UML Domain
- Evaluating UML's ability to support analysis
- Evaluating in Context of Other Approaches
- Breadth over Depth
- Technology and Research Transfer

11.1.1 Evaluating Transformation Techniques

This work emphasized four transformation types—abstraction, generalization, structuralization, and translation—that are used to bridge the eight types of UML views we are currently supporting (plus a ninth type of view called C2 [Egyed and Medvidovic 1999]).

Our abstraction process was validated in a number of dimensions. First, we validated its rules by analyzing their semantic implications along the lines discussed in Section 7.3.1.3. Furthermore, we applied our technique on a series of models and evaluated the results in the context of both consistency checking and reverse engineering:

- i. Reverse Engineering: our abstraction technique can be used to reverse engineer high-level models out of more concrete (lower-level) models.
- ii. Consistency Checking: our abstraction technique can also be used to enable consistency checking between existing concrete and abstract models.

Although the two uses (reverse engineering and consistency checking) have different goals in mind, the validation of the principles of our abstraction mechanism remains identical. Some of the models we used to evaluate abstraction were provided by industry (e.g., the validation of a small part of a Satellite Telemetry Processing, Tracking, and Commanding System (TT&C) [Alvarado 1998]). Other models were self-made. For instance, we reverse engineered and abstracted our own tools like AAA [Gacek 1998] and UML/Analyzer (see Section 8) and we validated consistency among half a dozen UML models with up to 100 classes. We also applied our abstraction concept towards consistency checking between UML designs and the C2 architecture description language [Taylor et al. 1996] where we were able to demonstrate C2 structural and behavioral violations [Egyed and Medvidovic 2000]. It must be noted that C2 was neither created by us nor was our abstraction technique build for that use in mind. Finally, our abstraction techniques were (and still are being) validated by a number of research institutions and we made improvements based on their comments (Mitre Organization and Rational Software must be noted).

We invented the relation abstraction technique in collaboration with Rational Software in 1997 [Egyed and Kruchten 1999]. Independently, at a later time, the group of Racz and Koskimies [Racz and Koskimies 1999] came up with a class compression method that exploits the same concepts as our relation abstraction technique. They, however, did not create automated abstraction rules nor did they create tool support for automated abstraction in their work. Nevertheless, we see their work as a form of validation of ours since they independently made similar observations about class patterns.

Finally, Rational Software adopted our relation abstraction technique and built its own implementation called Rose/Architect [Egyed and Kruchten 1999] (implemented by Ensemble Systems for Rational Software). Our tool (UML/Analyzer) incorporates all features of Rose/Architect and also addresses classifier abstraction, reliabilities, cardinalities, complex rules as well as scalability issues.

To support structuralization and generalization, we adopted transformation methods of other researchers [Koskimies et al. 1998] [Schönberger et al. 1999] [Ehrig et al. 1997]. We validated their transformation method in the context of consistency checking and found a number of deficiencies for that use. It must be noted that some of those transformation methods were not built for consistency checking although a few also claimed their approach to be useful for that purpose (e.g., SCED [Koskimies et al. 1998]). Our work showed that transformation requires more than the simple conversion of modeling information. Thus, the context of our framework enables other tools (e.g., like SCED) and methods to be evaluated for their usefulness and fitness towards view integration. Our framework also enables other tools to be used for consistency checking. Our framework thus combines view integration approaches.

11.1.2 Evaluating Comparison Methods

There exists an accepted notion of what consistency means and what it does not mean. For instance, the case of a model element that was not properly refined denotes a very clear case of an inconsistency. Our work, therefore, did not have to validate what consistency is. Our work only had to validate whether our approach is capable of identifying it. In our framework, consistency checking is based on consistency rules and on a transformation's ability to transform model elements to make them directly comparable in the context of those rules.

We have built consistency rules based on the semantic dependencies between model elements. Those semantic dependencies were derived out of the UML definition itself as well as examples of its usage. We have then validated consistency rules in context of numerous examples. In sections 7.5.2 and 8 we showed some of those examples. We have also applied our rules to validate consistency between UML and the C2 architecture description language as we discussed above. This implies that our rules are generic enough to scale to other types of views outside the UML domain.

Consistency rules and their ability to enable direct comparisons between model elements of same and different types of views have been used by other view integration approaches. Defining rules is mandatory in defining the proper or improper usage of models. Consistency rules therefore describe the invariants of software modeling. Our framework extends consistency rules with transformation methods that allow their direct comparison in cases where others fail. In this work, we demonstrated that

transformation can be successfully coupled with consistency checking, resulting in a broader coverage of potential inconsistency types. Our approach also stands out in its ability to detect inconsistencies no other approach can detect. Section 11.1.3 below will discuss those.

We also built tool support to automatically identify eight types of inconsistencies. The inconsistencies found are fundamental types (meaning we believe that they address the most important consistent refinement problems). To the best of our knowledge, no one else has been able to identify them yet.

11.1.3 Evaluating Effectiveness, Efficiency, and Reliability

We would have liked to support this work with strong indications of its effectiveness, efficiency and reliability. The main problem is that such numbers would only make sense in comparison to other view integration techniques. To the best of our knowledge no such comparison exists, primarily because view integration approaches tend to be very different in their coverage. For instance, our approach is able to detect a series of inconsistencies that other approaches cannot detect. It is, therefore, not meaningful to compare efficiency or reliability numbers in that context. Despite that, we were able to reason about those factors. This section summarizes those.

We measure effectiveness in terms of our approach's ability to detect inconsistencies that have not been addressed by other approaches. For instance, our abstraction method enables the comparison between high-level and low-level diagrams. This type of comparison is very important during refinement and maintenance of software projects. Validating a higher-level diagram based on a lower-level diagram ensure that properties of higher-level diagrams are maintained in a consistent fashion.

We found 21 types of inconsistencies between diagrams at different levels of abstraction. And we found additional 30 types of inconsistencies between other types of diagrams. We neither claim that this list is complete nor do we claim that our approach could replace other approaches. Quite the contrary. View integration is a complex field. Our approach uses a unique approach, however, is not complete at this point. The works of [Nuseibeh et al. 1994], [Grundy and Hosking 1996] and others show that other approaches are able to handle other situations quite well—in some cases even better than ours (also refer to related works in Section 10). For instance, the works on ViewPoints [Finkelstein et al. 1991] allows the

detection of inconsistencies that are closer to the requirements engineering domain. Our approach handles product models only. However, at the same time it cannot be claimed that our approach is worse. Instead we find that both approaches have respective advantages and they complement each other.

We measure efficiency in the speed in which inconsistencies can be identified. Here we have to consider two dimensions—regular and evolutionary efficiency. Our work on view integration is unique in its approach to scalability. We found that scalability is a problem that is discussed extensively in the view integration community, however, is rarely ever addressed. Our work shows how increased reuse of transformation results and a reduced redundancy model are valuable contributions that have strong impact onto the efficiency on consistency checking. Instead of having to validate the same model element for every consistency checking cycle, past transformation and/or comparison results can be reused. Our approach thus improves evolutionary efficiency through extensive reuse.

Efficiency also has to do with the speed of inconsistency detection within a single validation cycle (non-evolutionary). In that context we studied consistency checking between abstract and concrete diagrams (via our tool UML/Analyzer) and found significant improvements of automation. Our tool was able to perform abstraction transformations in fractions of seconds that manually would have taken hours to perform (recall Figure 129).

Although our process strongly improves the speed of transformation and consistency checking, its impact onto reliability has to be studied further. We applied our consistency checking process onto a series of examples and at the same time performed those consistency-checking tasks manually. We then compared the results of both approaches in terms of false errors or oversights. Interestingly, we observed that both approaches were able to detect inconsistencies that the other was not able to. We also found some cases where our approach detected an error where there actually was none. However, in the cases we evaluated, we found that the tool was able to detect most of the same errors as the human but did so much faster. Given the existence of false errors, one nevertheless has to see the findings of our approach as indications of potential inconsistencies instead of factual ones (although in some cases the findings are quite reliable). This constraint does not disqualify our approach for two reasons: (1) having support which

locates potential inconsistencies in a very short amount of time is still better than having to find all of them manually; and (2) the reliability of our approach can be increased (see Section 11.2).

11.1.4 Evaluating Scalability

To date, a number of view comparison approaches have been proposed but the major problem in automating them is scalability. We do not make the claim that our approach is free from that problem; however, this work discussed a series of measures we have undertaken to address it.

The basic structure of our view integration framework already addresses scalability. We use mapping to restrict what to transform/compare (instead of random comparison) and we use transformation to simplify how to compare. We integrated our techniques to remember past transformation results and, thus, increased the amount of transformation reuse. This implies that instead of having to re-verify the entire model every time a change is introduced, we only have to compare the effects of changes with the existing other parts of a model—from an evolutionary standpoint a very significant improvement. We demonstrated on an example how 9 classifiers, 9 relations, and 14 trace links could be reduced to only 5 classifiers, 5 relations, and 6 trace links.

To address evolutionary scalability, we discussed the concept of a reduced redundancy model that limits potential inconsistency introduction between user-defined elements and derived elements during evolution (a change propagation issue). Instead of having to define 36 transformation methods for the 9 types of views we are supporting, we demonstrated how a subset of them (only 14 transformation methods) are sufficient to enable the same coverage. Finally, we defined a transformation framework with associated reliability measures to identify promising transformation paths. Most of those scalability concepts have also been implemented in our tool UML/Analyzer demonstrating the automatability of our concepts.

11.1.5 Evaluating Applicability outside UML Domain

This work primarily concentrated on using UML, although, we claimed that our approach is more generally applicable. We therefore integrated another type of view into our framework to show two things: (1) our approach's ability to handle other heterogeneous types of views; and (2) our approach's ability to link two separate model worlds to ensure consistency between them. In [Egyed and

Medvidovic 2000] we showed how our approach can be used in the context of an architecture description model such as C2 [Taylor et al. 1996]. That work combines C2 to UML integration to enable a formal approach to software development as a complement to a generic (less formal) development approach (e.g., UML).

We use the translation method of Abi Antoun-Medvidovic [Abi-Antoun and Medvidovic 1999] to transform C2 models into an UML equivalent and we then use our abstraction method to enable a comparison between the UML equivalent of C2 and the UML abstraction. This type of view comparison corresponds to scenario d) of our view transformation framework in Figure 43 where both views (the C2 architectural view and the UML class diagram view) have to be transformed into a third type of view (an UML supported intermediate model [Medvidovic et al. 2001]) to enable their direct comparison in context of that third intermediate representation.

11.1.6 Evaluating UML's ability to support analysis

In the process of using UML for view integration, we had an excellent opportunity to evaluate UML's suitability towards view analysis. In particular we questioned whether UML is suited to support automated consistency checking. This section discusses the difficulties of UML in supporting automated consistency checking. Although many of the difficulties we found could be minimized to a large extent by using UML's extensibility mechanism, we find that there are some exceptions. In particular we find the current UML meta-model insufficient in describing and handling transformation results. Furthermore, the scalability measures discussed here can only be supported with great difficulties.

11.1.6.1 Reduced Redundancy Model

We discussed reduced redundancy models as a means of capturing and maintaining derived information without having to worry about their maintenance. The better a reduced redundancy model is implemented, the more effective and scalable is view integration in terms of evolution. The only disadvantage of the reduced redundancy model is that it cannot be fully supported in UML. UML's meta model has a clear definition on how model elements have to interrelate and the concept of a bridge (a

bridge was required in between model elements) is not supported. Bridges are located between model elements and each bridge may have two or more attachment placeholders for model elements (ports).

Not being able to build a good reduced redundancy model does not disable automated consistency checking, however, it does decrease its effectiveness. In Section 7.7.3 we showed two cases of a reduced redundancy model—one that could be built using UML and a better one that currently cannot be built in UML. To enable a better-reduced redundancy model, we needed to augment the UML meta-model, thus violating the UML standard. The changes have, however, only little impact onto how the user sees UML. Nevertheless, this example shows a case, where existing UML concepts are not sufficient in supporting view analysis.

11.1.6.2 Explicit and Implicit Treatment of Traces

Trace information between views are very important in our view integration framework. For our framework, we need abstraction, generalization, structuralization, translation, origin, and interpretation traces. UML only has explicit definitions for one of them: abstraction. Abstraction is defined in UML as a dependency between model elements that is stereotyped as «Abstraction». UML supports generalization partially as instance and type relationships. This relationship is, however, not an explicit trace nor does it apply to all specific model elements. For consistency checking it does not make a difference whether traces are treated explicitly or implicitly, however, it turns out that implicit links are only supporting those elements they are attached to and not others. We use UML's extensibility mechanism to define explicit dependency links for generalization, structuralization, translation, and interpretation. Those links are stereotyped accordingly. Explicit dependency links can be used on all model elements.

11.1.6.3 Ambiguous and Partial Interpretations

Consistency checking takes interpretations and compares them with the (user-defined) original model elements. In case of differences, inconsistencies are found. Comparison was discussed as being about equality; however, there are exceptions as in the case of ambiguous results. For instance, a call in a sequence diagram could either be a query or an action (in a statechart diagram) since we do not know

whether it causes a state change or not. Since this distinction is important for comparison between the sequence and statechart diagrams, capturing transformation results has to handle that ambiguity. Similarly, there are cases where transformation yields partial results. For instance, in some cases of abstraction the direction of interaction may be computed without knowing the type of the interaction.

UML does not have an explicit notion of how to describe interpretation relationships between user-defined elements and derived elements; nor does it have an explicit notion on how to describe ambiguous and partial transformation results. UML's extensibility mechanism can mitigate some of those problems. For instance, a dependency relationship stereotyped as «Interpretation» can be used to simulate the interpretation relationship. The issue of partial or ambiguous transformation results is, however, more complex since UML has no concept on how to define some variations of them. For instance, it is not possible to define a derived relation of unknown type with only some cardinality information attached.

11.1.7 Evaluating in the Context of Other Approaches

We also evaluated our work in context of other approaches. In Section 10 we discussed a dozen related view integration approaches and rated their strengths and weaknesses in the context of 24 criteria. In the following we briefly discuss how our approach competes in this scheme.

- **Ergonomics:** Our work emphasizes the technology aspect of view integration at the expense of user interface aspects. We therefore only partially address this problem. The works on JViews and ViewPoint are significantly better.
- **Analysis:** Our work introduces new techniques for enabling automated consistency checking. Although other approaches also have strong consistency checking support, our work is able to detect a series of inconsistencies others cannot. Our work therefore complements others.
- **Synthesis:** Like analysis, our work introduces new synthesis methods that enable improved transformation. Our work therefore complements others.
- **Automation:** Not many approaches have full tool support from inception to detection of inconsistencies. Only five out of twelve other approaches have a similar coverage.

- **Identification:** Like many other approaches, we provide extensive assistance for inconsistency identification. As discussed above, our approach is able to detect inconsistencies others cannot.
- **Resolution:** We currently do not support automated inconsistency resolution, however, we see a lot of potential for that. The works on JViews and ViewPoints are good guides.
- **Formal Notation:** Our approach is based on diagrammatic views. Currently we do not envision the need for formal notations since they are less useful for design languages.
- **Graphical Notation:** We provide extensive support for graphical notations. Since we are supporting 8 types of views (not including C2), our work has attempted the largest view coverage yet. Other approaches use at most two types of views.
- **Document:** Our emphasis is on graphical notations. Other approaches are more suited for document-based validation and verification.
- **Abstract/Concrete:** Our approach provides the strongest support for consistency checking between abstract/concrete diagrams to date.
- **Generic/Specific:** We also provided support for consistency checking between generic and specific views. Although some other approaches are stronger, we must note that we have not investigated that problem nearly as much as abstraction.
- **Structural/Behavioral:** Consistency checking approaches for structural and behavioral diagrams are generally weak across related views. Ours does not yet contribute extensively either.
- **Requirements:** We have no support for requirements modeling in our thesis. We have however started to investigate this issue in [Medvidovic et al. 2001] and [Gruenbacher et al. 2000].
- **Architecture:** We integrated our approach with the C2 architecture description language to validate consistency between C2 architectures and UML refinements [Egyed and Medvidovic 1999] [Egyed and Medvidovic 2000].
- **Design:** We strongly support design modeling (8 out of 9 views).
- **Coding:** We directly do not support coding, however, through reverse engineering we are able to generate low-level class diagrams which can be abstracted and validated.

- **Forward:** Strong support like all others.
- **Reverse:** Our approach is equally useful for reverse engineering since we created transformation methods and built consistency rules that work independent of development flow.
- **Change Management:** We have partially addressed the change management problem of view integration. We find that our approach adds a new and unique way of dealing with this issue via our improved reuse.
- **Basic:** In terms of scalability our approach adds new concepts. We showed that the basic scalability problem of having to compare all views with all others can be improved through complex transformation and improved reuse. Basically no other approach has addressed that issue.
- **Within Cycle:** We showed that reusing transformation results improves scalability for other consistency checking activities within the same validation cycle.
- **Evolutionary:** We showed that reuse improves evolutionary validation. We, however, found that evolutionary reuse results in extensive disadvantages in terms of change management. We, therefore, introduced the reduced redundancy model to mitigate that problem. No studies were performed whether our approach indeed improves evolution since our emphasis was not geared towards that.
- **Process Support:** we provide no process support
- **Property Support:** we provide no property support

Summarizing, our approach adds value over existing consistency checking approach in its improved and comprehensive handling of abstractions and refinement, and in its strong improvements in scalability.

11.1.8 Breadth over Depth

View integration of heterogeneous types of views is an elaborate subject. During the creation of this work, it became more obvious that the problem is too extensive to be completely addressed by us as part of this thesis. This caused a dilemma since it was our goal to provide a view integration framework that could scale beyond one or two types of diagrams. In fact, this is another dimension in which our work is distinguishable from other view integration solutions. Our framework was meant to cover a more

comprehensive set of views: from abstract to concrete; from generic to specific; and from structural to behavioral. To address this problem we chose a breadth and depth approach to view integration.

Breadth to consider a wide range of views

- Transformation is improved if multiple views are evaluated (recall sequence to statechart transformation)
- View integration has more complexity if a wide range of models has to be supported
- Integration must cover extensive types of diagrams and views
- Some view integration concerns cannot be revealed unless multiple views are investigated

Depth to consider complexity aspects

- Important view integration concerns cannot be revealed if views are not evaluated in detail
- Scalability and complexity concerns only become obvious by a rigorous treatment of the entire consistency checking process
- Full automatability can only be claimed if entire “analysis lifecycle” from mapping to transformation to comparison is supported

Our solution to this dilemma was to explore both options. We selected a set of diagrams UML provided that sufficiently covered all model dimensions (recall Section 5). The superficial treatment of all those dimensions allowed us to build a framework that is comprehensible in the types of views it supports. For instance, the integral part of transformation as part of our view integration framework is a direct derivative of having had to evaluate a wide range of development models. Only by doing that, we realized how difficult it is to directly map and compare two different types of diagrams. Furthermore, the broad view coverage made us aware of the complexity of view integration in terms of how many transformation methods had to be supplied. Our solutions of using intermediate models and complex transformation are a direct result of addressing this broad problem.

In order to also understand the intricacies of step-by-step consistency checking in depth, we chose one type of transformation—abstraction in our case—and explored it in all detail. The in-depth

treatment of abstraction allowed us to build a framework that is comprehensible. Our in-depth treatment of abstraction resulted in a tool support that covers the entire view analysis life cycle including many of the scalability measures we discussed. The depth approach also resulted in a number of discoveries that the breadth approach would not have found. For instance, the evolutionary scalability problem and its need for a reduced redundancy model is a direct lesson learnt. Further, the creation of consistency rules and how they are supported by the model was only possible by investigating the details of abstraction.

11.1.9 Technology and Research Transfer

Our tool, UML/Analyzer is currently being evaluated by two groups within Mitre organization. A subset of our tools functionality was also implemented in a tool called Rose/Architect (for Rational Software). Additionally, about a dozen other groups requested the tool and are evaluating it for different uses. We found that its ability to abstract class diagrams makes it also very useful for reverse engineering.

11.2 Future Work

Our work provided a framework for automated and scalable consistency checking. In the course of discussing the details of our framework we revealed a series of cases where more work has to be done. We only investigated two types of UML views in depth. Future work requires the continuing validation of our framework towards other types of UML views as well as outside views (e.g., ADLs). Also we would like to apply our framework to larger, industrial projects.

Although automation can save considerable time and effort and improve the overall quality of model and product, full automation is certainly unrealistic today and will likely remain so for some time to come. This implies that synthesis and analysis will continue to incorporate a sizeable human commitment. Feasible and practical view integration must therefore adhere to special considerations and ergonomic constraints posed by human users (e.g., architect, designer, programmer). This problem requires a stronger attention towards formalisms, model design (construction), distribution, and human interactions than we have given to date. The following activities would further improve our integration approach:

- **Improving Reliability:** The reliability of our approach can be improved by improving the reliability of transformation and consistency checking. This task involves the refinement and validation of transformation and consistency rules.
- **Inconsistency Resolution:** Currently our approach is limited to inconsistency detection. Although we do not believe in automated inconsistency resolution, we do believe that assistance can be given by suggesting options on how to resolve them. Resolution involves a stronger emphasis on human computer interactions.
- **Smart Transformations:** The challenge of model evolution is that re-generation frequently overwrites instead of adapts. Take, for instance, code generators that can produce skeleton code out of designs. Frequently those generators assume waterfall-like situations where changes to the code may get lost after the re-generation of the design. Smart transformation allows the continuous evolution of several models with an intelligent way of updating them.
- **Product Families:** It has been recognized that product lines exhibit strong potential for reuse—both of program code and modeling data. What has been greatly neglected is that product lines also enable a stronger potential for automated refinement and consistency checking (plus some limited forms of inconsistency resolution) since the similarities in products enable a more meaningful comparison.
- **Distributed Modeling:** Distributed systems are important for modeling in two fashions: (1) to model distributed products; and (2) to support distributed modeling. Whereas the former has received strong attention in the research community, the latter has lagged behind. Modeling needs to become distributed to make use of new and powerful interaction technologies (such as the web) as well as to handle an increasingly distributed workforce. Consistency checking among distributed models is one example of the challenges of distributed modeling.
- **Model Connectors:** The problem of component connectors has received strong attention by the research community for some time now. Seeing modeling integration in analogy to architectural description languages (ADLs) may however shed new light onto this problem. ADLs talk in terms of components and connectors. If models (views) are the components of model-based development,

then how can ADL knowledge about connectors help in finding the bridges between their model components? This facet is an alternative way of seeing view integration.

- Component-based modeling involves the use of COTS (commercial-off-the-shelf) components in software products. Our research so far has indicated that component-based and model-based development does complement one another well in enabling refinement and inconsistency detection.

11.3 Conclusion

Model-based software development handles complexities by allowing development concerns to be addresses, solved, and interpreted on an individual basis (separation of concerns). Model-based software development is thus essential in developing large-scale, complex, and labor-intensive software systems. The widespread acceptance of modeling languages such as the Unified Modeling Language (UML) attest to that. Models, despite their invaluable strengths, exhibit one major weakness. To enable a separation of concerns, models form their individual closed-world environments. These closed-world environments hinder the communication and interaction between individual views. Interaction, however, must happen to enable information exchange and to ensure consistency.

This work presented a view integration framework with support for automated synthesis and analysis. Our synthesis techniques assist the replication of information between views. Automated synthesis reduces the manual, error-prone, and repetitive activities that occur during the exchange of modeling information. Our analysis techniques validate consistency between replicated information present in multiple views. Automated analysis reduces the manual, error-prone and repetitive activities that accompany consistency checking.

The key to scalable and less complex consistency checking is in automated transformation coupled with the reuse of transformation results. The observation that transformation can simplify consistency checking was already made by other researchers (e.g., [Koskimies et al. 1998]), however, none of them addressed the scalability problem related to handling and maintaining derived information. Our work, therefore, introduced a series of scalability improvements (reduced number of transformation methods and reduced redundancy model).

Currently, we have automated our view integration framework in context of class, object, and C2 diagrams. We have also shown the benefits of using our integration techniques in saving considerable human effort and in its ability to locate types of inconsistencies that no other inconsistency approach can locate (automatically). Also our approach enables inconsistencies to be identified as early on as they are created. Every time new data is added to the model, our approach and tool can be used to validate them.

12 References

1. Abd-Allah, A.: "Composing Heterogeneous Software Architectures," PhD Dissertation, University of Southern California, Los Angeles, CA 0089-0781, USA, 1996.
2. Abi-Antoun, M. and Medvidovic, N.: "Enabling the Refinement of a Software Architecture into a Design," *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, Fort Collins, CO, October 1999.
3. Allen, R., Garlan, D.: "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, July 1997.
4. Alvarado, S.: "An Evaluation of Object Oriented Architecture Models for Satellite Ground Systems," *Proceedings of the 2nd Ground Systems Architecture Workshop (GSAW)*, El Segundo, CA, February 1998.
5. AT&T: "Best Current Practices: Software Architecture Validation," AT&T, Murray Hill, NJ, 1993.
6. Balzer, R.: "Tolerating Inconsistency," *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, pp. 158-165, May 1991.
7. Belkhouche, B. and Lemus, C.: "Multiple View Analysis and Design," *Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development*, October 1996.
8. Boehm, B.W.: *Software Engineering Economics*, Prentice Hall, 1981.
9. Boehm, B. W.: "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, 21(5), pp. 61-72, May 1988.
10. Boehm, B. W.: "Verifying and Validating Software Requirements and Design Specifications," *Software Risk Management* (Boehm ed.), IEEE Computer Society Press, pp.205-218, 1989.
11. Boehm, B. W.: "Anchoring the Software Process," *IEEE Software*, pp. 73-82, July 1996.
12. Boehm, B., Egyed, A.: "Optimizing Software Product Integrity through Life-Cycle Process Integration," *Journal for Computer Standards and Interfaces*, 21(1), pp. 63-75, May 1999.
13. Boehm, B., Egyed, A., Kwan, J., Madachy, R.: "Using the WinWin Spiral Model: A Case Study," *IEEE Computer*, pp. 33-44, July 1998.
14. Boehm, B. W., Ross, R.: "Theory W Software Project Management: Principles and Examples," *IEEE Transactions on Software Engineering*, pp. 902-916, July 1989.
15. Booch, G.: *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 1994.
16. Booch, G.: *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, 1996.
17. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*, Addison Wesley, 1999.
18. Brooks, F.P.: *The Mythical Man-Month*, Addison Wesley, 1995.
19. Brooks, F. P. Jr.: "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, April 1987.

20. Carmel, E., Whitaker, R., George, J.: "PD and Joint Application Design: A Transatlantic Comparison," *Communications of the ACM*, pp. 40-48, June 1983.
21. Carmichael, A.: *Object Development Methods*, New York, SIGS Books, 1994.
22. Chen, P.: "The entity relationship model towards a unified view of data," *ACM Transactions on Database Systems*, pp. 9-36, 1976.
23. Cheng, B. H. C., Wang, E. Y., Bourdeau, R. H., and Richter, H. A.: "Bridging the Gap Between Informal and Formal Approaches to Software Development," *Proceedings of Software Engineering Research Forum*, November 1995.
24. Coad, P., Yourdon, E.: *Object-Oriented Analysis*, Yourdon Press, 1991a.
25. Coad, P., Yourdon, E.: *Object-Oriented Design*, Yourdon Press, 1991b.
26. Conklin, J., Begeman, M.: "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," *ACM Transaction of Information Systems*, pp. 303-331, October 1988.
27. Cutts, G.: "Structured systems analysis and design methodology," *Information Technology for Organizational Systems*, Elsevier, pp.363-70, 1988.
28. Dardenne, A., Fickas, S., and Lamsweerde, A.: "Goal-Directed Concept Acquisition in Requirement Elicitation," *Proceedings of 6th International Workshop on Software Specification and Design (IWSSD 6)*, pp. 14-21, October 1993.
29. Delugach, H. S.: "An Approach to Conceptual Feedback in Multiple Viewed Software Requirements Modeling," *Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development*, October 1996.
30. DeMarco, T.: *Structured Analysis and System Specification*, New York, Yourdon Press, 1978.
31. Easterbrook, S., Nuseibeh, B.: "Using ViewPoints for Inconsistency Management," *IEE Software Engineering Journal*, November 1995.
32. Egyed, A.: "Using Model Transformation to Detect Inconsistencies between Heterogeneous Views," *submitted to the 8th Conference on Foundations of Software Engineering (FSE 8)*, 2000.
33. Egyed, A. and Gacek, C.: "Automatically Detecting Mismatches during Component-Based and Model-Based Development," *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pp. 191-198, October 1999.
34. Egyed, A. and Hilliard, R.: "Architectural Integration and Evolution in a Model World," *Proceedings of 4th International Software Architecture Workshop co-located with ICSE 2000*, Limerick, Ireland, June 2000.
35. Egyed, A. and Kruchten, P.: "Rose/Architect: a tool to visualize architecture," *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS)*, Maui, HI, January 1999.
36. Egyed, A. and Medvidovic, N.: "Extending Architectural Representation with View Integration," *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, Fort Collins, CO, October 1999.

37. Egyed, A. and Medvidovic, N.: "A Formal Approach to Heterogeneous Software Modeling," *Proceedings of 3rd Foundational Aspects of Software Engineering (FASE)*, Berlin, Germany, March 2000.
38. Egyed, A., Mehta, N., and Medvidovic, N.: "Software Connectors and Refinement in Family Architectures," *Proceedings of 3rd International Workshop on Development and Evolution of Software Architectures for Product Families (IWSAPF)*, Las Palmas de Gran Canaria, Spain, March 2000.
39. Ehrig, H., Heckel, R., Taentzer, G., Engels, G.: "A Combined Reference Model- and View-Based Approach to System Specification," *International Journal of Software Engineering and Knowledge Engineering*, 7(4), pp. 457-477, 1997.
40. Eliens, A.: *Object-Oriented Software Development*, Addison Wesley, 1995.
41. Fagan, M. E.: "Advances in software inspections," *IEEE Transactions on Software Engineering (TSE)*, 12(7), pp. 744-751, 1986.
42. Ferguson, J. e. al.: "Software Acquisition Capability Maturity Model," *Technical Report CMU/SEI-96/TR-020, ESC-TR-96-020*, Pittsburg, PA, 1996.
43. Finkelstein, A., Gabbay, D., Hunnter, A., Kramer, J., Nuseihbeh, B.: "Inconsistency Handling in Multi-Perspective Specifications," *Transactions on Software Engineering (TSE)*, 20(8), pp. 569-578, August 1994.
44. Finkelstein, A., Kramer, J., Nusibeh, B., Finkelstein, L., Goedicke, M.: "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development," *International Journal on Software Engineering and Knowledge Engineering*, pp. 31-58, March 1991.
45. Fowler, M.: *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.
46. Gacek, C., Abd-Allah, A., Clark, B. K., and Boehm, B.: "On the Definition of Software System Architecture," *Proceedings of the First International Workshop on Architectures for Software Systems*, pp. 85-95, Seattle, WA, 1995.
47. Gacek, C.: "Detecting Architectural Mismatches During System Composition," PhD Dissertation, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA, 1998.
48. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns Elements of Reuseable Object-Oriented Software*, Addison Wesley, 1994.
49. Garlan, D., Monroe, R., and Wile, D.: "ACME: An Architecture Description Interchange Language," *Proceedings of CASCON'97*, November 1997.
50. Gieszl, L. R.: "Traceability for Integration," *Proceedings of the 2nd Conference on Systems Integration (ICSI 92)*, pp. 220-228, 1992.
51. Gotel, O. C. Z. and Finkelstein, C. W.: "An analysis of the requirments traceability problem," *Proceedings of the First International Conference on Requirements Engineering*, pp. 94-101, 1994.
52. Grady, J.O.: *Systems Integration*, Boca Raton, FL, CRC Press, 1994.

53. Gruenbacher, P., Egyed, A., and Medvidovic, N.: "Separation of Concern in Requirements Negotiation and Architecture Modeling," *Proceedings of Workshop on Multi-dimensional Separation of Concerns in Software Engineering co-located with ICSE 2000*, Limerick, Ireland, June 2000.
54. Grundy, J., Hosking, J., Mugridge, W.: "Supporting flexible consistency management via discrete change description propagation," *Software Practice and Experience*, 26(9), pp. 1053-1083, 1996.
55. Grundy, J. C., Hosking, J. G.: "Constructing Integrated Software Development Environments with MViews," *Journal of Applied Software Technology*, 2(3/4), pp. 133-160, 1996.
56. Grundy, J. C., Hosking, J. G., Mugridge, W. B., and Amor, R. W.: "Support for Constructing Environments with Multiple Views," *Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development*, October 1996.
57. Grundy, J. C., Mugridge, W. B., and Hosking, J. G.: "Static and dynamic visualisation of software architectures for component-based systems," *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, pp. 426-433, June 1998.
58. Harel, D.: "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, 8, 1987.
59. Hilliard, R.: "Views and Viewpoints in Software Systems Architecture," *Position paper for the First Working IFIP Conference on Software Architecture*, February 1999.
60. Hofmeister, C., Nord, R. L., and Soni, D.: "Describing Software Architecture with UML," *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pp. 145-159, San Antonio, TX, February 1999.
61. Humphrey, W.S.: *A Discipline for Software Engineering*, Reading, MA, Addison-Wesley, 1995.
62. Hunter, A. and Nuseibeh, B.: "Analysing Inconsistent Specifications," *Proceedings of 3rd International Symposium on Requirements Engineering (RE97)*, January 1997.
63. Hunter, A., Nuseibeh, B.: "Managing Inconsistent Specifications: Reasoning, Analysis, and Action," *ACM Transactions on Software Engineering and Methodology*, 7(4), pp. 335-367, October 1998.
64. IEEE Architecture Working Group: "Recommended Practice for Architectural Description," *IEEE P1471/D5.2 Information Technology Draft*, December 1999.
65. Inverardi, P., Wolf, A. L.: "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model," *IEEE Transactions on Software Engineering*, April 1995.
66. Jackson, M.: "Some Complexities in Computer-Based Systems and Their Implications for System Development," *Proceedings of International Conference on Computer Systems and Software Engineering (CompEuro '90)*, pp. 344-351, Tel-Aviv, Israel, May 1990.
67. Jackson, M.A.: *System Development*, London, Prentice Hall, 1983.
68. Jackson, M.: *Software Requirements & Specifications*, Addison-Wesley, 1995.
69. Jacobson, I.M., Christerson, M., Jonsson, P., Overgaard, G.: *Object-Oriented Software Engineering : A Use Case Driven Approach*, Addison-Wesley, 1992.

70. Jacobson, I., Booch, G., Rumbaugh, J.: *The Unified Software Development Process*, Addison Wesley, 1999.
71. Khriiss, I., Elkoutbi, M., and Keller, R.: "Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams," *Proceedings for the Conference of the Unified Modeling Language*, June 1998.
72. Koskimies, K., Systä, T., Tuomi, J., Männistö, T.: "Automated Support for Modelling OO Software," *IEEE Software*, pp. 87-94, January 1998.
73. Kruchten, P.B.: *The Rational Unified Process*, Addison Wesley, 1998.
74. Kuhn, D. A.: *A Discription of the Systems Engineering Capability Maturity Model Appraisal Method Version 1.1*, Carnegie Mellon University, Pittsburgh, PA, 1996.
75. Luckham, D. C., J. Vera, J.: "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, September 1995.
76. Magee, J. and Kramer, J.: "Dynamic Structure in Software Architectures," *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, San Francisco, CA, October 1996.
77. Magee, J., Kramer, J.: *Concurrency: State Models and Java Programs*, Wiley, 1999.
78. Medvidovic, N., Rosenblum, D. S., and Taylor, R. N.: "A Language and Environment for Architecture-Based Software Development and Evolution," *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pp. 44-53, May 1999a.
79. Medvidovic, N., Egyed, A., and Rosenblum, D.: "Round-Trip Software Engineering Using UML: From Architecture to Design and Back," *Proceedings of the 2nd Workshop on Object-Oriented Reengineering (WOOR)*, pp. 1-8, Toulouse, France, September 1999b.
80. Medvidovic, N., Gruenbacher, P., Egyed, A., and Boehm, B.: "Software Lifecycle Connectors: Bridging Models across the Lifecycle," *submitted to 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, 2001.
81. Medvidovic, N. and Rosenblum, D. S.: "Assessing the Suitability of a Standard Design Method for Modeling Software Architectures," *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pp. 161-182, February 1999.
82. Medvidovic, N., Taylor, R. N.: "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, 26(1), pp. 70-93, January 2000.
83. Merriam-Webster: *Merriam-Webster's Collegiate Dictionary*, Merriam-Webster Incorporated, 1996.
84. Moriconi, M., Qian, X., Riemenschneider, R. A.: "Correct Architecture Refinement," *IEEE Transactions on Software Engineering*, April 1995.
85. Mullery, G.: "CORE: A Method for Controlled Requirements Specification," *Proceedings of 4th International Conference on Software Engineering (ICSE 4)*, pp. 126-135, September 1979.

86. Narayanaswamy, K. and Goldman, N.: "'Lazy' Consistency: A Basis for Cooperative Software Development," *Proceedings of International Conference on Computer-Supported Cooperative Work (CSCW'92)*, pp. 257-264, Toronto, Ontario, Canada, November 1992.
87. NASA: "Software Formal Inspection Process Standard," *NASA-STD-2202-93*, 1993.
88. Nuseibeh, B.: "Computer-Aided Inconsistency Management in Software Development," *Technical Report DoC 95/4, Department of Computing, Imperial College, London SW7 2BZ*, 1995.
89. Nuseibeh, B.: "Towards a framework for managing inconsistency between multiple views," *Proceedings of the Viewpoint 96: International Workshop on Multiple Perspectives in Software Development*, October 1996.
90. Nuseibeh, B., Kramer, J., Finkelstein, A.: "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification," *IEEE Transactions on Software Engineering*, pp. 760-773, October 1994.
91. Nuseibeh, B.: "A Multi-Perspective Framework for Method Integration," PhD Dissertation, Imperial College of Science, Technology and Medicine, London, England, October 1994.
92. Nuseibeh, B.: "To Be And Not To Be: On Managing Inconsistency in Software Development," *Proceedings of 8th International Workshop on Software Specification and Design (IWSSD-8)*, pp. 164-169, Schloss Velen, Germany, March 1996.
93. OMG: Unified Modeling Language Specification Version 1.3, OMG, 1999.
94. Orr, K.: Structured Requirements Definition, Topeka Kansas, Ken Orr and Associates, 1981.
95. Övergaard, G.: "A Formal Approach to Relationships in the Unified Modeling Language," *Proceedings of the Workshop on Precise Semantics for Software Modeling Techniques (PSMT'98)*, 1998.
96. Paulk, M.C., Weber, C. V., Curtis, B., Chrissis, M. B., Eds.: The Capability Maturity Model: Guidelines for Improving the Software Process, Reading, MA, Addison-Wesley, 1995.
97. Perry, D. E., Wolf, A. L.: "Foundations for the Study of Software Architectures," *ACM SIGSOFT Software Engineering Notes*, October 1992.
98. Potts, C. and Takahashi, K.: "An Active Hypertext for System Requirements," *Proceedings of the 7th International Workshop on Software Specification and Design (IWSSD 7)*, pp. 62-68, December 1993.
99. Racz, F. D. and Koskimies, K.: "Tool-Supported Compression of UML Class Diagrams," *Proceedings of the 2nd International Conference on the Unified Modeling Language (UML)*, October 1999.
100. Rechtin, E.: "System Architecting, Creating & Building Complex Systems," *Prentice Hall, Englewood Cliffs, NJ*, 1991.
101. Riemenschneider, R. A.: "Checking the Correctness of Architectural Transformation Steps via Proof-Carrying Architectures," *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pp. 65-81, February 1999.

102. Robbins, J. E., Medvidovic, N., Redmiles, D. F., and Rosenblum, D. S.: "Integrating Architecture Description Languages with a Standard Design Method," *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pp. 209-218, Kyoto, Japan, April 1998.
103. Robertson, S., Robertson, J.: *Mastering the Requirements Process*, Addison-Wesley, 1999.
104. Ross, D. T.: "Structured Analysis (SA): A language for communicating ideas," *IEEE Transactions on Software Engineering*, 3(1), pp. 16-34, 1977.
105. Royce, W. W.: "Managing the development of large software systems: Concepts and techniques," *Proceedings of 9th International Conference on Software Engineering (ICSE 9)*, 1970.
106. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F.: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
107. Rumbaugh, J., Jacobson, I., Booch, G.: *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.
108. Sage, A. P., Lynch, C. L.: "Systems Integration and Architecting: An Overview of Principles, Practices, and Perspectives," *Systems Engineering, The Journal of the International Council on Systems Engineering, Wiley Publishers*, 1(3), pp. 176-226, 1998.
109. Schoman, K., Ross, D. T.: "Structured analysis for requirements definition," *IEEE Transactions on Software Engineering*, 3(1), pp. 6-15, 1977.
110. Schönberger, S., Keller, R. K., and Khriiss, I.: "Algorithmic Support for Model Transformation in Object-Oriented Software Development," *Theory and Practice of Object Systems (TAPOS)*, 1999.
111. Selic, B., Gullekson, G., Ward, P. T.: *Real-Time Object Oriented Modeling*, New York, John Wiley and Sons, 1994.
112. Selic, B. and Rumbaugh, J. "Using UML for Modeling Complex Real-Time Systems," <http://www.objecttime.com/otl/technical/umlrt.pdf>, March 1998.
113. Shaw, M., Garlan, D.: *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
114. Sheard, S. A. and Lake, J. G.: "Systems Engineering Standards and Models Compared," *Proceedings of the 8th International Symposium on Systems Engineering (INCOSE)*, pp. 589-605, Vancouver, Canada, 1998.
115. Shlaer, S., Mellor, S. J.: *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1989.
116. Shlaer, S., Mellor, S. J.: *Object Lifecycles: Modeling the World in States*, Yourdon Press, 1991.
117. Siegfried, S.: *Understanding Object-Oriented Software Engineering*, IEEE Press, 1996.
118. Sommerville, I.: *Software Engineering*, Addison-Wesley, 1996.
119. Sommerville, I., Sawyer, P.: *Requirements Engineering: A Good Practice Guide*, John Wiley & Sons, 1997.
120. Song, X., Osterweil, L. J.: "Toward Objective, Systematic Design-Method Comparisons," *IEEE Software*, 9(3), pp. 43-53, 1992.

121. Tarr, P., Osher, H., Harrison, W., and Sutton, S. M. Jr.: "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proceedings of the 21st International Conference on Software Engineering (ICSE 21)*, pp. 107-119, Los Angeles, CA, May 1999.
122. Taylor, R. N., Medvidovic, N., Anderson, K. N., Whitehead, E. J. Jr., Robbins, J. E., Nies, K. A., Oreizy, P., Dubrow, D. L.: "A Component- and Message-Based Architectural Style for GUI Software," *IEEE Transactions on Software Engineering*, 22(6), pp. 390-406, 1996.
123. Wang, E. Y. and Cheng, B. H. C.: "A Rigorous Object-Oriented Design Process," *Proceedings of the International Conference on Software Processes (ICSP5)*, June 1998.
124. Wang, E. Y., Richter, H. A., and Cheng, B. H. C.: "Formalizing and Integrating the Dynamic Model within OMT," *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, May 1997.
125. Warmer, J., Kleppe, A.: *The Object Constraint Language*, Reading, MA, Addison Wesley, 1999.
126. Warnier, J.D.: *Logical Construction of Programs*, New York, Van Nostrand, 1977.
127. Weaver, P.: *Practical SSADM*, London, Pitman, 1993.
128. Wile, D.: "AML: An Architecture Meta-Language," *Proceedings of the 14th International Conference on Automated Software Engineering (ASE'99)*, Cocoa Beach, FL, October 1999.
129. Wirfs-Brock, R., Wilkerson, B., Wiener, L.: *Designing Object-Oriented Software*, Prentice Hall, 1990.