
NON-FUNCTIONAL REQUIREMENTS IN SOFTWARE ENGINEERING

by

Lawrence Chung

*Department of Computer Science
The University of Texas at Dallas*

Brian A. Nixon

*Department of Computer Science
University of Toronto*

Eric Yu

*Faculty of Information Science
University of Toronto*

John Mylopoulos

*Department of Computer Science
University of Toronto*



SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

Non-functional requirements in software engineering / by Lawrence Chung...[et al.].

p. cm. -- (The Kluwer international series in software engineering)

Includes bibliographical references.

ISBN 978-1-4613-7403-9 ISBN 978-1-4615-5269-7 (eBook)

DOI 10.1007/978-1-4615-5269-7

1. Software engineering. 2. Computer software--Quality control. I. Series. II. Chung, Lawrence.

QA76.758 .N65 1999

005.1 21--dc21

99-046023

Copyright © 2000 by Springer Science+Business Media New York

Originally published by Kluwer Academic Publishers in 2000

Softcover reprint of the hardcover 1st edition 2000

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photo-copying, recording, or otherwise, without the prior written permission of the publisher, Springer Science+Business Media, LLC

Printed on acid-free paper.

THE KLUWER INTERNATIONAL SERIES IN SOFTWARE ENGINEERING

Series Editor

Victor R. Basili

*University of Maryland
College Park, MD 20742*

Also in the Series:

FORMAL SPECIFICATION TECHNIQUES FOR ENGINEERING MODULAR C PROGRAMS, by *TAN Yang Meng*; ISBN: 0-7923-9653-7

TOOLS AND ENVIRONMENTS FOR PARALLEL AND DISTRIBUTED SYSTEMS, by *Amr Zaky and Ted Lewis*; ISBN: 0-7923-9675-8

CONSTRAINT-BASED DESIGN RECOVERY FOR SOFTWARE REENGINEERING: Theory and Experiments, by *Steven G. Woods, Alexander E. Quilici and Qiang Yang*; ISBN: 0-7923-8067-3

SOFTWARE DEFECT MODELING, by *Kai-Yuan Cai*; ISBN: 0-7923-8259-5

The Kluwer International Series in Software Engineering addresses the following goals:

- To coherently and consistently present important research topics and their application(s).
- To present evolved concepts in one place as a coherent whole, updating early versions of the ideas and notations.
- To provide publications which will be used as the ultimate reference on the topic by experts in the area.

With the dynamic growth evident in this field and the need to communicate findings, this series provides a forum for information targeted toward Software Engineers.

NON-FUNCTIONAL REQUIREMENTS IN SOFTWARE ENGINEERING

Contents

List of Figures	ix
List of Tables	xvii
LEGEND FOR FIGURES	xix
Preface	xxix
1. INTRODUCTION	1
1.1 Introduction	1
1.2 The Nature of Non-Functional Requirements	6
1.3 Literature Notes	9
Part I The NFR Framework	
2. THE NFR FRAMEWORK IN ACTION	15
2.1 Using the NFR Framework	16
2.2 Acquiring Domain Knowledge	18
2.3 Acquiring and Cataloguing NFR Knowledge	18
2.4 Identifying NFRs	19
2.5 Decomposing NFR Softgoals	21
2.6 Dealing with Priorities	25
2.7 Identifying Possible Operationalizations	27
2.8 Dealing with Implicit Interdependencies among Softgoals	30
2.9 Recording Design Rationale	33
2.10 Selecting Among Alternatives	35
2.11 Evaluating the Impact of Decisions	37
2.12 Cataloguing Development Methods and Correlations	42
2.13 Discussion	44
3. SOFTGOAL INTERDEPENDENCY GRAPHS	47
3.1 Kinds of Softgoals	48
3.2 Interdependencies	54
3.3 The Evaluation Procedure	70

3.4	Coupling NFRs with Functional Requirements	80
3.5	Discussion	85
4.	CATALOGUING REFINEMENT METHODS AND CORRELATIONS	89
4.1	Refinement Methods	90
4.2	NFR Decomposition Methods	90
4.3	Operationalization Methods	111
4.4	Argumentation Methods and Templates	119
4.5	Correlations	129
4.6	Putting Them All Together: The Goal-Driven Process	137
4.7	Discussion	141
4.8	Related Literature for the Framework	142

Part II Types of Non-Functional Requirements

5.	TYPES OF NFRs	153
5.1	Categorizations of NFRs	155
5.2	Standards	158
5.3	A List of NFRs	159
5.4	Our Approach: The NFR Framework	159
5.5	Literature Notes	160
6.	ACCURACY REQUIREMENTS	161
6.1	Accuracy Concepts	163
6.2	Decomposition Methods	167
6.3	Operationalization Methods	175
6.4	Argumentation Methods	180
6.5	Correlations	181
6.6	Illustration	184
6.7	Discussion	194
7.	SECURITY REQUIREMENTS	197
7.1	Security Concepts	198
7.2	Decomposition Methods	201
7.3	Operationalization Methods	204
7.4	Argumentation Templates and Methods	207
7.5	Correlations	207
7.6	Illustration	208
7.7	Discussion	213
8.	PERFORMANCE REQUIREMENTS	217
8.1	Performance Concepts	218
8.2	Factors for Dealing with Performance Requirements	223
8.3	Refinement Methods	225

8.4	Operationalization Methods from Software Performance Engineering	233
8.5	Argumentation Methods and Templates	236
8.6	Correlations	238
8.7	Illustration	239
8.8	Discussion	247
9.	PERFORMANCE REQUIREMENTS FOR INFORMATION SYSTEMS	249
9.1	Language Features and Implementation Techniques for Information Systems	250
9.2	Example: A Research Management System	252
9.3	Extending the Performance Type	258
9.4	Organizing Issues via Language Layers	259
9.5	Decomposition Methods for Handling Data Management	264
9.6	Methods for Handling Inheritance Hierarchies	267
9.7	Methods for Handling Integrity Constraints and Long-Term Processes	273
9.8	Organizing Performance Methods	277
9.9	Organizing Correlations	280
9.10	Illustration	282
9.11	Discussion	283

Part III Case Studies and Applications

10.	INTRODUCTION TO THE STUDIES AND APPLICATIONS	291
10.1	Introduction	292
10.2	Characteristics of Domains Studied	293
10.3	Our Approach to Conducting the Studies	297
10.4	Observations from Studies	300
10.5	Literature Notes	300
11.	A CREDIT CARD SYSTEM	301
11.1	Domain Description and Functional Requirements	301
11.2	Non-Functional Requirements	304
11.3	Dealing with Performance Requirements	305
11.4	Dealing with Security and Accuracy Requirements	323
11.5	Discussion	328
11.6	Literature Notes	329
12.	AN ADMINISTRATIVE SYSTEM	331
12.1	Introduction	331
12.2	Domain Description, Functional Requirements and Organizational Workload	331
12.3	Non-Functional Requirements	333
12.4	Recording Domain Information in a Design	334
12.5	Overview of SIGs	334
12.6	Time Softgoals for Managing Long-Term Tax Appeal Processes	336
12.7	Operationalization Methods for Integrity Constraints	340

12.8 Dealing with a Tradeoff	346
12.9 Discussion	350
12.10 Literature Notes	350
13. APPLICATION TO SOFTWARE ARCHITECTURE	351
13.1 Introduction	352
13.2 Cataloguing Software Architecture Concepts using the NFR Framework	354
13.3 Illustration of the Architectural Design Process	358
13.4 Discussion	365
13.5 Literature Notes	366
14. ENTERPRISE MODELLING AND BUSINESS PROCESS REDESIGN	367
14.1 Introduction	367
14.2 The Strategic Dependency Model	370
14.3 The Strategic Rationale Model	374
14.4 Discussion	381
14.5 Literature Notes	382
15. ASSESSMENT OF STUDIES	383
15.1 Feedback from Domain Experts	383
15.2 Discussion: Lessons Learned for Conducting Studies	387
15.3 Literature Notes	389
POSTSCRIPT	391
BIBLIOGRAPHY	399

List of Figures

0.1	Logos for figures.	xx
0.2	Conventions for abbreviations and usage of fonts.	xxi
0.3	Legend for Softgoal Interdependency Graphs.	xxii
0.4	Legend for Softgoals and Interdependencies.	xxiii
0.5	Legend for Functional Requirements.	xxiv
0.6	Kinds of Refinements.	xxv
0.7	The “individual impact” of an offspring upon its parent for selected contribution types during the First Step of the evaluation procedure (Chapter 3). Parent labels are shown in the table entries.	xxvi
0.8	The “individual impact” of an offspring upon its parent during the First Step of the evaluation procedure (Chapter 3). An elaboration of Figure 0.7.	xxvi
0.9	Label propagation for selected contribution types and offspring labels during the First Step.	xxvii
0.10	Legend for Performance Requirements (Chapters 8 and 9).	xxviii
0.11	Legend for Business Process Redesign (Chapter 14).	xxviii
2.1	A catalogue of some NFR Types.	19
2.2	An initial Softgoal Interdependency Graph with NFR softgoals representing requirements for performance and security of customer accounts.	20
2.3	Decomposing NFR softgoals into more specific non-functional requirements.	22
2.4	Further decomposition of a security softgoal.	23
2.5	Considering a performance softgoal.	24
2.6	Decomposing a performance softgoal.	25
2.7	Identifying a softgoal as a priority.	26
2.8	Identifying possible operationalizations for NFR softgoals.	28
2.9	Detecting implicit interdependencies among existing softgoals.	31
2.10	Detecting implicit interdependencies among existing and other softgoals.	32

2.11	Recording design rationale.	34
2.12	Selecting among alternatives.	36
2.13	Evaluating the impact of decisions.	38
2.14	Relating decisions to Functional Requirements.	41
2.15	A catalogue of operationalization methods for achieving confidentiality.	43
2.16	A catalogue showing the impact of operationalizing softgoals upon NFR softgoals.	44
3.1	Representing sample non-functional requirements as NFR softgoals.	50
3.2	Portions of NFR type catalogues.	50
3.3	Sample operationalizing softgoals.	52
3.4	Representation of claims softgoals.	53
3.5	Decomposition and prioritization of NFR softgoals.	55
3.6	Kinds of Refinements.	56
3.7	Refining NFR softgoals into operationalizing softgoals.	57
3.8	Recording design rationale using claim softgoals.	59
3.9	A decomposition with an <i>AND</i> contribution.	61
3.10	Examples of several contribution types.	62
3.11	Intuitive distinction among contribution types.	63
3.12	Grouping positive and negative contribution types.	64
3.13	A conflict in contributions.	65
3.14	A softgoal interdependency graph with various contribution types.	69
3.15	Catalogue of label values.	72
3.16	Label propagation for selected contribution types and offspring labels during the First Step.	75
3.17	Examples of “automatic” label propagation during the Second Step.	76
3.18	Examples of developer-directed label propagation during the Second Step.	77
3.19	Meaning of symbols in softgoal interdependency graphs.	78
3.20	Recording developer’s decisions to choose or reject softgoals.	80
3.21	Determining the impact of developer’s decisions, using the evaluation procedure.	81
3.22	Relating functional requirements and the target system to a SIG.	82
3.23	Dealing with functional requirements guided by considerations of NFRs.	84
4.1	A catalogue of NFR decomposition methods.	91
4.2	A catalogue of NFR decomposition methods, including those for specific NFRs.	92
4.3	Definition and application of the <code>AccountResponseTimeViaSub-class</code> decomposition method.	93

4.4	Definition and application of ResponseTimeViaSubclass , a parameterized decomposition method.	95
4.5	Another application of the ResponseTimeViaSubclass method.	97
4.6	The StaffingAndAmountForOperatingCostViaSubclassAndDollar-Limit method, parameterized on two topics.	98
4.7	A catalogue of NFR Types.	99
4.8	Definition and application of the AccountSecurityViaSubType method.	100
4.9	The AccountQualityViaSubType method, parameterized on NFR type.	101
4.10	The SubType method, parameterized on NFR Type and Topic, and applied to performance of accounts.	102
4.11	The SubType method, applied to adaptability of insurance claims processing.	103
4.12	The SubType type decomposition methods, with varying degrees of parameterization.	104
4.13	The Subclass method, applied to space performance of accounts.	106
4.14	The Subclass method, applied to modifiability of packages.	107
4.15	The Subclass topic decomposition methods, with varying degrees of parameterization.	108
4.16	The Attribute decomposition method and one of its applications.	109
4.17	A Softgoal Interdependency Graph with two method applications.	110
4.18	A catalogue of operationalization methods.	111
4.19	The PerformFirst method, applied to an operation on an attribute.	112
4.20	The CompressedFormat operationalization method and an application.	113
4.21	The Auditing method and one of its applications.	114
4.22	The Validation method and one of its applications.	115
4.23	Refining an operationalizing softgoal and adding a topic.	116
4.24	Applying the AuthorizationViaSubType method to accounts, resulting in several offspring of an operationalizing softgoal.	117
4.25	A Softgoal Interdependency Graph with applications of various decomposition and operationalization methods.	118
4.26	A catalogue of argumentation methods.	119
4.27	A template for claims about validation.	121
4.28	Claims about operationalization methods.	122
4.29	A template for the VitalFew argumentation method for prioritization, and an example of its usage in a claim.	123
4.30	A template for claims about prioritization.	125
4.31	Another template for claims about prioritization.	126
4.32	Modifying the offspring labels of an AND interdependency.	127
4.33	Modifying the parent label of an AND interdependency.	128
4.34	A SIG with operationalization and argumentation methods.	129

4.35	Sample correlations.	131
4.36	Different kinds of inferences which can be made by using a correlation rule.	133
4.37	A correlation catalogue.	135
4.38	Benefits of using a common component in refinements of an operationalization.	136
4.39	Selection of operationalizing softgoals and argumentation softgoals.	138
4.40	The impact of decisions on NFRs and Functional Requirements.	139
5.1	Software Quality Characteristics Tree (From [Boehm76]).	157
5.2	Types of non-functional requirements (From [Sommerville92]).	158
6.1	An example of information flow.	164
6.2	An accuracy type catalogue.	165
6.3	A catalogue of accuracy decomposition methods.	168
6.4	A description of information flow.	170
6.5	A catalogue of accuracy operationalization methods.	176
6.6	The ValidationResourceAvailability method.	179
6.7	A catalogue of accuracy argumentation methods.	181
6.8	Decompositions for accurate travel expenses.	185
6.9	Operationalizing the accuracy of reimbursement requests.	188
6.10	Detecting a negative impact on a security requirement.	190
6.11	Evaluation of selective certification of expense summaries.	191
6.12	Relating functional requirements to the target system.	193
7.1	A catalogue of security types.	198
7.2	A catalogue of security operationalization methods.	205
7.3	Refinement of a security softgoal by subtypes.	209
7.4	SIG for confidential accounts.	210
8.1	The Performance Type.	220
8.2	Characteristics of the Performance Type.	220
8.3	Catalogue of Decomposition Methods for Performance.	225
8.4	A SubType decomposition.	226
8.5	Decomposing a performance softgoal using the Subclass method.	227
8.6	Using the IndividualAttributes method.	228
8.7	A decomposition based on implementation components.	228
8.8	Using the FlowThrough method to link layers.	230
8.9	A template for inter-layer refinement.	230
8.10	A prioritization argument.	231
8.11	Using the EarlyFixing method.	233
8.12	Refining an EarlyFixing operationalizing softgoal.	234
8.13	Positive and negative impacts of an operationalizing softgoal.	236
8.14	Catalogue of Operationalization Methods for Performance.	237
8.15	A correlation catalogue for Performance.	238
8.16	A decomposition on attributes.	240
8.17	A prioritization argument based on workload.	241
8.18	Consideration of an operationalizing softgoal.	242

8.19	An inter-layer interdependency link.	242
8.20	Initial operationalizing softgoals.	243
8.21	Refined operationalizing softgoals.	245
8.22	Evaluation of the Softgoal Interdependency Graph.	246
9.1	Definitions of the Employee and Researcher classes.	253
9.2	Defining ComputerResearcher as a specialization of Researcher .	254
9.3	Entity (data) classes in the functional requirements for the research administration example.	254
9.4	Transaction classes in the functional requirements.	255
9.5	A long-term research administration process represented as a script.	256
9.6	Adding ManagementTime to the performance type.	258
9.7	Adding ManagementTime and characteristics to the performance type.	259
9.8	Performance issues arranged in a grid.	260
9.9	Layered organization of performance knowledge.	261
9.10	Space of implementation alternatives arranged by layer.	263
9.11	Arrangement of attribute values in the presence of inheritance hierarchies.	268
9.12	Horizontal splitting of attributes.	268
9.13	Vertical splitting of attributes.	268
9.14	Attributes stored as “triples.”	269
9.15	Organization of Performance decomposition methods.	278
9.16	Some Performance decomposition methods for Layers 6 and 5.	279
9.17	Organization of Performance operationalization methods.	280
9.18	A performance correlation catalogue for information system development.	281
9.19	Dealing with inheritance hierarchies.	282
9.20	Linking another layer to the graph.	284
10.1	Overview of studies presented in Part III.	294
10.2	Overview of studies presented elsewhere.	295
11.1	Attributes of credit cardholders.	302
11.2	Classes in the credit card system.	302
11.3	Information Flow for the credit card system.	303
11.4	Some main NFRs for the credit card system.	304
11.5	Initial softgoal for fast cancellation of credit cards.	306
11.6	Softgoal decomposition and prioritization with argumentation.	307
11.7	Further softgoal decomposition and prioritization.	308
11.8	Selection of Operationalizing Softgoals.	309
11.9	Using inter-layer interdependency links.	310
11.10	Decomposition based on implementation components.	311
11.11	Evaluating the impact of decisions after selecting operationalizing softgoals.	312
11.12	Dealing with transaction hierarchies at Layer 4.	314
11.13	Dealing with priority operations at Layer 3.	315

11.14	Dealing with credit card attributes at Layer 2.	317
11.15	Evaluating the impact of decisions.	318
11.16	Considering inheritance hierarchies at Layer 4.	319
11.17	Selecting attribute storage methods at Layer 2.	321
11.18	Evaluating the impact of decisions.	322
11.19	SIG for storage of sales information.	323
11.20	Main Security Requirements for the credit card system.	324
11.21	Accuracy and Confidentiality Requirements for the credit card system.	325
11.22	Interactions among softgoals.	325
11.23	Evaluating the use of an operationalization.	327
11.24	Refining an Internal Confidentiality softgoal.	328
11.25	Evaluating the impact of alarms on Internal Confidentiality.	329
12.1	The tax appeals process represented as a Script.	335
12.2	Initial softgoal of good time performance for managing transitions of appeal process.	336
12.3	Decomposition into softgoals for individual transitions.	337
12.4	Further decompositions at Layer 6.	338
12.5	An inter-layer refinement.	339
12.6	Considering some operationalizing softgoals.	340
12.7	An Argument about an Operationalization.	341
12.8	Refining an Operationalizing Softgoal.	342
12.9	Evaluation of the SIG.	343
12.10	Time and Space softgoals and their refinements.	346
12.11	Identifying priorities using workload-based arguments.	347
12.12	Developing an hybrid method to deal with tradeoffs.	348
13.1	Catalogue of some NFR Types considered for software architecture.	355
13.2	A method for refining a modifiability softgoal for the system.	356
13.3	A method for refining a modifiability softgoal for the process.	356
13.4	A generic Correlation Catalogue, based on [Garlan93].	357
13.5	Initial NFR Softgoals for a KWIC system.	358
13.6	Refining softgoals using methods.	359
13.7	Tradeoffs among operationalizations.	361
13.8	Domain-Specific Correlation Catalogue for KWIC Example.	362
13.9	Evaluating the impact of the chosen alternative on NFRs.	364
14.1	Strategic Dependency Model for existing auto insurance claims handling.	371
14.2	Strategic Dependency Model for letting the insurance agent handle claims.	373
14.3	Strategic Dependency Model for letting the body shop handle claims.	374
14.4	Strategic Rationale Model relating softgoals and tasks for insurance claims handling.	375
14.5	Strategic Rationale Model for handling claims centrally.	377

LIST OF FIGURES xv

14.6	Strategic Rationale Model for handling small claims by the insurance agent.	378
14.7	Strategic Rationale Model for handling small claims by the body shop.	379
14.8	Overall Strategic Rationale Model for Claims Handling.	380

List of Tables

1.1	RADC software quality consumer-oriented attributes [Keller90].	2
3.1	The “individual impact” of an offspring upon its parent for selected contribution types during the First Step. Parent labels are shown in the table entries.	74
3.2	The “individual impact” of an offspring upon its parent during the First Step. Parent labels are shown in the table entries.	74
5.1	Software quality factors and criteria (From [Keller90]).	156
5.2	A list of non-functional requirements.	160
6.1	An accuracy correlation catalogue.	183

LEGEND FOR FIGURES

We present a collected legend for the figures of this book.

Figures in this book have a “logo” in the top left corner indicating the type of the figure. Logos for some figures types are given in Figure 0.1.

Some types of figures contain sub-figures. In Figure 0.1, their names are indented and preceded by a hyphen.

There are also *Informal* figures in Chapter 2, which use an approximate syntax.

Legend	
Logo	Explanation
<i>Claim Template</i> <i>Contribution Catalogue</i> <i>Contribution Type Examples</i> <i>Correlation Catalogue</i> <i>Evaluation Catalogue</i> <i>Evaluation Examples</i> <i>FRs</i> <i>Informal Correlation Catalogue</i> <i>Informal Legend</i> <i>Informal SIG</i> <i>Information Flow</i> <i>Label Catalogue</i> <i>Layered Structuring</i> <i>Legend</i> <i>Method Application — Initial SIG</i> — Resultant SIG <i>Method Catalogue</i> <i>Method Definition, Parameterized on Topic</i> <i>Method Definition, Parameterized on NFR Type</i> <i>Method Definition, Parameterized on NFR Type and Topic</i> <i>Method Definition, Unparameterized</i> <i>Method Hierarchy</i> <i>NFR Type Catalogue</i> <i>Refinement Catalogue</i> <i>Rule Definition, Parameterized on Topic</i> <i>Rule Application — Initial SIG</i> — Resultant SIG <i>SIG</i> <i>Softgoal Examples</i> <i>Strategic Dependency Model</i> <i>Strategic Rationale Model</i> <i>Template — Initial SIG</i> — Resultant SIG <i>Template Usage — Initial SIG</i> — Resultant SIG	Functional Requirements (Chapter 2) (Chapter 2) (Chapter 2) (Chapters 8 & 9)

Figure 0.1. Logos for figures.

<i>Legend</i>
<u>Abbreviations</u>
FR = Functional Requirements
IF = Information Flow
NFR = Non-Functional Requirements
PM = Policy Manual
SIG = Softgoal Interdependency Graph
SPE = Software Performance Engineering
<u>Usage of Fonts</u>
Helvetica Roman : <i>Elements of the NFR Framework, SIGs, catalogues and functional requirements</i>
<i>Helvetica Italic</i> : <i>Parameters</i>
<i>CAPITAL HELVETICA ITALICS</i> : <i>Contributions</i>
<i>Times Italic</i> : <i>Descriptions</i>
<i>Times Roman</i> : <i>Descriptions and some information flow</i>

Figure 0.2. Conventions for abbreviations and usage of fonts.

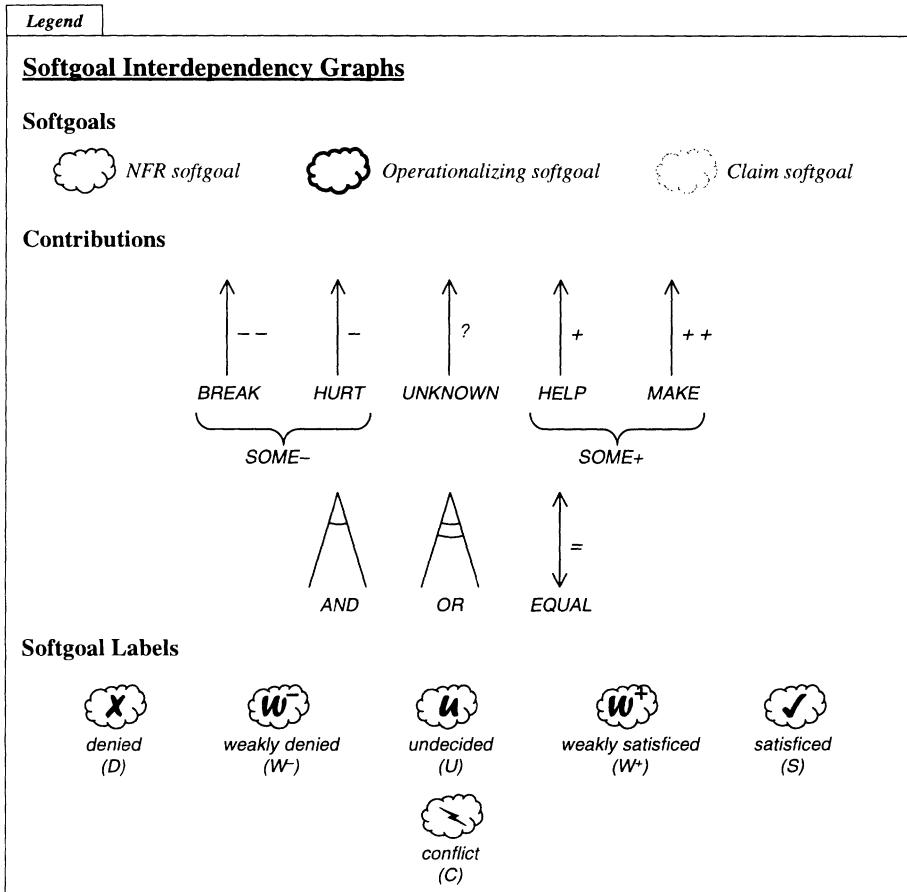


Figure 0.3. Legend for Softgoal Interdependency Graphs.

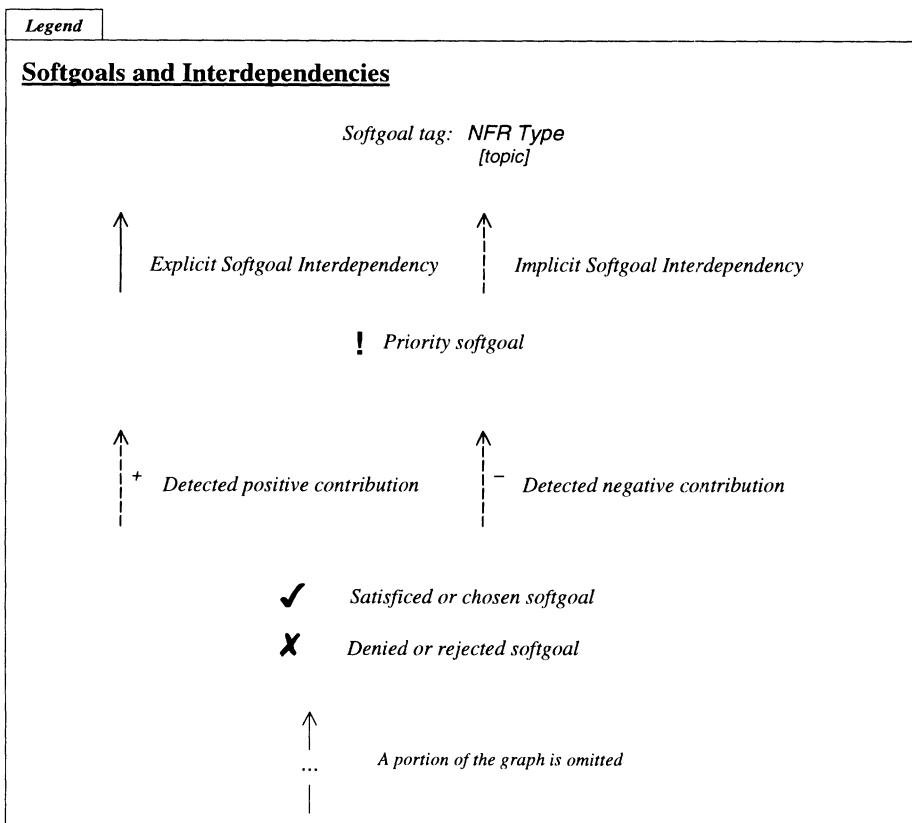


Figure 0.4. Legend for Softgoals and Interdependencies.

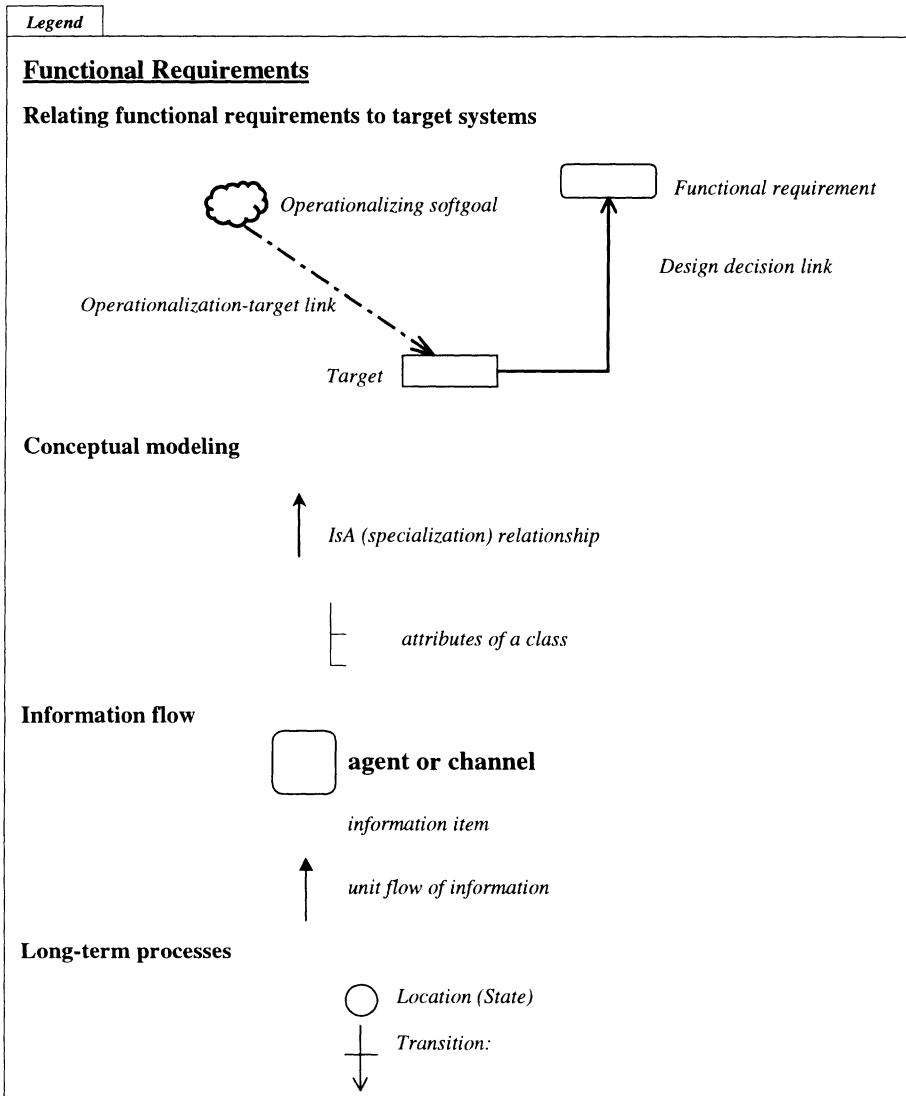


Figure 0.5. Legend for Functional Requirements.

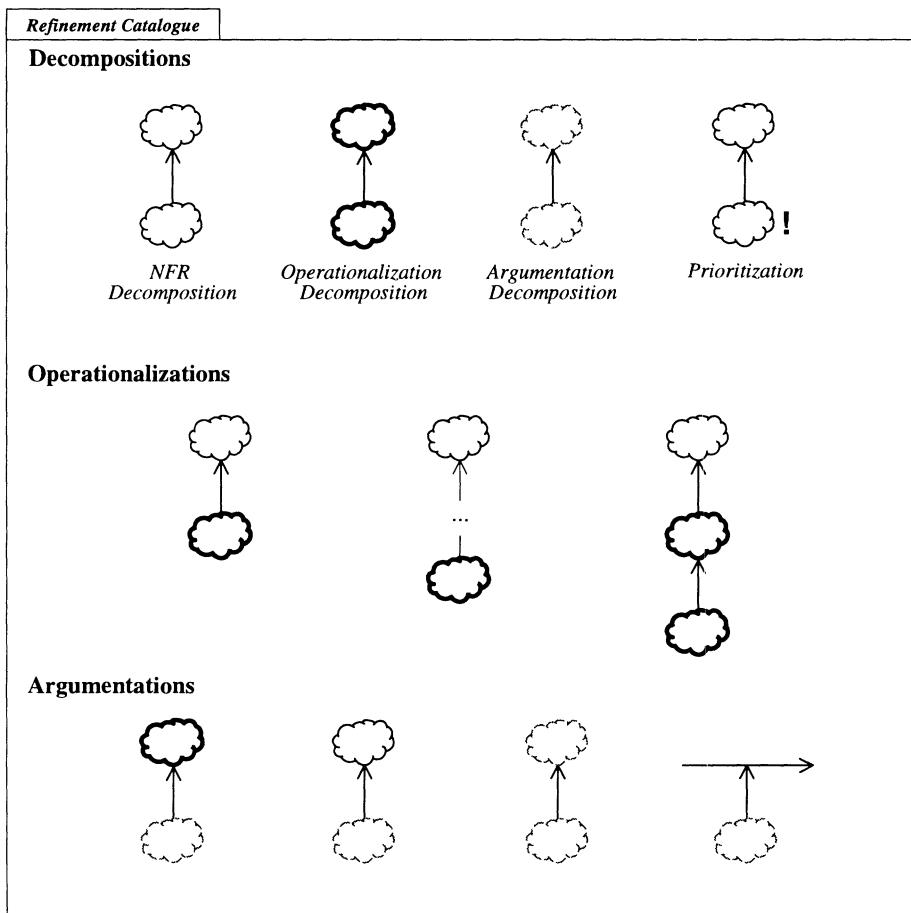


Figure 0.6. Kinds of Refinements.

<i>Evaluation Catalogue</i>				
<i>Individual Impact of offspring with label:</i>	<i>upon parent label, given offspring-parent contribution type:</i>			
	SOME-	?	(UNKNOWN)	SOME+ =
✗ (Denied)	W ⁺	U	W ⁻	✗
✗ (Conflict)	✗	U	✗	✗
U (Undetermined)	U	U	U	U
✓ (Satisfied)	W ⁻	U	W ⁺	✓

Figure 0.7. The “individual impact” of an offspring upon its parent for selected contribution types during the First Step of the evaluation procedure (Chapter 3). Parent labels are shown in the table entries.

<i>Evaluation Catalogue</i>								
<i>Individual Impact of offspring with label:</i>	<i>upon parent label, given offspring-parent contribution type:</i>							
	BREAK	SOME-	HURT	?	HELP	SOME+	MAKE	=
✗	W ⁺	W ⁺	W ⁺	U	W ⁻	W ⁻	✗	✗
✗	✗	✗	✗	U	✗	✗	✗	✗
U	U	U	U	U	U	U	U	U
✓	✗	W ⁻	W ⁻	U	W ⁺	W ⁺	✓	✓

Figure 0.8. The “individual impact” of an offspring upon its parent during the First Step of the evaluation procedure (Chapter 3). An elaboration of Figure 0.7.

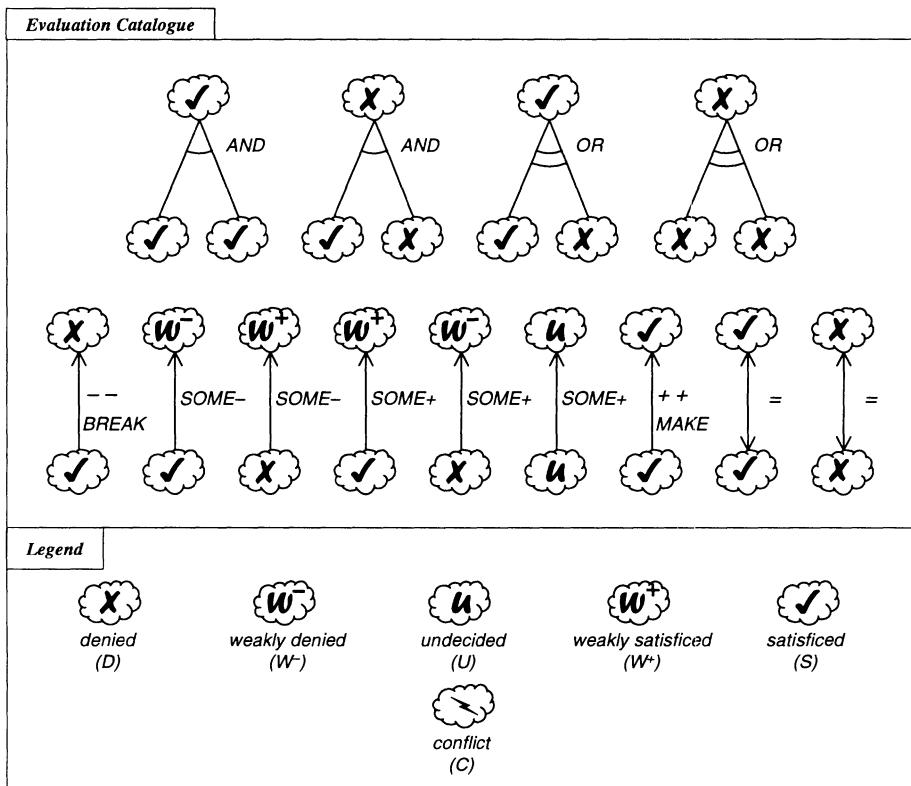


Figure 0.9. Label propagation for selected contribution types and offspring labels during the First Step.

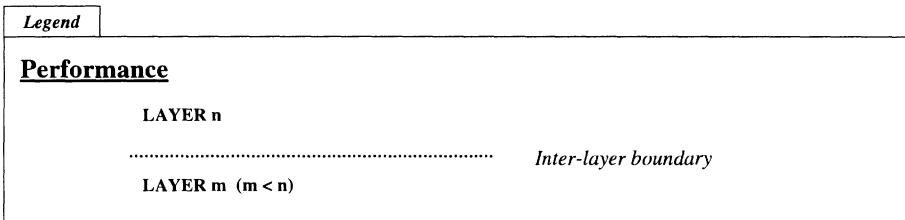


Figure 0.10. Legend for Performance Requirements (Chapters 8 and 9).

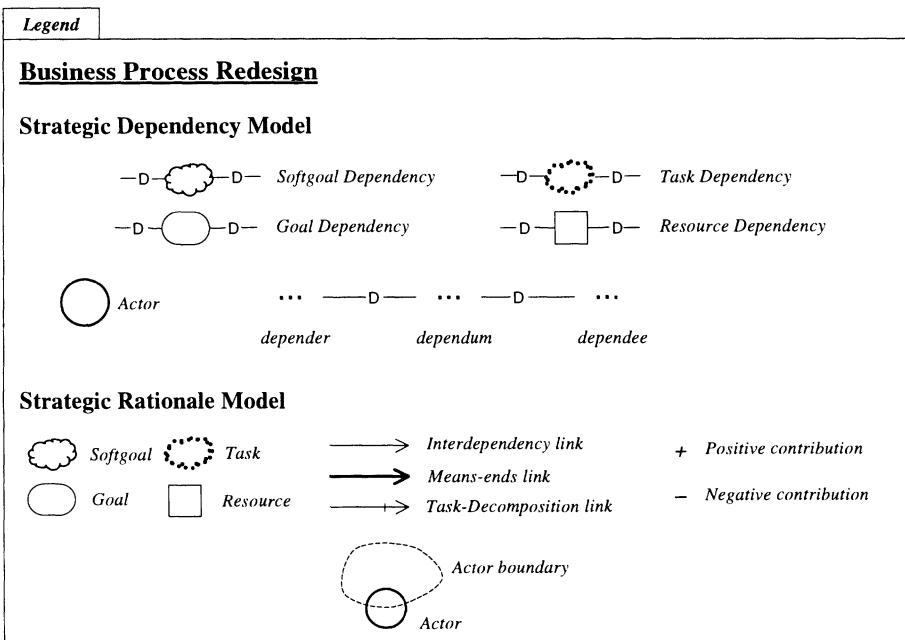


Figure 0.11. Legend for Business Process Redesign (Chapter 14).

Preface

The material in this book is based on the following publications, which are overviewed in Chapter 1: [Chung93a] [Nixon97a] [Yu94b] [Mylopoulos92a] [Chung94a,b] [Chung91a, 93b] [Nixon91, 93, 94a] [Chung95b] [Chung95c,d] [Yu94c] [Mylopoulos97] [Chung91b] [Nixon90]. The “Literature Notes” and “Discussions” at the end of each chapter describe the source publications for the particular chapter.

In writing this book, we have benefitted from the work of many people.

The following diagrams are taken from, or in some cases adapted from, the source publications listed below.

<i>Table or Figure in this Book</i>	<i>Source Publication</i>
Table 1.1	[Keller90]
Figure 5.1	[Boehm76]
Figure 5.1	[Keller90]
Figure 5.2	[Sommerville92]
Figure 13.4	[Garlan93]

The following trademarks have been used: SADT, a trademark of Softech Inc., and REFINE, a trademark of Reasoning Systems Inc.

We express our sincere gratitude to Scott Delman, Senior Publishing Editor of Kluwer Academic Publishers for first suggesting this book, and for his ongoing encouragement to complete it. We thank Scott for having the patience of Job during the writing and editing of this book. Scott, ably aided by Sharon Palleschi and Melissa Fearon (Editorial Assistants) and Suzanne St. Clair (Electronic Production Manager), provided help in many ways.

We thank Professor Victor R. Basili, Series Editor, The Kluwer International Series in Software Engineering, for reviewing the book proposal and arranging a review of a draft manuscript. We thank the anonymous reviewer for helpful comments.

Our deep appreciation for her contribution to our research goes to Elizabeth D'Angelo. Her excellent work in preparing the figures, proofreading the manuscript and suggesting helpful improvements has made the presentation more consistent and understandable. With the authors she designed the graphical notation used in this book. She has cheerfully dealt with numerous revisions. We also thank Ricardo D'Angelo for his help with the figures.

We thank our colleagues, Sol Greenspan, Alex Borgida, Matthias Jarke and Joachim Schmidt for sharing their expertise and ideas over the years. We have truly benefitted from collaborating with them.

We also thank Alex Borgida for coauthoring a source paper for Chapter 14.

The thesis committee members of the first three authors have also played an important role in the development of the material presented herein. We also appreciate the helpful comments received from the reviewers (often anonymous) of the source papers of this book. Many colleagues have read drafts of our earlier papers and offered helpful literature references, and we thank them, including Kostas Kontogianis, Ken Sevcik, Stewart Green, Vinay Chaudhri, Isabel Cruz, David Lauzon, Sheila McIlraith and Will Hyslop.

We thank colleagues who have provided us with references to identify issues and examples, organized workshops, and encouraged us to write up our work. They include Stephen Fickas, Anthony Finkelstein, Martin Feather, Peri Loucopoulos, Barry Boehm, John Callahan, David Garlan and Dewayne Perry.

A number of people with domain expertise aided our case studies, by providing domain documents, reviewing our studies, offering feedback, and referring us to others. We thank Lou Melli, Dominic Covvey, Bill Polimenakos, Brian Brett, David Braidwood, Honey Robb and Patrick Daly.

Many thanks to Belinda Lobo for excellent office support, as well as preparing the legend of logos for figures, and cheerfully maintaining our files.

We also thank Niloo Hodjati for carefully proofreading several chapters, Joseph Makuch for valuable systems support, and Daniel Gross for helpful discussions.

Many people have helped us develop tools for the NFR Framework, including Ian Maione, Thomas Rose, David Lauzon, Martin Staudt, Bryan Kramer and Martin Stanley. Although tools are not detailed in this book, the help of these people permitted us to try out some of the ideas of the Framework.

We acknowledge with gratitude the financial support received from the Institute for Robotics and Intelligent Systems (IRIS), the Consortium for Software Engineering Research (CSER) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Many colleagues and friends have encouraged us to complete this book.

Finally, we thank our families for their love and support throughout the development of this book.

1 INTRODUCTION

1.1 INTRODUCTION

The complexity of a software system is determined partly by its functionality — i.e., what the system does — and partly by global requirements on its development or operational costs, performance, reliability, maintainability, portability, robustness and the like. These *non-functional requirements* (or *NFRs*)¹ play a critical role during system development, serving as selection criteria for choosing among myriads of alternative designs and ultimate implementations. Errors of omission or commission in laying down and taking properly into account such requirements are generally acknowledged to be among the most expensive and difficult to correct once a software system has been implemented [Brooks87] [Davis93]. The cry from managers, software engineers, and users alike for software that is “better, cheaper, faster, friendlier” further illustrates the need to deal comprehensively with such requirements during the development process.

Surprisingly, non-functional requirements have received little attention in the literature and are definitely less well understood than other, less critical factors in software development. As far as software engineering practice is

¹The term “non-functional requirements” was used by Roman [Roman85]. NFRs are also referred to as *quality attributes* [Boehm78] [Keller90], *constraints* [Roman85], *goals* [Mostow85], *extra-functional requirements* [Shaw89], and *non-behavioural requirements* [Davis93] in the literature. We adopt the term *non-functional requirements* to distinguish them from functional ones.

2 NON-FUNCTIONAL REQUIREMENTS IN SOFTWARE ENGINEERING

concerned, they are generally stated informally during requirements analysis, are often contradictory, difficult to enforce during software development and to validate, when the software system is ready for delivery.

The only glimmer of technical light in an otherwise bleak landscape originates in technical work on software quality metrics that allow the quantification of the degree to which a software system meets non-functional requirements [Keller90, Boehm76, Boehm78, Basili91, Fenton97, Musa87, Lyu96].

Acquisition Concern	User Concern	Quality Attribute
Performance — How well does it function?	How well does it utilize a resource?	Efficiency
	How secure is it?	Integrity
	What confidence can be placed in what it does?	Reliability
	How well will it perform under adverse conditions?	Survivability
	How easy is it to use it?	Usability
Design — How valid is the design?	How well does it conform to the requirements?	Correctness
	How easy is it to repair?	Maintainability
	How easy is it to verify its performance?	Verifiability
Adaptation — How adaptable is it?	How easy is it to expand or upgrade its capability or performance?	Expandability
	How easy is it to change?	Flexibility
	How easy is it to interfere with another system?	Interoperability
	How easy is it to transport?	Portability
	How easy is it to convert for use in another application?	Reusability

Table 1.1. RADC software quality consumer-oriented attributes [Keller90].

Even though there is no formal definition, there has been considerable work on characterizing and classifying non-functional requirements. In a report published by the Rome Air Development Center (RADC, now known as the Rome Laboratory) [Bowen85], non-functional requirements (“software quality attributes” in their terminology) are classified into consumer-oriented (or *software quality factors*) and technically-oriented attributes (or *software quality criteria*). The former class of software attributes refers to software qualities observable by the consumer, such as efficiency, correctness and interoperabil-

ity. The latter class addresses system-oriented requirements such as anomaly management, completeness and functional scope. Table 1.1 shows the RADC consumer-oriented attributes. The non-functional requirements listed in the table apply to all software systems. However, additional requirements may apply for special classes of software. For instance, precision would be an important non-functional requirement for a numerical analysis software package, while accuracy (of maintained information) might feature prominently during the development of an information system.

Two basic approaches characterize the systematic treatment of non-functional requirements and we shall refer to them as *product-oriented* and *process-oriented*. The first attempts to develop a formal framework so that a software system can be evaluated as to the degree to which it meets its non-functional requirements. For example, measuring software visibility may include, among other things, measuring the amount of branching in a software system. This might be achieved globally with a criterion such as: “There shall be no more than X branches per 1 000 lines of code” or locally with a criterion such as “There shall be no more than Y% of system modules that violate the above criterion.” The product-oriented approach has received almost exclusive attention in the literature and is nicely overviewed in [Keller90].

Earlier work by Boehm et al. [Boehm78] considered quality characteristics of software, noting that merely increasing designer awareness would improve the quality of the final product. Also supporting a quantitative approach to software quality, Basili and Musa [Basili91] advocate models and metrics of the software engineering process from a management perspective. On a different track, Hauser et al. [Hauser88] provide a methodology for reflecting customer attributes in different phases of automobile design.

An alternative approach, explored in this book, is to develop techniques for justifying design decisions *during the software development process*. Instead of evaluating the final product, the emphasis here is on trying to rationalize the development process itself in terms of non-functional requirements. Design decisions may positively or negatively affect particular non-functional requirements. These positive and negative *interdependencies* can serve as basis for arguing that a software system indeed meets a certain non-functional requirement or explaining why it does not.

Orthogonally, treatments of non-functional requirements can be classified into *quantitative* and *qualitative* ones. Most of the product-oriented approaches alluded to earlier are quantitative in the sense that they study quantitative metrics for measuring the degree to which a software system satisfies a non-functional requirement. The process-oriented treatment proposed here, on the other hand, is definitely qualitative, adopting ideas from qualitative reasoning [AI84]. It should be acknowledged that a process-oriented treatment of non-functional requirements need not be qualitative. Indeed, one could imagine quantitative measures for, say, software visibility that can be used as the system is being developed to offer advance warning that non-functional requirements are not being met. Qualitative techniques were chosen here primarily because

it was felt that the problem of quantitatively measuring an incomplete software system is even harder than that of measuring the final product.

Of course, neither product-oriented quantitative metrics nor process-oriented qualitative ones have a monopoly on properly treating non-functional requirements. They are best seen as complementary, both contributing to a comprehensive framework for requirements engineering. This framework would use, among other things, process-oriented, qualitative methods during the early stages of requirements analysis, when the engineer is still trying to understand the problem and has vague ideas about how to constrain its solution. Product-oriented, quantitative methods, on the other hand, are most appropriate during software requirements definition, where ideas have crystallized into a coherent solution which is specified in terms of functionalities and measurable quality factors.

The objective of this book is to describe a novel framework, called the *NFR Framework*, for representing and analyzing non-functional requirements. In the classification scheme introduced earlier, the Framework is process-oriented and qualitative. A cornerstone of the Framework is the concept of *softgoal*, which represents a goal that has no clear-cut definition and/or criteria as to whether it is satisfied or not. The reader may have guessed already that softgoals are used to represent non-functional requirements. Softgoals are related through relationships which represent the influence or interdependency of one softgoal on another. A qualitative analysis method is included in the Framework for deciding the status of softgoals, given that other, related softgoals are satisfied or have been found to be unsatisfiable. In fact, throughout the book we shall speak of softgoals being *satisficed* [Simon81] rather than satisfied, to underscore the *ad hoc* nature of softgoals, both with respect to their definition and their satisfaction.

Two sources of ideas were particularly influential on our work. The first involves work on the capture and management of design rationale through decision support systems, such as those described in [J. Lee90, 91] and [Hahn91]. Lee's work, for example, adopts an earlier model for representing design rationale [Potts88] and extends it by making explicit the goals presupposed by arguments. The work reported here can be seen as an attempt to adopt and specialize this model to the representation and use of non-functional requirements. The second source of ideas is the DAIDA environment for information system development [Jarke92a, 93b] which has provided us with a comprehensive software development framework covering notations for requirements modelling, design, implementation and decision support, as well as a starting point on how the treatment of non-functional requirements might be integrated into that framework. Users of the DAIDA environment are offered three languages through which they can elaborate *requirements*, *design* and *implementation* specifications. In developing a design specification, the developer consults and is constrained by corresponding requirements specifications. Likewise, the generation of an implementation is guided by a corresponding design specification.

An early description of the Framework and an account of how it relates to DAIDA can be found in [Chung91a].

A further description of work related to the NFR Framework is found at the end of Chapter 4.

Design decisions relate implementation objects to their design counterparts, and design objects to their requirements counterparts. To account for design decisions, *softgoal interdependencies* are used. The Framework presented in this book focusses on these interdependencies and how they can be justified in terms of non-functional requirements.

The NFR Framework helps developers produce customized solutions, by considering characteristics of the particular domain and system being developed. These characteristics, including non-functional requirements, functional requirements, priorities and workload, influence the choice of development alternatives for a particular system. To deal with the large number of possible development alternatives, developers can consult the Framework's design catalogues. These catalogues organize past experience, standard techniques, and knowledge about particular NFRs, their interdependencies and tradeoffs.

The body of the book consists of three parts. The first part includes Chapters 2–4 and is intended to introduce the notions of *softgoal* and *softgoal interdependencies*, and the analysis that can be performed on *softgoal interdependency graphs* (SIGs). The second part of the book includes Chapters 5–9 and is intended to show how one can deal with particular types of non-functional requirements by producing catalogues which specialize the NFR Framework of Part I. In particular, Part II deals with accuracy and performance requirements for information systems, as well as security requirements for software systems. In the third part of the book we examine a number of case studies and applications, to demonstrate the usefulness of the Framework's approach to a variety of systems and domains. The domains studied include credit cards and government administration. We present some initial feedback from practitioners on the Framework and studies. The Framework is also applied to software architecture and business process redesign. This part of the book includes Chapters 10–15. Finally, the Postscript presents some concluding comments and directions for future work.

The material of this book is based largely on the Ph.D. thesis of Lawrence Chung [Chung93a], which develops a qualitative framework for representing and reasoning with non-functional requirements, and demonstrates its usefulness in dealing with particular classes of requirements. Another major source of material is the Ph.D. thesis of Brian Nixon [Nixon97a] which applies the NFR Framework to performance requirements for a particular class of software, information systems. Eric Yu's Ph.D. thesis [Yu94b] and subsequent work on business process redesign also have adopted concepts from the NFR Framework, most notably the notion of *softgoal*.

Other sources of material for the book include an earlier, more formal presentation of the NFR Framework [Mylopoulos92a], and tutorial introductions to the Framework [Chung94a,b]. This book also draws on papers which

have applied the Framework to deal with specific classes of NFRs: accuracy [Chung91a], security [Chung93b], and performance [Nixon91, 93, 94a]. Chung and Nixon [Chung95b] present a number of case studies related to information system development [Nixon93, 94a, 97a] [Chung93a,b]. Chung, Nixon and Yu [Chung95c,d] apply the NFR Framework to software architecture, while Yu, Mylopoulos and Borgida [Yu94c] [Mylopoulos97] discuss applications to business process redesign. Other earlier sources include work on information systems development, taking requirements to designs [Chung91b] and designs to implementations [Nixon90]. All material selected for inclusion in this book has been integrated, updated and simplified. In addition, the presentation, figures and terminology have been revamped.

Before presenting the details of the Framework in Part I, we will discuss the nature of NFRs.

1.2 THE NATURE OF NON-FUNCTIONAL REQUIREMENTS

non-functional requirement — in software system engineering, a software requirement that describes not what the software will do, but how the software will do it, for example, software performance requirements, software external interface requirements, software design constraints, and software quality attributes. Non-functional requirements are difficult to test; therefore, they are usually evaluated subjectively.

[Thayer90]

functional requirement — A system/software requirement that specifies a function that a system/software system or system/software component must be capable of performing. These are software requirements that define behaviour of the system, that is, the fundamental process or transformation that software and hardware components of the system perform on inputs to produce outputs.

[Thayer90]

Why are non-functional requirements so vital and yet so difficult to address?

Non-functional requirements (NFRs) address important issues of quality for software systems. NFRs are vital to the success of software systems. If NFRs are not addressed, the results can include: software which is inconsistent and of poor quality; users, clients and developers who are unsatisfied; and time and cost overruns to fix software which was not developed with NFRs in mind. Let's consider some aspects of NFRs.

Non-functional requirements can be *subjective*, since they can be viewed, interpreted and evaluated differently by different people. Since NFRs are often stated briefly and vaguely, this problem is compounded.

Non-functional requirements can also be *relative*, since the interpretation and importance of NFRs may vary depending on the particular system being considered. Achievement of NFRs can also be relative, since we may be able to

improve upon existing ways to achieve them. For these reasons, a “one solution fits all” approach may not be suitable.

Furthermore, non-functional requirements can often be *interacting*, in that attempts to achieve one NFR can hurt or help the achievement of other NFRs. As NFRs have a global impact on systems, localized solutions may not suffice.

For all these reasons, non-functional requirements can be difficult to deal with. Yet, dealing with NFRs can be vital for the success of a software system. Hence it is important to deal with them effectively. To better understand these aspects of NFRs, let us consider a simple scenario.

Suppose you want to develop a credit card information system. You may or may not be able to build it, depending on whether you have enough staff, funds, time, and other resources. Now suppose you want to build the system so that it is fast and accurate. You might be able to build a system, but will it be fast and accurate? Perhaps you consider it to be so, but your manager does not. Or perhaps you consider accuracy to be most important, but your manager really values speed. Now, without even getting your manager involved, would the system be fast on a day when there are a lot of credit card transactions? And would it be accurate even if merchants are being targeted by fraudsters? If not, perhaps you can make it more accurate, by doing more validation. But this extra processing may slow down the system.

The notion of “goal” has been used in a variety of settings. One prevalent use of this notion has its foundations in traditional problem-solving and planning frameworks in Artificial Intelligence. Used in that way, goals (e.g., “build a fast and accurate system”) are considered as obligations to be accomplished or satisfied. If goals can indeed be absolutely accomplished, they are said to be solvable or satisfiable; otherwise they are unsolvable or unsatisfiable. Put differently, a goal should be either satisfied or unsatisfied, and nothing else, in every possible imaginable situation (e.g., “The system is fast and accurate, day and night, during peak shopping periods, during crime waves, etc.”). With this two-valued logic, then, a goal evaluates to true, if it is satisfied, and false, otherwise. And absolutely nothing in-between.

In this context, a set of “sub-goals” is introduced to satisfy a given goal, where the relationship between the sub-goals and the parent goal is either *AND* or *OR*. When the relationship is *AND*, the goal is satisfied if all of its sub-goals are; when the relationship is *OR*, the goal is satisfied if any of its sub-goals is. This process of forming sub-goals continues until no goal, or sub-goal, can be further refined into more detailed ones. *But can this notion of “goal” always be equally-well applied to non-functional requirements?*

If the answer is yes, we should be able to say that non-functional requirements (e.g., “build a fast and accurate credit card system”) can *always* be “accomplished” or “satisfied” in a clear-cut sense. *But can we?*

Firstly, since NFRs can have a *subjective* nature, some solutions to non-functional requirements may be considered accomplished by some people, but not by others (e.g., “You feel the system is fast, but your manager doesn’t.”).

Not only that, solutions may be considered accomplished at one time, but not at another, even by the same person (e.g., “The system is fast for you overnight, but not during peak daytime shopping periods.”).

Secondly, since NFRs can have a *relative* nature, they may be poorly accomplished, marginally accomplished, well accomplished, etc. (e.g., “a system,” “an accurate system,” “a very accurate system,” etc.). In some cases, we may be able to find better solutions.

Furthermore, since some NFRs can be *interacting* and global in nature, some design decisions can have side-effects. In other words, design decisions which can contribute positively towards a particular non-functional requirement can also contribute negatively towards another, such as cost or convenience (e.g., “spending more processing time to improve accuracy.”). They can also contribute positively to other non-functional requirements (e.g., “an accurate system increases consumer confidence.”). Thus, non-functional requirements can be helpful (synergistic) or hurtful (antagonistic) to each other.

In this discussion, an interesting observation can be made about the way non-functional requirements are considered fulfilled or not fulfilled. It involves some kind of qualifications (e.g., “you want the system to be accurate *especially when there are fraud attempts*,” and “you want it to be fast *especially during peak periods*.”) Such qualifications may in turn be explained by other qualifications (e.g., “cardholders and merchants like consistent response time for sales authorization, *whether it is a peak shopping period or not*.”).

This kind of iterative qualification or reasoning, called a *dialectical style of reasoning*, allows one to state arguments, for or against other statements, in a recursive manner [Aristotle, 350 B.C.] [Toulmin58]. Reasoning with dualism (e.g., true and false) is helpful, and may suffice in some circumstances. However, in addition we may use a dialectical style of reasoning which handles subjectivity, relativity and interaction, reflecting the possible natures of non-functional requirements.

From the above discussion, we cannot always say that non-functional requirements are “accomplished” or “satisfied” in a clear-cut sense. Hence, we need something different than the notion of “goal” in traditional problem-solving and planning frameworks. *If so, what is an appropriate notion of “goal” when dealing with NFRs?*

In order to handle the possible natures of NFRs, we do not insist that all goals be met absolutely. (Of course, there may well be some absolute requirements, and they can be handled accordingly.) Rather, NFRs are treated as *softgoals*. A softgoal does not necessarily have *a priori*, clear-cut criteria of satisfaction. Although some of these can be measured and quantified, a qualitative approach can be used when exploring the possible development alternatives.

Now what is the appropriate notion of “relationship between softgoals”? Here, the strict AND/OR goal reduction of logical formalisms does not quite work. Instead, qualitative reasoning, a “lighter” (weak) form of reasoning, is used to reason about softgoals. We say that softgoals can *contribute* positively or negatively, and fully or partially, towards achieving other softgoals. This

approach is called “satisficing,” a term used by Herbert Simon in the 1950s, to refer to satisfying at some level a variety of needs, without necessarily optimizing results. The term has its origins in the word “satisfice,” which the *Oxford English Dictionary* reports was used as early as 1561. The reader may think of satisficing as *being sufficiently satisfactory*.

Expressing non-functional requirements as softgoals helps put concerns about NFRs foremost in the developer’s mind. Using the NFR Framework, then, the developer can “build quality into” systems, striving for NFRs systematically, rather than in an *ad hoc* manner.

The approach is *developer-directed*. Developers build quality into systems and produce customized solutions. This is done by using their expertise in the specific NFRs and in the domain under consideration, along with catalogues of expert knowledge about NFRs and development techniques. In this way, the developer uses NFRs to drive the overall development process and focus on the *process* of meeting NFRs. Along with their expertise and catalogues, developers’ creativity (their intellectual “spark”) is an important part of the development process.

1.3 LITERATURE NOTES

Parts of this chapter are based on [Mylopoulos92a].

The sources of each chapter, along with discussions of related work are presented in the “Literature Notes” and “Discussion” sections at the end of each chapter.

A discussion of related literature for the NFR Framework is presented in Section 4.8.

| The NFR Framework

2 THE NFR FRAMEWORK IN ACTION

Consider the design of an information system, such as one for managing credit card accounts. The system should debit and credit accounts, check credit limits, charge interest, issue monthly statements, and so forth.

During the development process of requirements elaboration, systems design and implementation, a developer needs to make decisions such as:

- How frequently will account information be updated?
- How will customer identity be validated — e.g., by using personal identification numbers (PIN codes) or biometrics?
- Will a certain group of data be stored locally or replicated over multiple sites?

These development decisions have important implications for the security, performance, accuracy, cost and other aspects of the eventual system. The significance of these *non-functional requirements* (or *software quality attributes*) are widely recognized. Attaining software quality attributes can be as crucial to the success of the system as providing the functionality of the system. For example, inaccurate credit account information can lead to monetary loss and damage to the reputation of a financial institution, while poor response time could lead to poor morale and eventually loss of customers.

Most conventional approaches to system design are driven by functional requirements. Developers focus their efforts primarily on achieving the desired

functionality of the system – calculating account interest, issuing monthly statements, etc. Although decisions about *how* to achieve functionality are made along the way, usually with non-functional requirements (such as cost and performance) in mind, these considerations may not be systematic, and may often not be documented. Furthermore, these software quality attributes may often be viewed as *consequences* of the decisions, but not something that the developer can strive for in a coherent, well thought-out way.

Note that in this section we are not using the terms “design” or “development” to refer to a particular phase of development. Rather, they are used in the broad sense of the process of developing a target artifact by starting with a source specification and producing *constraints* upon the target artifact. This could occur at various phases of development (e.g., requirements specification, conceptual design, or implementation).

2.1 USING THE NFR FRAMEWORK

In contrast to functionality-driven approaches, the NFR Framework uses non-functional requirements such as security, accuracy, performance and cost to drive the overall design process. The Framework aims to put non-functional requirements foremost in the developer’s mind.

There are several major steps in the design process:

- acquiring or accessing knowledge about:
 - the particular domain and the system which is being developed,
 - functional requirements for the particular system, and
 - particular kinds of NFRs, and associated development techniques,
- identifying particular NFRs for the domain,
- decomposing NFRs,
- identifying “operationalizations” (possible design alternatives for meeting NFRs in the target system),
- dealing with:
 - ambiguities,
 - tradeoffs and priorities, and
 - interdependencies among NFRs and operationalizations,
- selecting operationalizations,
- supporting decisions with design rationale, and
- evaluating the impact of decisions.

These are not necessarily sequential steps, and one may also need to iterate over them many times during the design process. A developer may choose

refinements, having operationalizations in mind; thus the development process may move up and down, rather than being strictly top-down.

It would be extremely helpful, if at each step in the process, the developer could draw on available knowledge that is relevant to that step in the process. This is precisely what the NFR Framework aims to provide. The Framework offers a structure for representing and recording the design and reasoning process in graphs, called *softgoal interdependency graphs (SIGs)*. The Framework also offers *cataloguing of knowledge* about NFRs and design knowledge, including development techniques. By providing SIGs and drawing on catalogues, the contextual information at each step can be used to trigger and bring forth previously-stored knowledge to help the developer carry out that step.

Softgoal Interdependency Graphs

The operation of the Framework can be visualized in terms of the incremental and interactive construction, elaboration, analysis, and revision of a *softgoal interdependency graph (SIG)*. The graph records the developer's consideration of *softgoals*, and shows the *interdependencies* among softgoals.

Major concepts of the Framework appear in the graphical form in SIGs. Softgoals, which are “soft” in nature, are shown as clouds. Main requirements are shown as softgoals at the top of a graph. Softgoals are connected by *interdependency links*, which are shown as lines, often with arrowheads. Softgoals have associated *labels* (values representing the degree to which a softgoal is achieved) which are used to support the reasoning process during design. Interdependencies show *refinements* of “parent” softgoals *downwards* into other, more specific, “offspring” softgoals. They also show the *contribution* (impact) of offspring softgoals *upwards* upon the meeting of other (parent) softgoals.

To determine whether softgoals are achieved, an *evaluation procedure (labelling algorithm)* is used, which considers labels and contributions, and, importantly, decisions by the developer.

It is important to note that the *developer* has control over what softgoals are stated, how they are refined, and the extent to which they are refined. The design process and evaluation procedure are *interactive*. Evaluation is also “semi-automatic,” i.e., assisted by a procedure (algorithm), but with the developer in control.

Cataloguing Design Knowledge

A very important aspect of the Framework is that developers are able to draw on an organized body of design knowledge (including development techniques) that has been accumulated from previous experience. This type of knowledge can be arranged in knowledge *catalogues*.

There are three kinds of catalogues used. One kind of catalogue represents knowledge about the particular types NFRs being considered, such as security and performance, and their associated concepts and terminology. Another kind of catalogue is used to systematically organize *development tech-*

niques (methods), which are intended to help meet requirements, and are available to developers. The third type of catalogue shows implicit interdependencies (*correlations, tradeoffs*) among softgoals.

The design knowledge in catalogues may come from many sources. General knowledge for various areas (e.g., performance, security, usability) are typically available in textbooks, developer guides and handbooks. More specialized knowledge may be accumulated by specialists in industry and academia, or within an organization. Individual developers and teams also build up knowledge from their experiences over a number of projects that can be reused. By making this knowledge available in a single design framework, developers can draw on a broad range of expertise, including those beyond their own immediate areas of speciality, and can adapt them to meet the needs of their particular situations.

These catalogues, such as those offered in Part II, provide a valuable resource for use and re-use during development of a variety of systems. This was our experience in the case studies of Part III, where catalogued knowledge of domain information and NFRs was used throughout the design process. Thus other steps in the process can be aided by gathering and cataloguing knowledge *early* in the process.

Now we consider the various steps of the process of using the Framework.

2.2 ACQUIRING DOMAIN KNOWLEDGE

During the process, the developer will acquire and use information about the domain and the system being developed. This includes items such as *functional requirements*, expected organizational *workload*, and organizational *priorities*.

For a credit card system, for example, the functional requirements include operations to authorize purchases, update accounts and produce statements. Organizational workload includes the number of cardholders and merchants, and the expected daily volume of purchases. The credit card organization will have some priorities, such as emphasizing the fast cancellation of stolen cards, and the provision of fast authorization.

Of course the developer will need to acquire a *source specification* of the system, before moving towards a *target*. For example, the source might be a set of requirements, and the target might be a conceptual design. As another example, the developer might start with a conceptual design and move towards an implementation of the system.

The case studies of Part III include descriptions of the domains studied.

2.3 ACQUIRING AND CATALOGUING NFR KNOWLEDGE

The developer will be drawing on catalogues of knowledge of NFRs and associated development techniques.

To provide a terminology and classification of NFR concepts, *NFR type catalogues* are used. Figure 2.1 shows a catalogue of NFRs. The NFR types are arranged in a hierarchy. More general NFRs are shown above more specific

ones. For example, performance has *sub-types* time and space, which in turn have their own sub-types. The NFRs shown in bold face are considered in detail in this book.

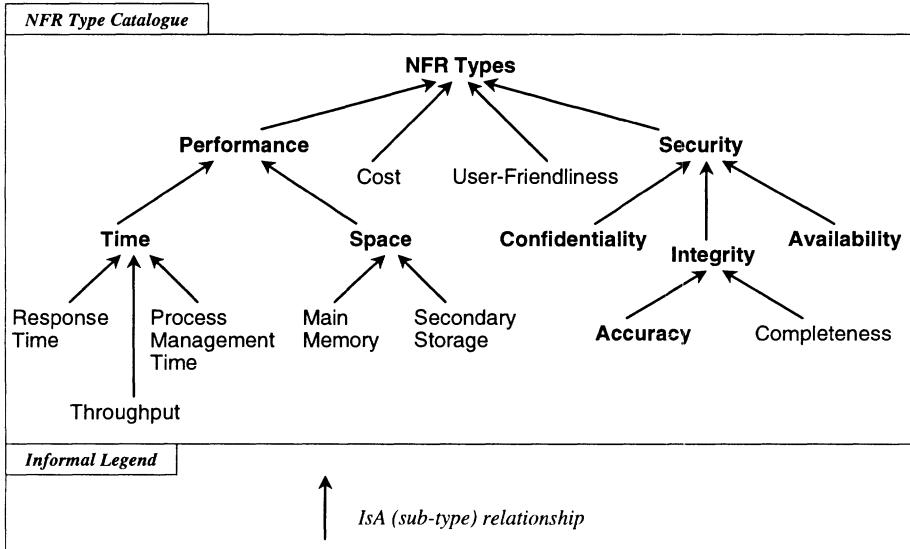


Figure 2.1. A catalogue of some NFR Types.

The NFR types provide a terminology to express requirements. For example, we can speak of the *security* of an account, or the *response time* for sales authorization.

Standard development techniques, along with methods of decomposing NFRs, are also organized into *method catalogues*. Interdependencies among NFRs (for example, stating that auditing helps security) are also organized into *correlation catalogues*. We will give examples of methods and correlations in this chapter, and will discuss these two types of catalogues in more detail in Section 2.12.

Normally, system developers can access existing catalogues from the start of development. The catalogues can be extended during development, to deal with additional or more refined concepts or development techniques which subsequently need to be addressed.

2.4 IDENTIFYING NFRs

In using the NFR Framework, one constructs an initial softgoal interdependency graph by identifying the main non-functional requirements that the particular

system under development should meet. In the credit card system example, these may include security of account information, and good performance in the storing and updating of that information. These non-functional requirements (NFRs) are then treated as *softgoals* to be achieved, i.e., they are goals which need to be clarified, disambiguated, prioritized, elaborated upon, etc. This particular kind of softgoal is called an *NFR softgoal*. As we will soon see, NFR softgoals are one of three kinds of softgoals.

The developer will identify specific possible development techniques. From among them, the developer will choose solutions in the target system that meet the source requirements specification. Thus the developer begins by systematically *decomposing* the initial NFR softgoals into more specific *subsoftgoals* (or *subgoals*).

Let us consider the requirements to “maintain customer accounts with good security” and “maintain customer accounts with good performance.”

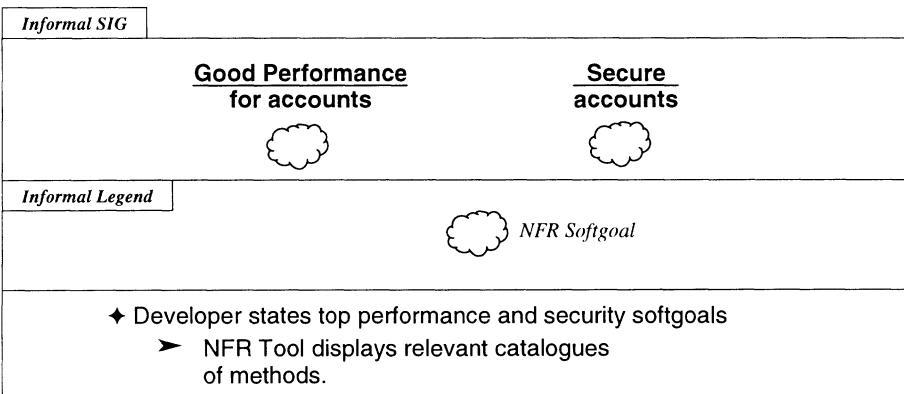


Figure 2.2. An initial Softgoal Interdependency Graph with NFR softgoals representing requirements for performance and security of customer accounts.

We are considering two non-functional requirements here, one for good performance for accounts, the other for good security of accounts. These non-functional requirements are represented as NFR softgoals *Good Performance for accounts* and *Secure accounts*. The NFR softgoals are represented by clouds, shown in the softgoal interdependency graph (SIG) of Figure 2.2.

Softgoals have an *NFR type*, which indicates the particular NFR, such as security or performance, addressed by the softgoal. In this chapter, the NFR types of some top-level softgoals are underlined in figures. Softgoals also have a subject matter or *topic*, here, accounts.

In this chapter, we use a syntax for SIGs which conveys the main points but is somewhat informal. Here the figure is an “informal” softgoal interdep-

dency graph (informal SIG). A more precise syntax is introduced in Chapters 3 and 4, and is used in the remainder of the book.

Figures in this book have a *logo* in the top left corner (such as *Informal SIG*) indicating the kind of figure. A list of the different kinds of logos for figures is given in Figure 0.1.

Some figures also have legends to describe new symbols. A collected *Legend for Figures* appears at the front of this book.

Figures in this chapter also have an informal description of the process of developing SIGs. A diamond introduces an action by the developer. A right-arrow introduces a response to the developer, done by consulting catalogues and executing algorithms (procedures).

Responding to a developer's decisions, along with the drawing of SIGs, can be provided by an interactive design support tool, or can be done "by hand" by the developer using "pencil and paper." The responses in the figures of this chapter are suggestive of how a tool, such as the "NFR Tool" [Chung93a, 94c], could be used. In this book, however, we do not assume that a particular method is used to draw graphs or respond to developers' decisions. Rather, the presentation below will focus on the usage of the NFR Framework's concepts. This is done to offer some evidence that the Framework is useful. However, we do not make a claim about the ease of handling large SIGs for large systems.

The following steps of using the NFR Framework can be viewed as an analysis of requirements, followed by a synthesis of operationalizations to meet the requirements. First, softgoals are broken down into smaller softgoals. We deal with ambiguities, and also consider domain information and priorities. Throughout the analysis we consider interdependencies among softgoals. Then we synthesize solutions to build quality into the system being developed. We consider possible alternatives for the target system, then choose some, and state reasons for decisions. Finally we see how well the main requirements have been met.

Interestingly, the NFR Framework is able to deal with different NFRs in one graph at the same time, even when the NFRs have different natures (here, performance and security). As we shall see, the NFR Framework can deal with interactions among these different NFRs.

2.5 DECOMPOSING NFR SOFTGOALS

In decomposing an NFR softgoal, the developer can choose to decompose its *NFR type* or its *topic*. In the example, the two softgoals share the same topic, accounts, but address different NFRs, performance and security.

Now we will decompose the two NFR softgoals of the example, starting with the security requirement.

The initial security requirement is quite broad and abstract. To effectively deal with such a broad requirement, the NFR softgoal may need to be broken down into smaller components, so that effective solutions can be found.

In addition, the requirement may be ambiguous. Different people may have different conceptions of what “security” constitutes in the context of credit card account information.

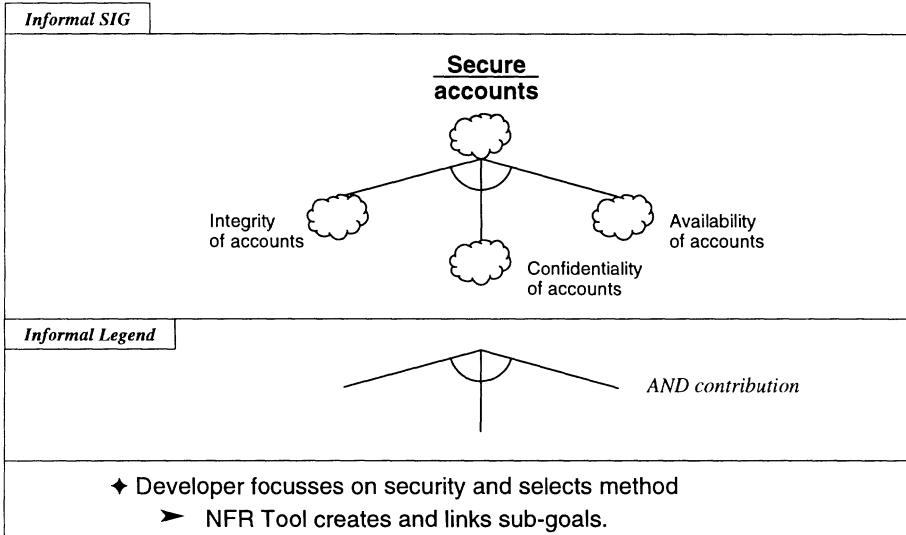


Figure 2.3. Decomposing NFR softgoals into more specific non-functional requirements.

By treating this high-level requirement as a *softgoal* to be achieved, we can decompose it into more specific subgoals which together “satisfice” (should meet) the higher-level softgoal. Thus the **Secure accounts** NFR softgoal can be decomposed into sub-softgoals for the

- integrity,
- confidentiality, and
- availability.

of the accounts. This is shown in Figure 2.3, which is an extension (downwards) of part of Figure 2.2. Such series of figures are used throughout this book to show the development of SIGs, where one figure builds on earlier ones.

In the graphical notation, clouds denote *softgoals* and lines represent *interdependencies* among softgoals.

Softgoals contribute, positively or negatively, to fulfilling other softgoals. There are different types of *contributions*. When *all* of several sub-softgoals are needed together to meet a higher softgoal, we say it is an *AND* type of contribution. Here, we say that *if* integrity, confidentiality *and* availability

are all met, *then* as a group their contribution will be to achieve the security softgoal. This is an *AND* contribution. It is shown with lines grouped by an arc.

Typically, softgoals are shown as being *refined* downwards into subsoftgoals (subgoals), and subgoals *contribute* upwards to parent softgoals.

It is interesting to note that steps used in constructing SIGs can draw on catalogues, such as the NFR Type catalogue of Figure 2.1. In Figure 2.3, for example, an entry (**Security**) in the type catalogue has specialized types (**Integrity**, **Confidentiality** and **Availability**). And in the SIG we see the same pattern, where inter-connected (interdependent) softgoals have these same types.

If patterns are expected to be re-used when building SIGs, *methods* can be defined and entered in a catalogue. Here, for example, a method can be defined which takes a parent softgoal of a particular type, and produces offspring softgoals which have sub-types of the parent's type. This **SubType** method is used in the SIG.

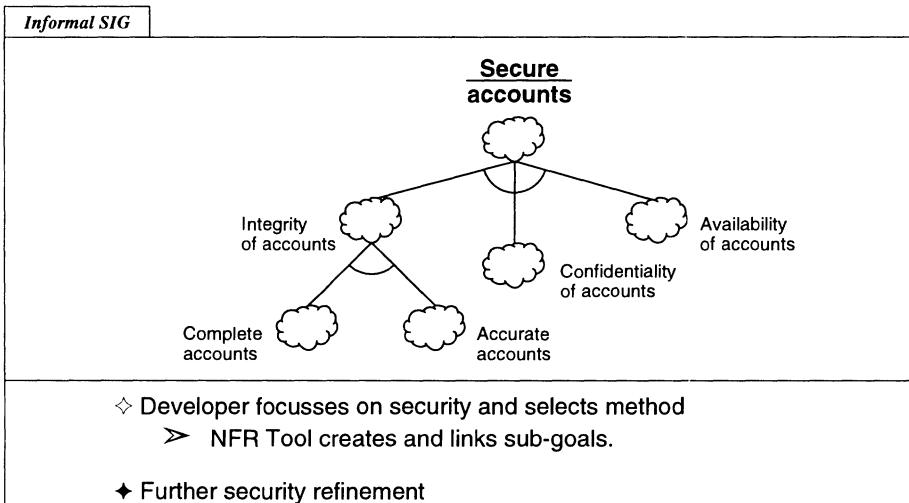


Figure 2.4. Further decomposition of a security softgoal.

The NFR softgoal that accounts will have integrity can be further decomposed into the subgoals that account information be complete, and accurate. This is shown in Figure 2.4. This is another application of the **SubType** method. Here, it considers the sub-types of **Integrity** from the catalogue of Figure 2.1.

In the descriptions at the bottom of figures in this chapter, solid diamonds and right-arrows represent new actions and responses, while outlined

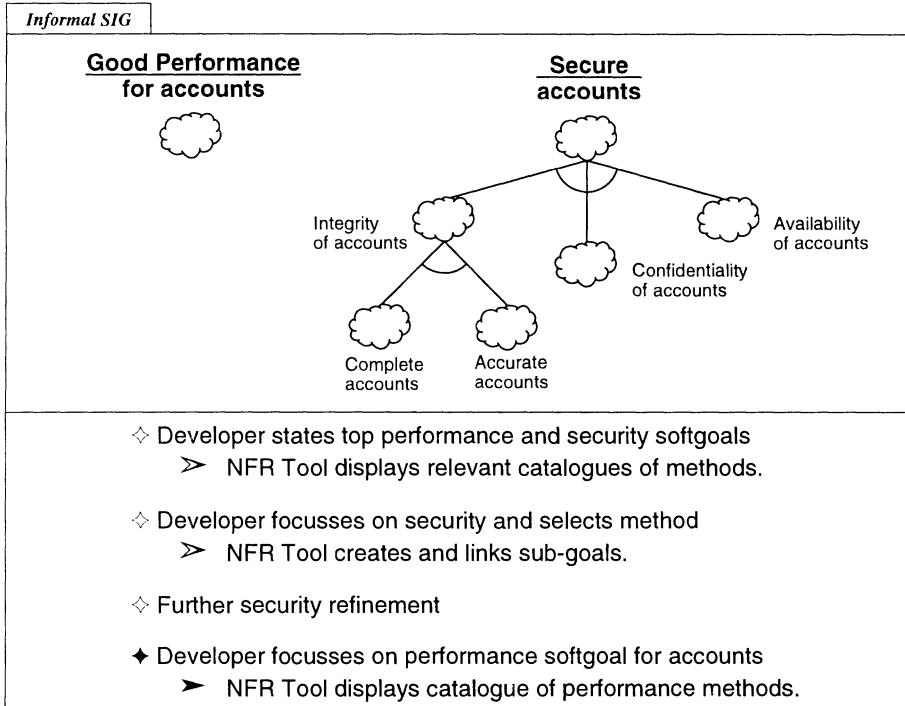


Figure 2.5. Considering a performance softgoal.

ones provide context by repeating actions and responses which were shown in earlier figures.

Recall that we started with two main NFRs, for security and performance of accounts. Figure 2.5 shows these initial NFRs from Figure 2.2, along with the security development of Figure 2.4.

Having refined the security requirement, the developer now focusses on the performance requirement.

The developer decides to decompose the performance softgoal with respect to its NFR type. This results, in Figure 2.6 (an extension of Figure 2.5), in two subgoals:

- one for good space performance for accounts, and
- one for good time performance for accounts.

Good time performance here means fast response time, and good space performance means using little space. Here the subgoals make an *AND* contribution to the performance softgoal.

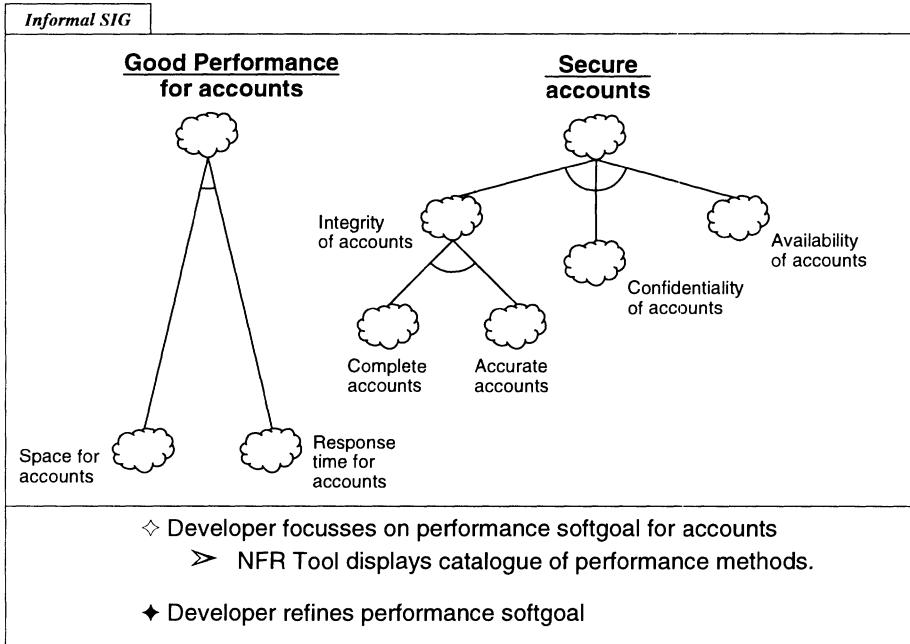


Figure 2.6. Decomposing a performance softgoal.

This is another use of the **SubType** method. It draws on the subtypes of **Performance** in the NFR type catalogue of Figure 2.1.

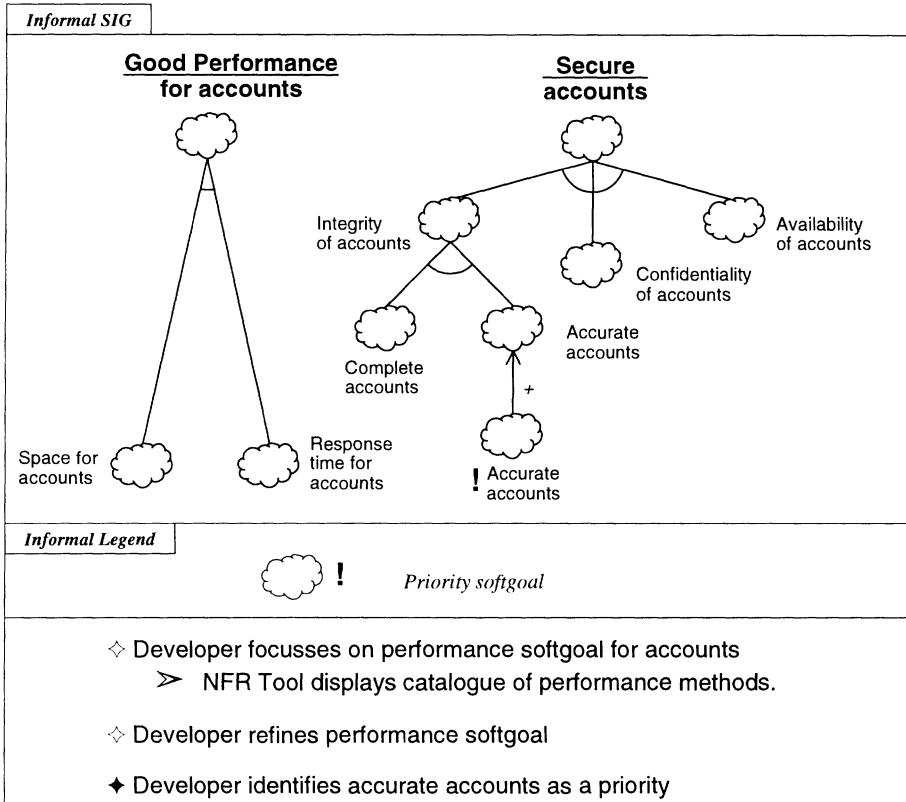
As well as their *NFR type*, softgoals can also be decomposed by their *topic*. For example, a performance softgoal for credit card accounts could be decomposed into performance softgoals for gold accounts and for regular accounts.

2.6 DEALING WITH PRIORITIES

Softgoal interdependency graphs can grow to be quite large and complex. How can a developer focus on what is important?

One way is to identify *priorities*. Extra attention can then be put towards meeting the priority softgoals.

Priorities can arise from consideration of several factors. These include *domain information* such as *organizational priorities* (e.g., a bank may consider security very important) and *organizational workload* (e.g., a credit card system may have a large number of sales to authorize each day). In addition, requirements can be identified as priorities during various phases of development.

**Figure 2.7.** Identifying a softgoal as a priority.

Some priority softgoals will be identified as *critical*, as they are vital to the success of the system or organization. Other softgoals will be noted as being *dominant*, as they deal with a significant portion of the organization's workload. Priority softgoals are identified by an exclamation mark (!).

Figure 2.7 identifies **Accurate accounts** as being a priority softgoal. This is shown by producing an offspring with the same type and topic, but noted as a priority by “!”.

The priority offspring contributes positively to the parent, and this is indicated by “+”. This positive contribution is an example of a *contribution type*, which shows the impact of offspring softgoals upon their parent softgoal.

The reasons for prioritizing a softgoal can be noted as design rationale, discussed in Section 2.9 below. Now that the priority is identified, it can be analyzed and dealt with.

For example, priorities may be used to make appropriate *tradeoffs* among softgoals. As an example, fast authorization of credit card sales may be given higher priority than fast determination of travel bonus points to be given to gold cardholders. Here, the developer may perform the authorization before the bonus calculation.

Knowledge of tradeoffs can be captured in catalogues, and made available for re-use in dealing with softgoal synergy and conflicts. Thus throughout the development process, various tradeoff considerations can be made.

2.7 IDENTIFYING POSSIBLE OPERATIONALIZATIONS

While the refinement process so far provides more specific interpretations of what the initial NFRs of “secure” and “good performance” mean, it does not yet provide means for actually accomplishing security and performance for accounts.

At some point, when the non-functional requirements have been sufficiently refined, one will be able to identify possible development techniques for achieving these NFRs (which are treated as NFR softgoals) and then choose specific solutions for the target system. The development techniques can be viewed as methods for arriving at the “target” or “destination” of the design process.

However, it is important to note that there is a “gap” between NFR softgoals and development techniques. To get to the destination, one must bridge the gap. This involves performing analysis, and dealing with a number of factors. These include ambiguities, priorities, tradeoffs, and other domain information such as the workload of the organization. These factors will have to be addressed at various steps in the process.

We show how development techniques are identified here, and how they are selected in Section 2.10.

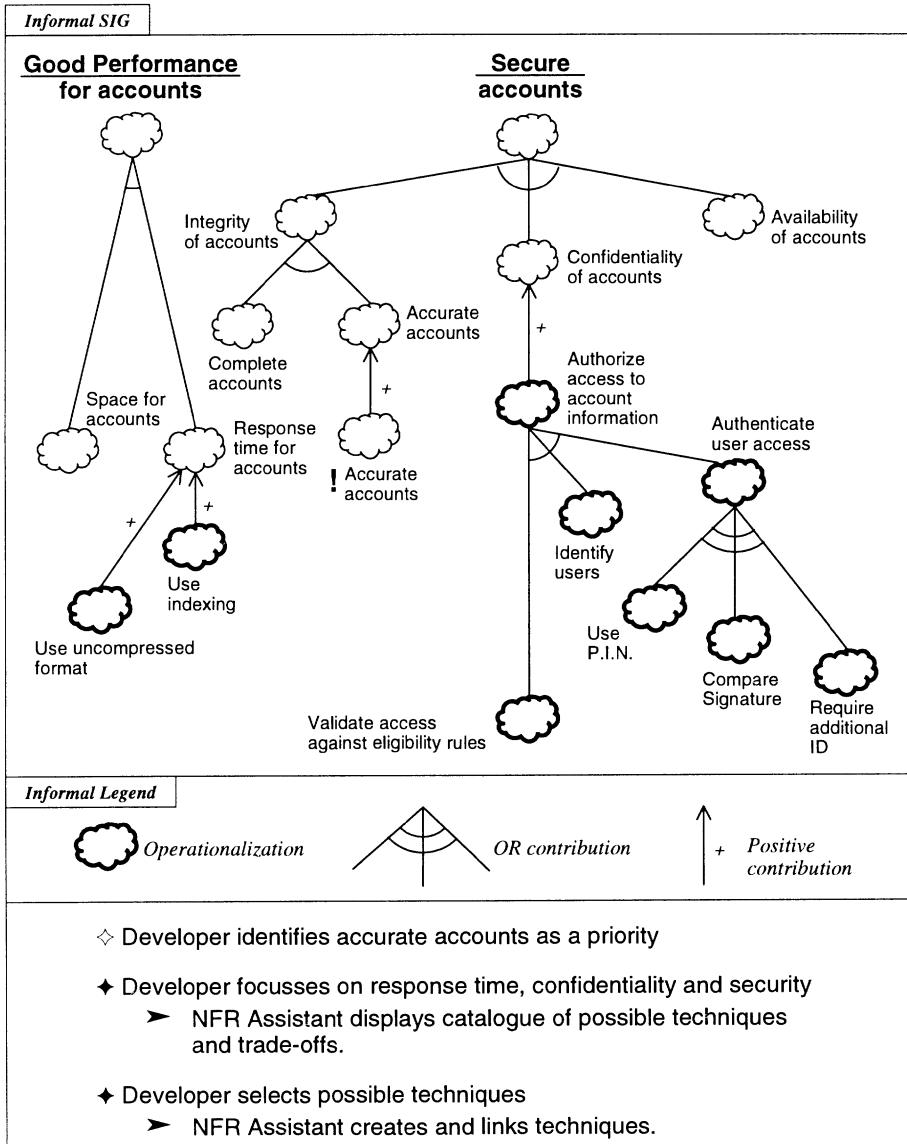
Let us consider the challenge of providing good **Response time** for accounts. One possible alternative is to use indexing (Figure 2.8). In this case, **Use indexing** is a development technique that can be implemented. It is a candidate for the task of meeting the response-time NFR. But it is no longer a non-functional requirement. This is contrasted with **Response time**, which is still a *software quality attribute*, i.e., a non-functional requirement, since it is not something that can be implemented directly.

We call the development techniques *operationalizations* of the NFR softgoals.

We say that indexing *operationalizes* response time. We also say that the response time NFR is *operationalized by* indexing.

Operationalizing softgoals are drawn as thick (dark) clouds, and are another kind of softgoal.

Note that operationalizations are not limited to operations and functions. Instead, operationalizations can correspond to data, operations and constraints in the target system.

**Figure 2.8.** Identifying possible operationalizations for NFR softgoals.

Like other softgoals, operationalizing softgoals make a *contribution*, positive or negative, towards parent softgoals. Here, the use of indexing helps meet the NFR softgoal for good response time. This positive contribution is shown by “+” in Figure 2.8. There are several contribution types. Earlier, we saw the *AND* contribution type.

In addition, the use of an uncompressed format makes a positive contribution towards meeting the response time requirement. However, as we will see later, it has a negative impact on space performance.

Note that we have not yet chosen which operationalizations will be used in the target system. What we can say now is that the use of either indexing or an uncompressed format, or both, will make a positive contribution towards achieving good response time.

Let us now consider how to meet the security requirement, particularly **Confidentiality of accounts**. One development technique is to allow only authorized access to account information. **Authorize access to account information** is an operationalizing softgoal which makes a positive contribution to confidentiality.

The operationalizations can be drawn from catalogues of development techniques, based on expertise in security, performance, and information system development. Catalogues can aid the search for possible operationalizations.

The transition from NFR softgoals to operationalizing softgoals is a crucial step in the process, because NFRs need to be converted into something that can be implemented. However, one may not be able to convert initial requirements into a concrete operationalization in one step, i.e., the initial operationalization (development technique) may not be specific enough. Often, there needs to be further refinements and elaborations. Furthermore, there can be different ways for refining or elaborating these general operationalizing softgoals, i.e., one needs to continue to identify other possible development techniques and choose among them. In the NFR Framework, we continue to treat these operationalizing softgoals – both general ones as well as increasingly specialized and specific ones – as softgoals to be addressed. This allows us to use the same systematic framework to decompose operationalizing softgoals into more specific ones.

For example, the operationalizing softgoal **Authorize access to account information** can be detailed (further operationalized) in terms of a combination of:

- identifying users,
- authenticating user access, *and*
- validating access against eligibility rules.

The *AND* contribution joining operationalizing softgoals means that all three offspring have to be achieved, in order to achieve the parent softgoal, **Authorize access to account information**. If any one is not achieved, the parent softgoal will not be achieved.

In turn, authenticating user access can be decomposed into any one of several options: using a personal identification number (**PIN code**), comparing signatures, *or* requiring additional ID. Here, the contribution type is *OR*, since any one of the offspring can be used to meet the parent. That is, Authentication can be accomplished by using a PIN code, by comparing signatures, *or* by requiring additional identification.

Catalogues also show possible decompositions of operationalizations into more specialized operationalizations. The catalogues also show the contributions of specialized operationalizations towards their parents.

AND and *OR* contributions are drawn with arcs, and the direction of contribution is towards the arcs. However the other contribution types (such as “+” in the figure) are drawn with arrowheads on the lines, showing the direction of *contributions* toward parent softgoals. Note that the direction of the arrows is typically the opposite of the sequence in which softgoals are generated. Contribution types affect how the graph is evaluated, and are explained in full in Chapter 3.

We use the term *decomposition* when the parent and offspring have the same kinds of softgoals. We have already seen decomposition of NFR softgoals to other NFR softgoals, as well as decompositions of operationalizing softgoals into other operationalizing softgoals.

2.8 DEALING WITH IMPLICIT INTERDEPENDENCIES AMONG SOFTGOALS

At each step in the process, when we make choices in order to achieve a particular non-functional requirement (say, security of account information), it is very likely that some other non-functional requirements (e.g., user-friendly access, dealing with ease of use of the interface to a system) may be affected, either positively or negatively, at the same time. These interactions are very important because they have an impact on the decision process for achieving the other NFRs.

These interactions include positive and negative contributions. These different interactions can be dealt with in different ways.

We have already seen how developers can explicitly state interdependencies among softgoals, by using refinements. We call these *explicit interdependencies*. They are shown as solid lines in figures.

Now we consider interdependencies which are *detected* by comparing a portion of a SIG with a catalogue of relationships among softgoals. These *implicit interdependencies* (*correlations* among softgoals) are shown as dashed lines in figures.

Figure 2.9 shows some *correlations* (*implicit interdependencies*) among softgoals. These include positive and negative contributions.

Using an uncompressed format is negative (shown as “-”) for space (because compressed formats are more space-efficient), but positive (“+”) for response time (because we don’t need to uncompress the data before processing it). The positive contribution was stated explicitly. Now the negative part of

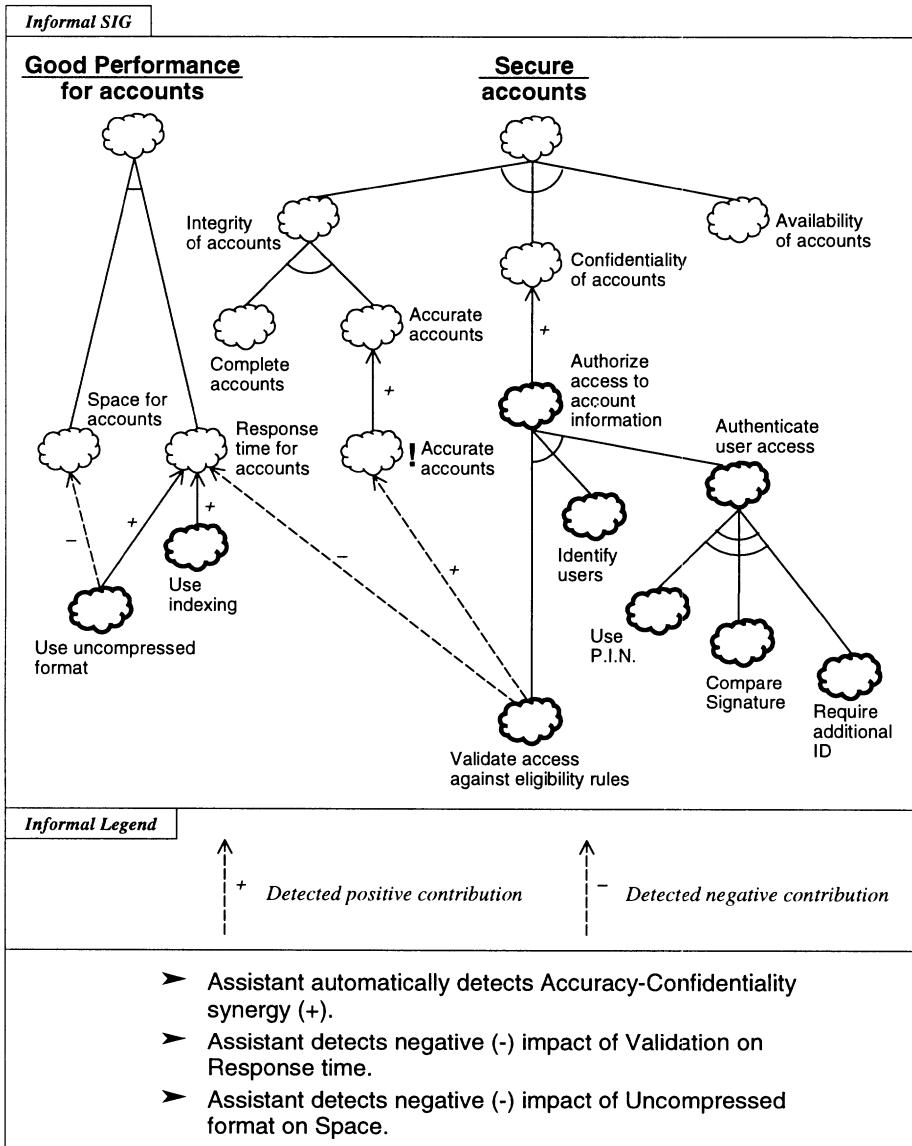


Figure 2.9. Detecting implicit interdependencies among existing softgoals.

this time-space tradeoff is detected using a correlation. By showing positive and negative contributions, correlations are one way of recording tradeoffs.

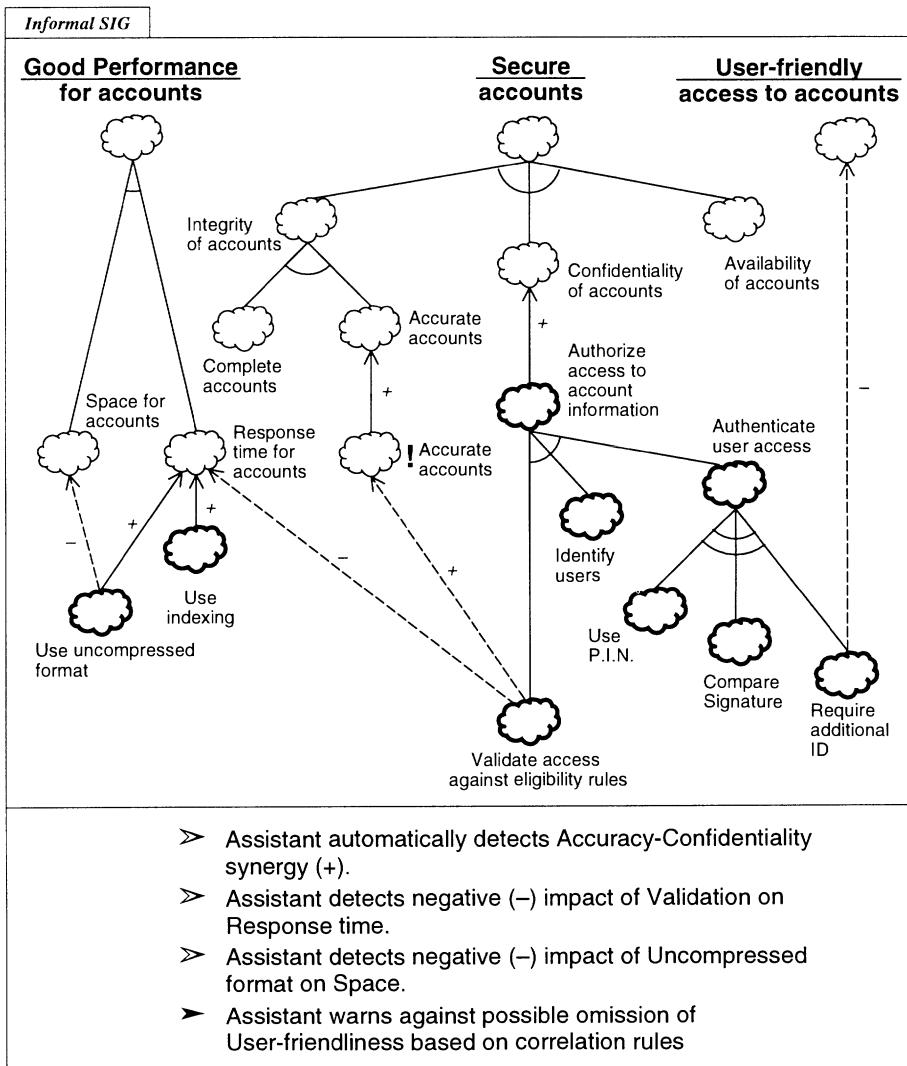


Figure 2.10. Detecting implicit interdependencies among existing and other softgoals.

As another example, **Validate access against eligibility rules** is one component of **Authorize access to account information**, which operationalizes the **Confidentiality of accounts** NFR softgoal. However, besides contributing to con-

fidentiality, validation also happens to have a positive effect on the accuracy of accounts, since ill-intentioned users can be denied access and prevented from committing forgery. On the other hand, **Validate access against eligibility rules** has a negative contribution to **Response time for accounts**, since validation induces extra overhead.

Implicit interdependencies can be detected as the graph is being developed. This is done by consulting (“by hand,” or with tool support) catalogues of positive and negative interdependencies among softgoals. These *correlation catalogues*, are discussed below in Section 2.12.

Figure 2.9 added *interdependency links* for correlations. Now we can consider correlations which add *softgoals*.

The examination of correlation catalogues may also lead to the identification of related NFRs which had not previously been considered relevant. In decomposing **Authenticate user access**, for example, one of the alternatives – **Require additional ID** – is detected to have a negative impact on **User-friendly access to accounts**. Although we had not been considering it in Figure 2.9, note that the NFR softgoal **User-friendly access to accounts** now appears in the SIG of Figure 2.10 as part of a correlation.

Thus correlations can add *softgoals* (Figure 2.10), as well as *interdependencies* (Figure 2.9) to softgoal interdependency graphs.

2.9 RECORDING DESIGN RATIONALE

An important premise of the Framework is that design decisions should be supported by well-justified arguments or *design rationale*. Reasons can be stated for making refinements, for selecting an alternative for the target system, etc. The Framework extends the goal-oriented approach to the treatment of arguments.

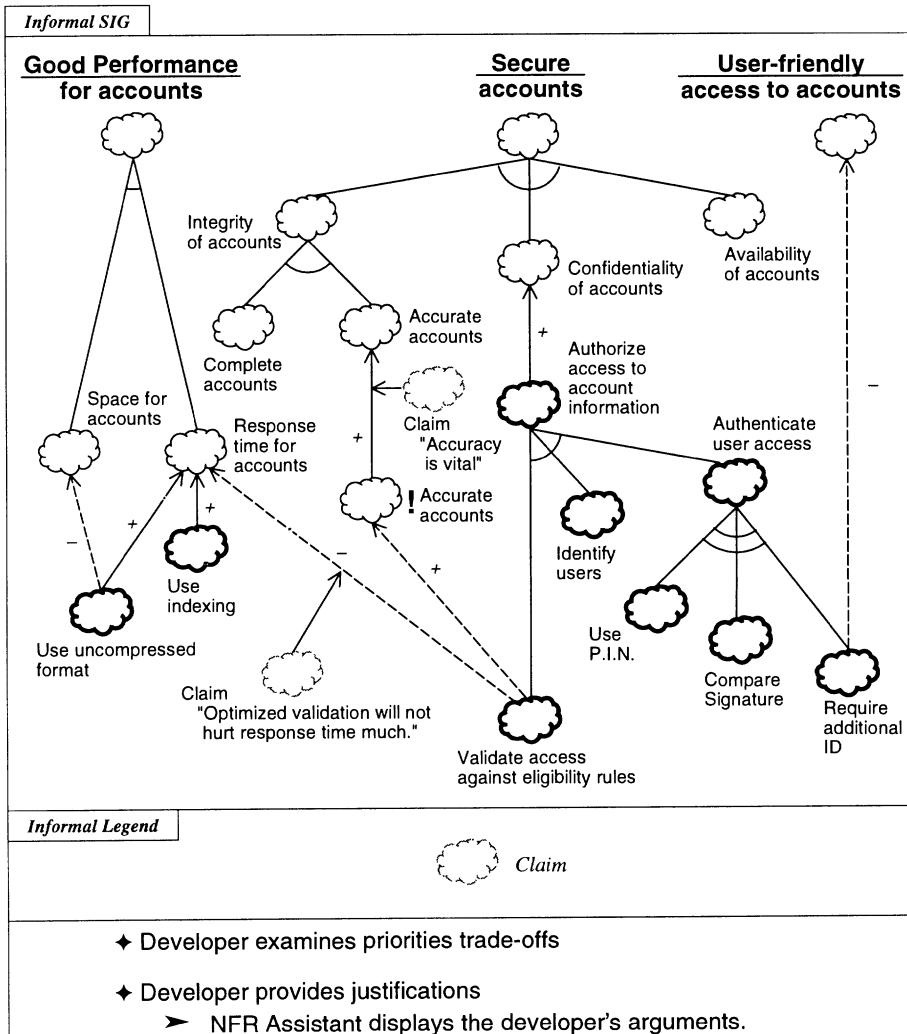
Let us consider two examples. First, we can state the reason for a *prioritization*, here of **Accurate accounts**. To support this prioritization, in Figure 2.11 we write **Claim “Accuracy is vital.”**

We call this a *claim softgoal* (or *claim*). Claim softgoals are the third kind of softgoal. Earlier we saw NFR softgoals and operationalizing softgoals.

Here the claim is attached to an interdependency link, which connects the prioritized softgoal **!Accurate accounts** to its parent, **Accurate accounts**.

Note that we use the term *claim softgoal* to refer to the statement itself. When the statement is used to argue for or against something, that constitutes an *argument*, i.e., the argument refers to the *relationship* between the claim and the thing argued about. Claim softgoals are represented as dashed clouds in the softgoal interdependency graph. Their interdependency links represent the arguments.

As a second use of claims, we can rationalize *tradeoffs*. Here, the specialized operationalization **Validate access against eligibility rules** helps to achieve the more general operationalization, **Authorize access to account information**.

**Figure 2.11.** Recording design rationale.

At the same time, it has generated some correlations. It has a *positive* impact on the softgoal for accuracy of accounts, which is a priority. However, it also has a *negative* impact on the softgoal for good response time for accounts.

In this case, the developer weighs the tradeoff and feels that the *Validate access against eligibility rules* operationalizing softgoal is still worth considering,

despite its negative contribution to **Response time for accounts** (Figure 2.11). In fact, this operationalizing softgoal will be chosen in the next section.

To support this position, the developer introduces the claim: **Claim** [“Optimized validation will not hurt response time much.”] The claim notes that optimization of “Validate access against eligibility rules” will mitigate its degradation of **Response time for accounts**.

Rationale can draw on domain information, such as organizational priorities and organizational workload (e.g., the number of credit card sales per day), as well as development expertise.

Note that as these factors change, the reasons and decisions may change. For example, if the volume of daily sales rises dramatically, then priorities may change. These could be reflected in the rationale, and different operationalizations could be selected, which could have an impact on meeting the top softgoals.

Claims are treated as softgoals related by interdependencies and associated contributions, in the same way that NFRs and their operationalizations are. Claims make *contributions*, positive or negative, to other softgoals. Such contributions are not shown in the figures of this chapter, but will be discussed in the next chapter.

2.10 SELECTING AMONG ALTERNATIVES

The refinement process continues until the developer considers that the possible solutions for the target system are sufficiently detailed, and that no other alternatives need be considered.

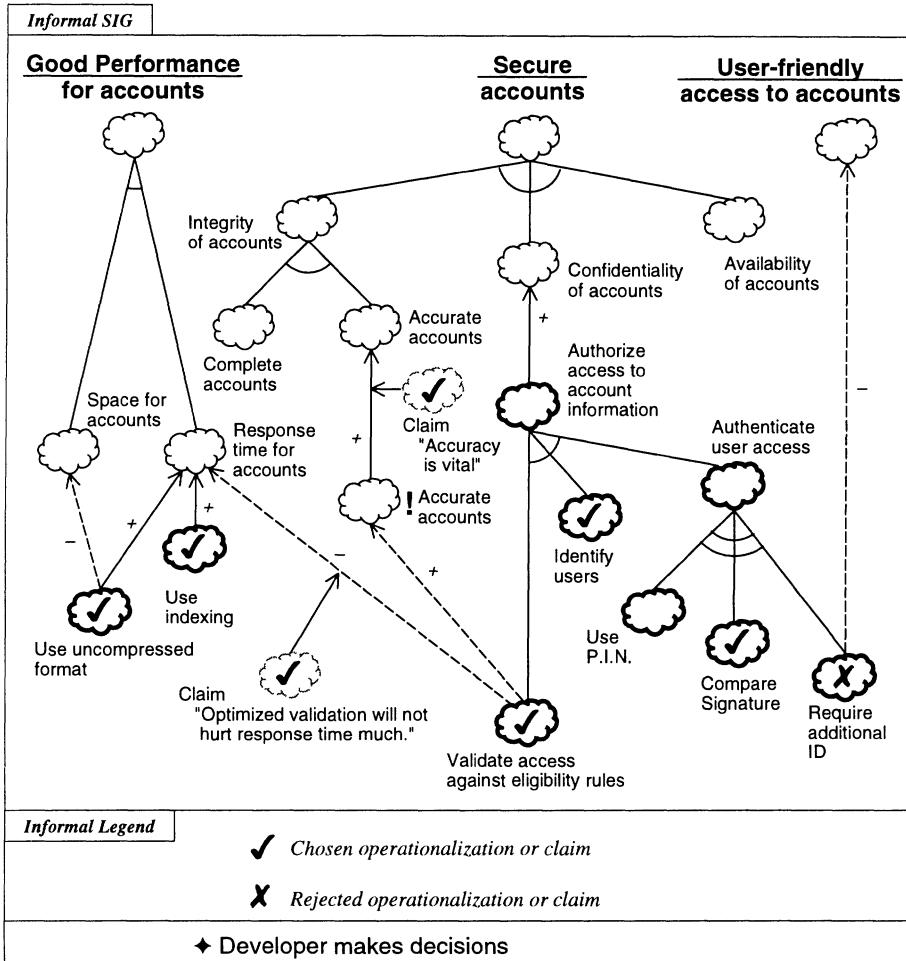
Along the way, the developer has considered NFRs, domain information, and addressed ambiguities, priorities and tradeoffs. The developer has then considered possible operationalizations, design rationale, and interdependencies among softgoals, using the expertise from the developer and from catalogues. Of course, early decisions need to be considered and reconsidered when making later decisions. All this information is recorded in the graph, and is available to help the developer select among alternatives.

Thus, developers are aided in their decision-making, by referring to the graphs, which have a history of the design process.

In expanding the softgoal interdependency graph, the developer is elaborating on the possible subparts and alternatives that one would like to consider as means for achieving the initial high-level softgoals. The graph thus represents a design space over which design decisions are to be made.

As softgoals are being refined, the developer will eventually reach some softgoals which are sufficiently detailed. The developer can accept or reject them, as part of the target system.

Now it is time to choose from among the possible operationalizations, in order to produce a target system. In addition, appropriate design rationale should be selected.

**Figure 2.12.** Selecting among alternatives.

There are choices of *operationalizing softgoals*. Of the many target alternatives in the graph (Figure 2.12), some are chosen (selected or “satisficed,” indicated by “✓”) and others are rejected (denied, indicated by “✗”).

Having identified three possible ways for authenticating user access, the developer decides that the **Compare Signature** operationalization is acceptable. Chosen solutions are represented in the graphical notation as a check-mark (“✓”) inside the node (Figure 2.12). On the other hand, rejected candidates, such as **Require additional ID**, are shown as “✗”.

To aid confidentiality, **Identify users** and **Validate access against eligibility rules** are also selected. To aid response time, indexing and an uncompressed format are chosen. Note that a decision need not be made for every operationalizing softgoal. This is the case for **Use P.I.N**, which is left blank.

As is the case for operationalizing softgoals, *claim softgoals (claims)* are also either accepted (satisficed) or rejected (denied). The claim for prioritizing accuracy is accepted, hence a check-mark (“ \checkmark ”) is placed in the **Claim[“Accuracy is vital.”]** claim softgoal. Likewise, the claim **Claim[“Optimized validation will not hurt response time much.”]** is accepted.

Now we turn to the impact of these decisions on top requirements.

2.11 EVALUATING THE IMPACT OF DECISIONS

The *evaluation* of softgoals and interdependencies determines the impact of decisions. This indicates whether high-level softgoals are met.

To determine the impact of decisions, both current and previous decisions are considered. Previous considerations and decisions are already reflected in the graph, as softgoals and interdependencies.

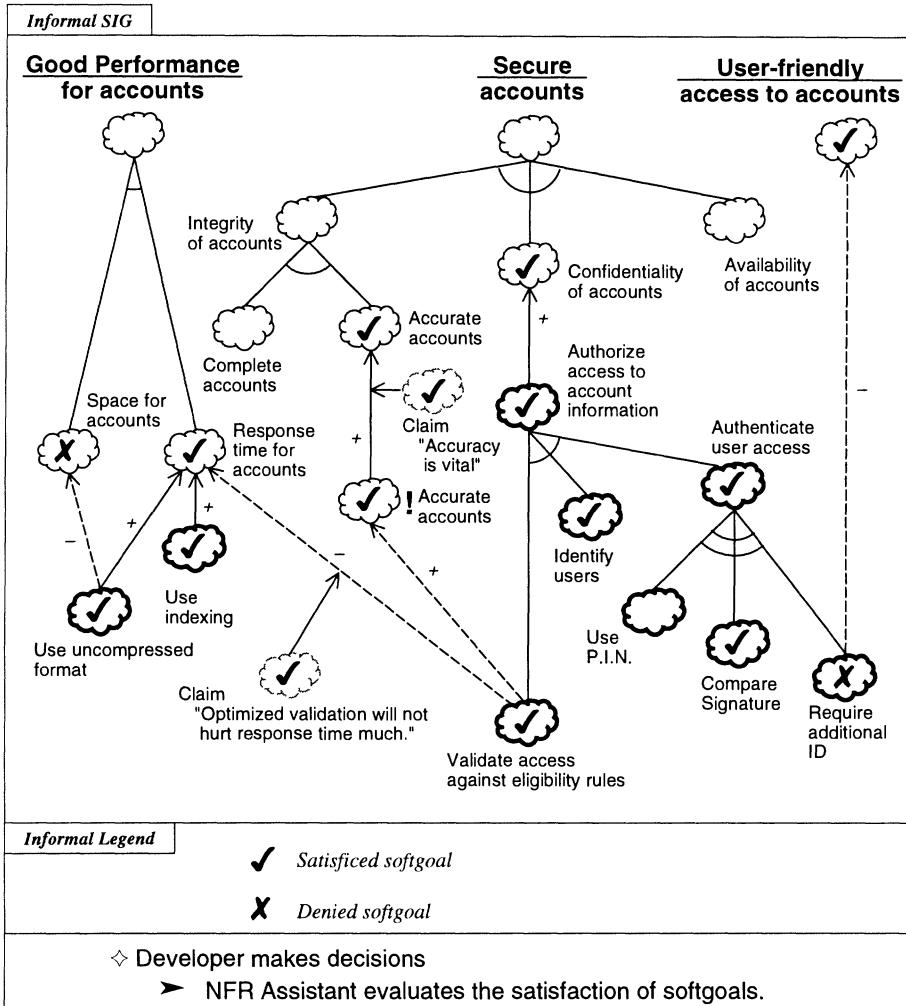
To reflect the nature of design reasoning, the evaluation and propagation of design decisions focusses on the question of whether a chosen alternative is “good enough,” i.e., whether it meets a softgoal *sufficiently*. This style of reasoning is appropriate for dealing with non-functional requirements since meeting these requirements is often a matter of degree, not a binary true-false decision. The NFR Framework builds on the notion of *satisficing*, which was used by Herbert Simon [Simon81] to refer to finding solutions that are sufficiently good, even if they may not be optimal. To emphasize the difference between this style of goal-oriented reasoning from the more conventional, binary logic-based, goal-oriented reasoning (e.g., [Nilsson71]), we use the term *softgoal* to refer to the kind of goal that requires satisficing.

The evaluation process can be viewed as working bottom-up, starting with decisions, which are often leaves of the graph, and often at its bottom. Then the *evaluation procedure* (or *labelling algorithm*) works towards the top of the graph, determining the impact on higher-level main softgoals. These top softgoals reflect overall non-functional requirements as stated by the developer and people in the organization for which the system is being developed.

In the NFR Framework, the evaluation of softgoals is represented by assigning labels (such as “ \checkmark ” and “ \times ”) to the clouds (softgoals) in the graph. Labels may be assigned by the developer, or computed from contributions from other nodes.

Roughly speaking, when there is a single offspring, a *positive* contribution “propagates” the offspring’s label to the parent. Thus a satisficed offspring results in a satisficed parent, and a denied offspring results in a denied parent.

On the other hand, a *negative* contribution will take the offspring’s label and “invert” it for the parent’s label. That is, a satisficed offspring leads to a denied parent, and a denied offspring leads to a (somewhat) satisficed parent.

**Figure 2.13.** Evaluating the impact of decisions.

This is the case in the lower left of Figure 2.13. The operationalizing softgoal **Use uncompressed format** makes a negative contribution towards the NFR softgoal **Space for accounts**. In addition, **Use uncompressed format** is satisfied (the label is shown as “✓”). When the satisfied label and negative contribution are considered, the result is that **Space for accounts** is *denied* (“✗”).

Note that in Figure 2.12, “ \checkmark ” indicated “leaf” operationalizations or claims which were selected directly by the developer. Now in Figure 2.13 (and the remainder of this book), its meaning is made more general. It indicates softgoals which are determined to be *satisficed*; this is determined by the developer, or by the evaluation procedure. Likewise “ \times ” represented rejected operationalizations or claims, but now more generally indicates *denied* softgoals.

Suppose a softgoal receives contributions from more than one offspring. Then the contribution of each offspring toward the parent is determined, using the above approach. The individual results are then combined.

For example, **Response time for accounts** has three offspring. **Use compressed format** is satisficed and makes a positive contribution, hence its individual result would be to satisfy the parent. The same is true for **Use indexing**. If these two were the only offspring, the combination of their individual positive results would lead to satisficing the parent. However, **Validate access against eligibility rules** is satisficed and makes a somewhat negative contribution to response time.

The developer combines the results — two satisficed and one denied — and sees there is a conflict. What value should be assigned to the parent softgoal? The developer could assign various values — satisficed, denied, or something in between. Here, the developer notes that claim that optimized validation will not hurt response time much, and labels **Response time for accounts** as satisficed.

Using the rule for satisficed softgoals and positive contribution links, **!Accurate accounts** is noted as being satisficed, and so is **Accurate accounts**.

It is interesting to note that the evaluation procedure works the same way, whether the interdependency link was explicitly stated by the developer, or implicitly detected.

Let us continue at the bottom right of Figure 2.13. Requiring additional identification is not chosen for authenticating access. This helps satisfy the requirement for user-friendly access to accounts.

Compare Signature and its siblings participate in an *OR* contribution to their parent, **Authenticate user access**. This means that if any of the offspring is acceptable, the parent will be acceptable. **Compare Signature** was chosen, so the operationalizing softgoal **Authenticate user access** can thus be automatically evaluated to be satisficed (“ \checkmark ”).

Earlier, the softgoals **Identify users** and **Validate access against eligibility rules** were selected, hence satisficed. Since they and **Authenticate user access** are all satisficed, then a checkmark can be propagated “upwards” along the *AND* contribution to the **Authorize access to account information** softgoal. Then **Confidentiality of accounts** is satisficed.

Let us take stock of how well the target system would meet main non-functional requirements. Accuracy, confidentiality, response time and user-friendly access requirements have been satisficed. The space requirement has been denied. For other requirements, such as performance, integrity and security, we have not indicated if they are satisficed or denied. To obtain answers

for some of these, we would need further refinement, further information, or a resolution of conflicts.

Interestingly, we have been able to address interactions between different kinds of non-functional requirements (here, accuracy and performance), even though the NFRs were initially stated as separate requirements.

In the NFR Framework, a softgoal can be assigned a *label*. So far, we have used labels such as “ \checkmark ” or “ \times ”. In fact, there are actually more “shadings” of label values than these ones. For example, softgoals can be weakly satisfied, or weakly denied. The propagation of these labels along interdependency links depends on the type of contribution.

We have shown some contribution types, such as *AND* and *OR*, in this chapter. There are also other contribution types indicating various combinations of positive and negative, and partial and sufficient contributions. In addition, *claims* provide positive or negative contributions; these are omitted from the Informal SIGs in Figures 2.11 through 2.14. Full details of labels and contribution types are given in Chapter 3.

The propagation of labels is interactive, since human judgement is needed at various points. The evaluation procedure (whether executed by a developer “by hand” or using an automated tool) will propagate labels as far as it can, at which point the developer can step in to provide values as appropriate. At any time, the developer can override previously assigned or computed labels, and can change contribution types. Details of the evaluation procedure are given in Chapter 3.

Developers can assess the status of their designs at any time, by examining how the status of the most detailed decisions contribute towards the top-level softgoals that they started with. The developer can thus make informed tradeoffs among the available alternatives.

Relating Functional Requirements to Decisions and NFRs

We can also relate *functional requirements* to NFRs and the decisions made for the target system.

So far, graphs started with top NFRs and resulted in operationalizations being selected. Now we graphically relate them to functional requirements and their associated chosen target design *specification* or *implementation*.

The top of Figure 2.14 shows the functional requirements (for maintaining accounts) and the top level NFRs, relating to the security, performance, etc., of maintaining accounts. The bottom of the figure links the chosen operationalizations to a description of the target system (shown in a rectangle at the bottom right). The right side of the figure links this description of the target system to the (source) functional requirements (shown in an oval at the top right).

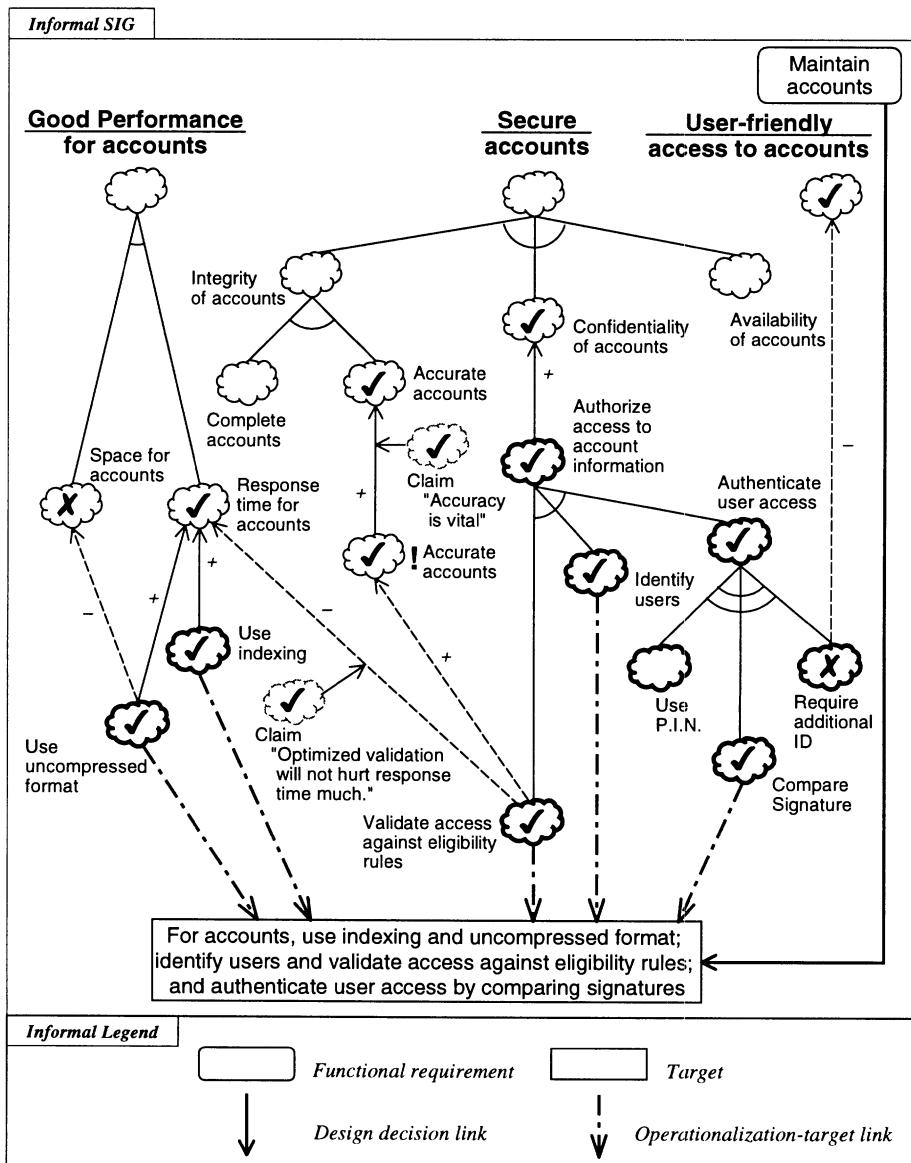


Figure 2.14. Relating decisions to Functional Requirements.

2.12 CATALOGUING DEVELOPMENT METHODS AND CORRELATIONS

Let us further consider how a developer comes up with refinements, including NFR decompositions, operationalizations, and claims that justify decisions.

In order to be able to bring relevant knowledge to the attention of the developer at each point in the design process, knowledge needs to be represented in a flexible catalogue structure. Three major kinds of catalogues are used to express design knowledge:

- *NFR type catalogues*: They encode concepts about particular types of NFRs, such as security and performance.
- *method catalogues*: They encode knowledge that helps refine graphs by decomposing softgoals and considering operationalizations.
- *correlation rule catalogues*: They encode knowledge that helps detect implicit interdependencies among softgoals.

Developers can browse through these catalogues to examine a current area of interest within a wide range of possible techniques.

Development knowledge can be catalogued to organize NFR types, methods, and correlation rules, which are used as a developer faces particular design decisions.

So far we have shown only a catalogue of NFR types in Figure 2.1. Let us now consider other kinds of catalogues.

Method Catalogues

Figure 2.15 shows a *catalogue of methods* for addressing the NFR of confidentiality. The catalogue is hierarchically classified: more specific methods (techniques) are placed under general ones.

The concept of *method* is applied uniformly to refinement throughout the Framework. Thus there are methods to assist in the decomposition of NFRs, methods for operationalizing, and methods for argumentation to identify claims.

Let us consider how catalogues can be used to aid refinement. Recall Figure 2.2, where we were starting with the NFR softgoal that accounts be maintained securely and with good performance. A search of catalogues (which can be done by hand, or be tool-assisted and semi-automated) for NFR types and methods could result in a suggestion to decompose an NFR softgoal into several sub-softgoals, which use sub-types of the type of the parent softgoal. This occurs through the invocation of a *method* which encodes this knowledge, here, about sub-types.

Some methods are domain-specific, e.g., dealing with particular development techniques and domains. Some other methods are fairly generic, applying to all softgoals of a particular NFR type. Others are quite generic, applying to a variety of domains and NFR types, e.g., a method decomposing an NFR

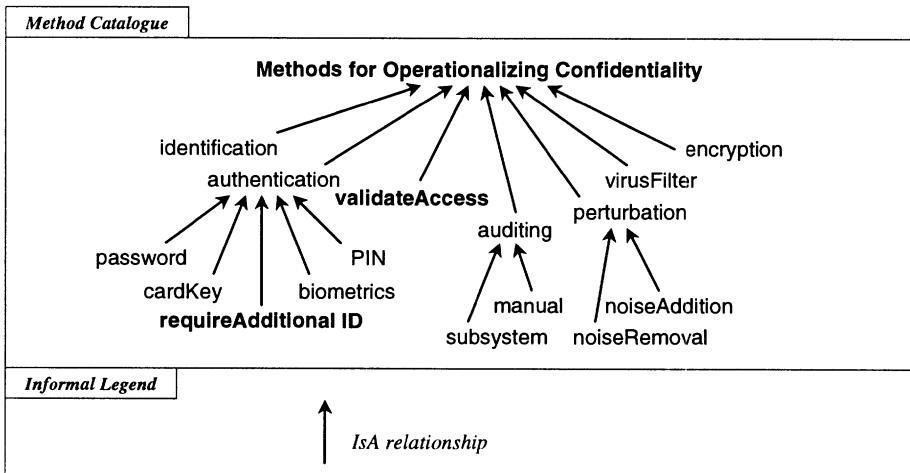


Figure 2.15. A catalogue of operationalization methods for achieving confidentiality.

softgoal for a data item into NFR softgoals for all components of that item. Methods that are generic tend to be more powerful, because they are applicable more widely. However, they can be harder to come up with. Many generic methods are already built into the Framework, and are developed by the Framework developers.

The NFR Framework comes with a collection of generic methods, which are described in Chapter 4. Other method catalogues are available for accuracy, security and performance requirements, and are discussed in Chapter 4 and Part II.

Note that the developer may have to select among a number of suggestions, possibly coming from different knowledge sources. The developer can also adapt a method by overriding some of its features.

Correlation Catalogues

Now let us consider a *catalogue of correlations*, showing *implicit interdependencies*. Figure 2.16 is an example of a catalogue of implicit interdependencies, and their contributions.

This catalogue shows the impact of various operationalizing softgoals (such as Validation and Indexing, shown on the vertical axis) upon NFR softgoals (such as Accuracy and Response Time, shown on the horizontal axis). The correlation catalogue entries show the contribution type. For example Validation makes a positive contribution towards Accuracy, and Indexing has a positive contribution towards Response Time.

Informal Correlation Catalogue						
Impact of offspring <i>Operationalizing Softgoal</i>	upon parent <i>NFR Softgoal</i>					
	Accuracy	Confidentiality	Response Time	Space	User-friendliness	
Validation	+	+	-			
Compression			-	+		
Indexing			+			
Authorization		+				
Additional ID		+				-

Figure 2.16. A catalogue showing the impact of operationalizing softgoals upon NFR softgoals.

Detected correlations are shown as dashed lines in graphs. In Figures 2.9 and 2.10, the implicit interdependencies between **Use uncompressed format** and **Space for accounts**, between **Require Additional ID** and **User-Friendly Access**, and among **Validate Access against Eligibility Rules**, **Response Time**, and **Accuracy of Accounts** are detected by correlation rules.

Catalogues can be prepared with different axes showing correlations between different groups of softgoals. For example, another correlation catalogue could show the impact of NFR softgoals upon other NFR softgoals. Yet another correlation catalogue could show the impact of operationalizing softgoals upon other operationalizing softgoals.

Catalogue-based correlations are detected by pattern-matching. Correlations can be used to establish interdependencies in a graph either automatically or after confirmation by a developer, depending on the developer's wishes. Like methods, correlations can be generic or specific.

2.13 DISCUSSION

In this chapter, we offered a glimpse into how the NFR Framework provides a systematic treatment of non-functional requirements, and how they are methodically introduced into steps in the design process.

During this process, the developer is constructing a record of how sub-softgoals contribute to higher softgoals, eventually contributing to the top-level softgoals. Throughout the development process, both selected and discarded

alternatives form part of the development history, and the softgoal interdependency graph keeps track of the impact of decisions upon the top-level softgoals. These records are displayed graphically.

Catalogues of NFR knowledge help with the time-consuming, and often difficult, search for development techniques.

It can be possible, at least for small examples, to use the NFR Framework starting from “first principles,” without the use of catalogues. However, our experience has been that catalogues are quite important, especially when considering NFRs for larger systems.

While it can take some time to develop methods and catalogues, our experience is that this results in a payoff, by speeding up the refinement process. Chapter 4 presents generic catalogues of methods and correlations, while Part II presents catalogues for particular types of NFRs (accuracy, security and performance).

Literature Notes

This chapter is adapted from [Chung94a,b].

The goal-oriented approach we have taken is meant to be intuitive, following and supporting the developer’s reasoning. However, it is not the same as the approaches in artificial intelligence or automated reasoning, such as AND-OR trees [Nilsson71]. By taking a “satisficing” approach [Simon81], the aim is to offer flexible support for a developer’s reasoning, and to allow it to be recorded. The purpose is not to automate development. Instead, development is intended to be highly interactive, where the human developer is in control.

The catalogues of refinement techniques and tradeoffs (e.g., [Chung93a] [Nixon97a] [Yu94b]) are based on work done by researchers and practitioners in particular areas. These include security [ITSEC91, Parker91, Clark87, Martin73], accuracy [Pfleeger89], and performance [C. Smith90, Hyslop91], as well as information system development. In addition, catalogues can deal with other NFRs, other classes of issues, e.g., Business Process Reengineering [Hammer95], and other aspects of development, e.g., software architecture [Garlan93].

3

SOFTGOAL INTERDEPENDENCY GRAPHS

In this chapter and the next, we present the NFR Framework in more detail. The NFR Framework helps developers deal with non-functional requirements (NFRs) during software development. The Framework helps developers express NFRs explicitly, deal with them systematically, and use them to drive the software development process rationally.

By focussing on the *process* of meeting requirements, the NFR Framework is process-oriented and complementary to traditional product-oriented approaches. Control of the process and the focus of development is given to the developer, who is helped with refining softgoals, dealing with tradeoffs and selecting among alternatives. All of this can be facilitated by catalogues of knowledge for specific non-functional requirements and specific domains.

The Framework provides a *goal-oriented* approach to dealing with NFRs. Unlike goals in traditional problem-solving and planning frameworks, non-functional requirements can rarely be said to be “accomplished” or “satisfied” in a clear-cut sense. Instead, different design decisions contribute positively or negatively, and with varying degrees, towards attaining a particular non-functional requirement. Accordingly, we speak of the *satisficing* of *softgoals* to suggest that generated software is expected to satisfy NFRs within acceptable limits, rather than absolutely.

Softgoals can be stated and refined, and alternatives can be chosen to move towards a target system. Consideration of design alternatives, analysis

of design tradeoffs and rationalization of design decisions are then all carried out in relation to the stated softgoals.

Softgoals are interdependent. To help a developer address non-functional requirements, the NFR Framework represents *softgoals* and their *interdependencies* in *softgoal interdependency graphs (SIGs)*. SIGs maintain a complete record of development decisions and design rationale, in concise graphical form. This graphical record of decisions made includes non-functional requirements and their associated alternatives, decisions, and reasons for decisions. To determine if top-level requirements are met, an evaluation procedure is offered.

The NFR Framework consists of five major components:

1. *Softgoals* are the basic unit for representing non-functional requirements. They help a developer deal with NFRs, which can have subjective, relative and interacting natures. There are three kinds of softgoals. *NFR softgoals* (or *NFRs*) represent non-functional requirements to be satisfied. *Operationalizing softgoals* are development techniques which help satisfy NFRs. *Claim softgoals* help justify decisions.
2. *Interdependencies* state interrelationships among softgoals. They record *refinements* of softgoals into offspring softgoals, and the *contributions* of offspring towards the satisfying of parent softgoals. Interdependencies can also state more general refinements among softgoals and interdependencies.
3. An *evaluation procedure* determines the degree to which a given non-functional requirement is satisfied by a set of design decisions.
4. *Methods* offer the developer catalogues of development techniques. They are realized by refining softgoals into other softgoals.
5. *Correlations* provide catalogues for inferring possible interactions, both positive and negative, among softgoals.

Methods and correlations help generate a softgoal interdependency graph, by drawing on catalogued knowledge of NFRs and software development. This helps knowledge of softgoal refinements and their interactions to be captured, catalogued, tailored and reused. Each method is instantiated explicitly by the developer, resulting in an *explicit interdependency*. On the other hand, correlations are not instantiated explicitly by the developer but are used to detect *implicit interdependencies*.

This chapter and the next present the five components of the NFR Framework. This chapter presents softgoals, interdependencies (refinements and contributions), and the evaluation procedure. The next chapter will present catalogues of methods and correlations.

The examples throughout this chapter continue with NFRs for the credit card account management information system introduced in Chapter 2.

3.1 KINDS OF SOFTGOALS

As discussed in Chapter 2, there are three distinct kinds of softgoals:

- *NFR softgoals*, e.g., **Accuracy of Accounts**,
- *operationalizing softgoals*, e.g., **Indexing**, and
- *claim softgoals*, e.g., “**Gold Card Accounts are important**”.

NFR softgoals act as overall constraints on the system, and are satisfied by the operationalizing softgoals which represent design or implementation components. Claim softgoals provide rationale for development decisions.

Only operationalizing softgoals appear in some form (such as an operation, data representation, constraint or assignment of an external agent to a task) in the target design or implementation. However, all the three kinds of softgoals appear in a softgoal interdependency graph, and may appear in supporting documentation.

Each softgoal has an associated *NFR type* and one or more *topics* (subjects). In **Accuracy of Accounts**, for example, Accuracy is the type and Accounts is the topic. When the type changes (e.g., **Response Time for Accounts**), so does the meaning of the softgoal. Similarly, when the topic changes, so does the meaning of the softgoal. After all, the accuracy of an account is quite different from the response time for an account, the response time for a bank deposit, etc.

In the above paragraphs and in Chapter 2, we used an informal notation for softgoals, e.g., **Response Time for Accounts**. This softgoal can be written more systematically as **ResponseTime[Accounts]**, where **ResponseTime** is the *NFR type* (or *type*), and **Accounts** is the *topic*, surrounded by brackets. Softgoals can have more than one topic, e.g., **FlexibleUserInterface[InfrequentUser, GoldAccount]**. This more systematic notation for softgoals is used in the remainder of this book, for both figures and textual descriptions.

Note that “topics” do not directly correspond to parameters as used in programming languages and standard computer science. The number of topics of a softgoal can vary, even for a given NFR type. In addition, topics can be refined in a specialization hierarchy.

NFR softgoals can have an associated *priority*. Priority softgoals are shown in figures with an exclamation mark (“!”). In addition, the kind of priority is shown in brace brackets. *Critical* softgoals are vital to the success of a system, e.g., **!Accuracy[GoldAccount]{critical}**. *Dominant* softgoals deal with a dominant part of the workload, e.g., **!ResponseTime[AuthorizeSales]{dominant}**.

Clouds represent softgoals in figures. Clouds have an associated *tag*, to describe the softgoal in textual form. Some softgoals and tags are shown in Figure 3.1. Note that descriptions (e.g., *Softgoal tag*) are shown in regular italics in figure legends, while parameterized items (e.g., *NFR Type[topic]*) are shown in sans serif (Helvetica) italics.

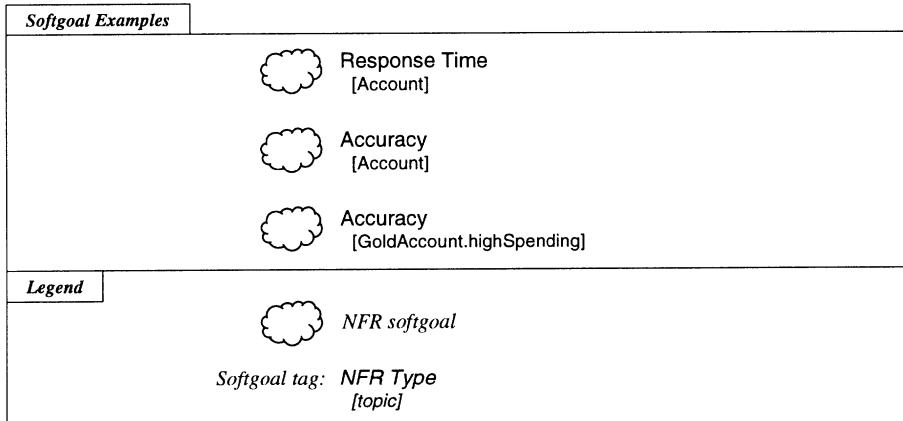


Figure 3.1. Representing sample non-functional requirements as NFR softgoals.

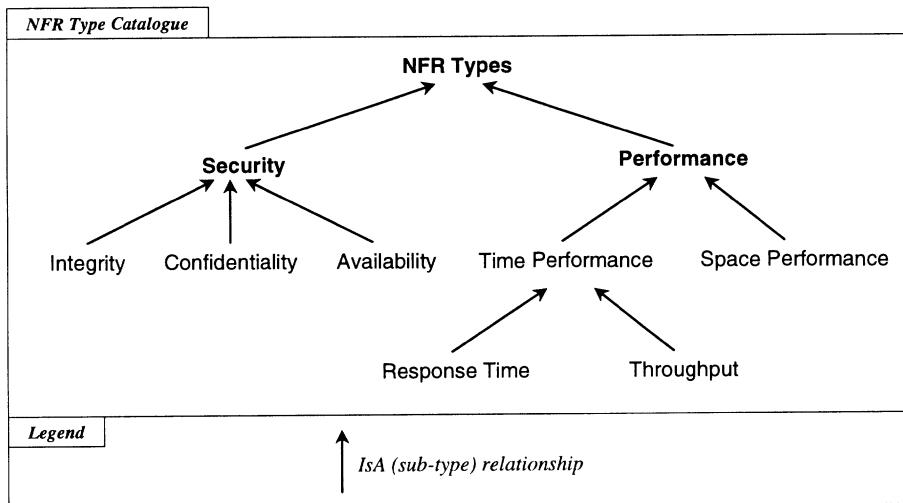


Figure 3.2. Portions of NFR type catalogues.

NFR Softgoals

NFR softgoals represent non-functional requirements as softgoals. The *types* of NFR softgoals are *NFR types*. NFR types include accuracy, performance, security, and other NFRs.

NFR types can be inter-related. The NFR Framework helps organize NFR types by arranging them into catalogues. General NFR types are shown above more specialized ones. They are arranged in IsA hierarchies (i.e., subtype or specialization hierarchies). This way, a large number of types become more manageable, and their often unclear meaning becomes clarified. Examples are shown in the *NFR type catalogue* of Figure 3.2. More detailed NFR type catalogues are discussed in Part II of the book.

Some sample NFR softgoals are shown in Figure 3.1. They are shown as thin (light) clouds. The first NFR softgoal is for response time of accounts, and the second one is for accuracy of accounts.

Let's consider the third NFR softgoal. This is an example of an NFR softgoal's topic referring to more specific information than, for example, the class `GoldAccount`. For example, it could refer to an *attribute* of `GoldAccount`, such as `GoldAccount.highSpending`. The resulting NFR softgoal for accuracy of high spending information of a gold account can be written as: `Accuracy[GoldAccount.highSpending]`. Here the value of `GoldAccount.highSpending` indicates whether spending on a gold account exceeds a pre-set amount. For example, if the total amount of spending on a gold account is \$13 000 when the pre-set amount is \$10 000, then this high spending will be reflected in the value of `GoldAccount.highSpending`. The interpretation of this softgoal is that all the high spending associated with the class of gold accounts ought to be maintained accurately in the system, e.g., in the database.

NFR softgoals act as *global* constraints on a system. They are global because they affect many parts of the system instead of any one particular part. They need to be satisfied by operationalizing softgoals, which will be described shortly.

Operationalizing Softgoals

When developers have refined the initial NFRs into a suitable set of NFR softgoals, they need to find solutions in the target system which will satisfice the NFR softgoals. These solutions provide operations, processes, data representations, structuring, constraints and agents in the target system to meet the needs stated in the NFR softgoals. They are called *operationalizations*, as they *operationalize* the NFR softgoals, i.e., they provide more concrete mechanisms in the target system, which meet the functional requirements and non-functional requirements. They are represented as *operationalizing softgoals*.

Operationalizing softgoals range over different *development techniques*, including *design techniques* during the design phase, or *implementation techniques* during the implementation phase. Put differently, operationalizing softgoals are the result of a decision-making process, and can be viewed as (components of) the solutions considered for the given problem or needs statement.

Figure 3.3 shows sample operationalizing softgoals. They are drawn with thick (dark) clouds. Like other softgoals, operationalizing softgoals have type and topic, where the topics associated with each type depend on the nature of the corresponding operationalizing softgoal.

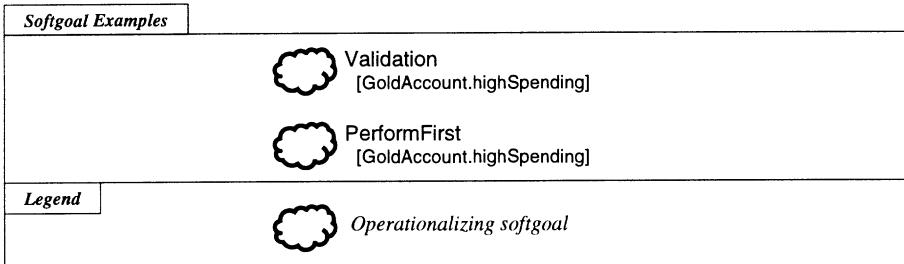


Figure 3.3. Sample operationalizing softgoals.

For instance, one way to satifice the accuracy softgoal mentioned earlier might be to validate all account data entered into the system. This can be represented as an operationalizing softgoal: `Validation[GoldAccount.highSpending]`. Here, `Validation` is the softgoal's NFR type. The softgoal's topic is `GoldAccount.highSpending`, an attribute of `GoldAccount`, as discussed before.

As another example, `PerformFirst[GoldAccount.highSpending]` is an operationalizing softgoal which orders the execution of operations. Operations relating to high spending are performed first. This may help satifice NFRs, e.g., for response time.

Claim Softgoals

Design rationale is represented by *claim softgoals (claims)*. Claim softgoals make it possible for domain characteristics (such as priorities and workload) to be considered and properly reflected into the decision making process. They serve as justification in support or denial of the way softgoals are prioritized, the way softgoals are refined and the way target components are selected.

This way, claim softgoals provide rationale for development decisions, hence facilitating later review, justification and change of the system, as well as enhancing traceability.

Sample claim softgoals are shown in Figure 3.4. Claim softgoals are drawn as dashed clouds. Claim softgoals have a *type* of `Claim`. The *topic* is a statement, typically of evidence or reasons.

Claims can be used to support prioritization, whereby efforts are focussed on priorities. For example, `Claim["One of vital few: high spending in gold accounts."]` argues that priority should be given to softgoals dealing with gold accounts.

Claims can be used to provide reasons for selecting possible solutions. For example, `["Priority actions can be performed first."]` argues for performing priority operations first, to help meet a priority softgoal.

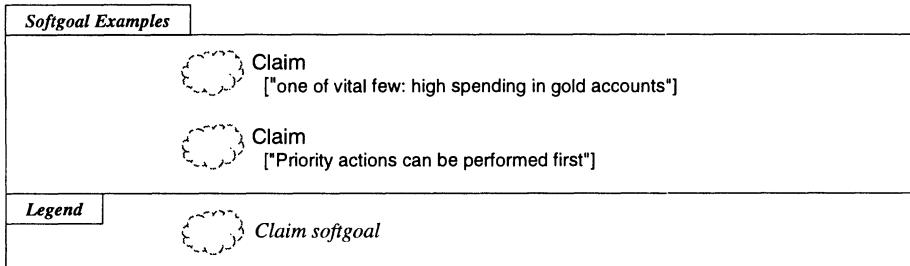


Figure 3.4. Representation of claims softgoals.

Claims can also be expressed in a more formal notation, using predicates. Such predicates can describe situations. For example, in the claim `Claim[vitalFew(GoldAccount.highSpending)]`, we have `vitalFew(...)` which is a *situation descriptor* used to argue that the `highSpending` attribute of gold accounts is of the “vital few,” i.e., is a priority.

During this justification process, claims may be considered weak, counter-intuitive or even wrong. For this reason, even claims themselves sometimes need to be supported or denied, hence acting as evidence or counter-evidence for other claims. Thus, like other kinds of softgoals, claims are also treated as softgoals to be satisfied.

Long Notation for Softgoals

Softgoals are most often written in a compact form, such as `!Accuracy[Account]{critical}`. In addition, softgoals can also be written in a longer form, using a frame-like notation:

NFR Softgoal account-accuracy

```

type: Accuracy
topic: Account
label: ✓
priority: critical
author: Jane Doe
creation time: 15 May 1999

```

An individual softgoal can have an optional identifier, here `account-accuracy`. It is written after the initial keywords which indicate the kind of softgoal (here, NFR softgoal). Later on, the identifier can be used by itself to designate the softgoal.

Each softgoal can also have an associated priority (criticality or dominance). Softgoals can also have other attributes, such as author, and time of creation of the softgoal.

Each softgoal has a *label* which indicates the degree to which the softgoal is “satisficed.” Labels will be explained when the evaluation procedure is presented in Section 3.3.

To summarize, three kinds of concepts, namely, NFR softgoals, operationalizing softgoals, and claim softgoals, are all uniformly treated as softgoals to be satisficed. Of the three, NFR softgoals and operationalizing softgoals need to be achieved if possible. In contrast, claims present domain knowledge and development knowledge which support (or deny) possible development decisions.

This treatment helps the developer to deal with non-functional requirements as design constraints, while considering operationalizing softgoals as development alternatives with associated tradeoffs. At the same time, development decisions are made and design rationale is stated. This broad spectrum of activities is carried out systematically, in order to achieve softgoals rather than to merely evaluate how good the resulting product is.

3.2 INTERDEPENDENCIES

Softgoals are inter-related via *interdependencies*. In the “downward” direction, *refinements* take a softgoal, the *parent*, and produce one or more *offspring* softgoals. Parents and offspring are related by interdependencies. In the “upward” direction, offspring softgoals make *contributions*, positively or negatively, to the satisficing of the parent softgoal.

Figure 3.5 shows a softgoal interdependency graph with some interdependencies resulting from refinements. The direction of contribution is shown by the arrowheads. The arrowhead points to the parent, and away from the offspring. The actual contribution values are not yet shown. They will be presented later.

We now discuss these two aspects of interdependencies, *refinements* and *contributions*.

Refinements

Refinements take an existing softgoal and relate it to other softgoals. There are three kinds of refinements: *decomposition*, *operationalization*, and *argumentation*. The kinds of refinements are shown in a *refinement catalogue* in Figure 3.6.

Decomposition.

Decompositions refine a softgoal into other softgoals of the same kind. There are three kinds of decompositions. These correspond to the three kinds

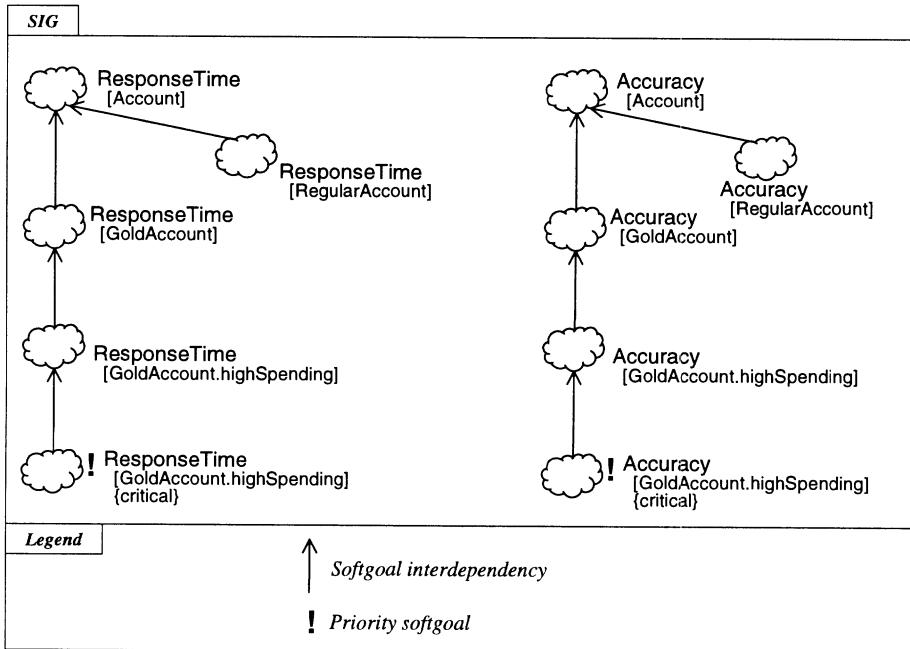


Figure 3.5. Decomposition and prioritization of NFR softgoals.

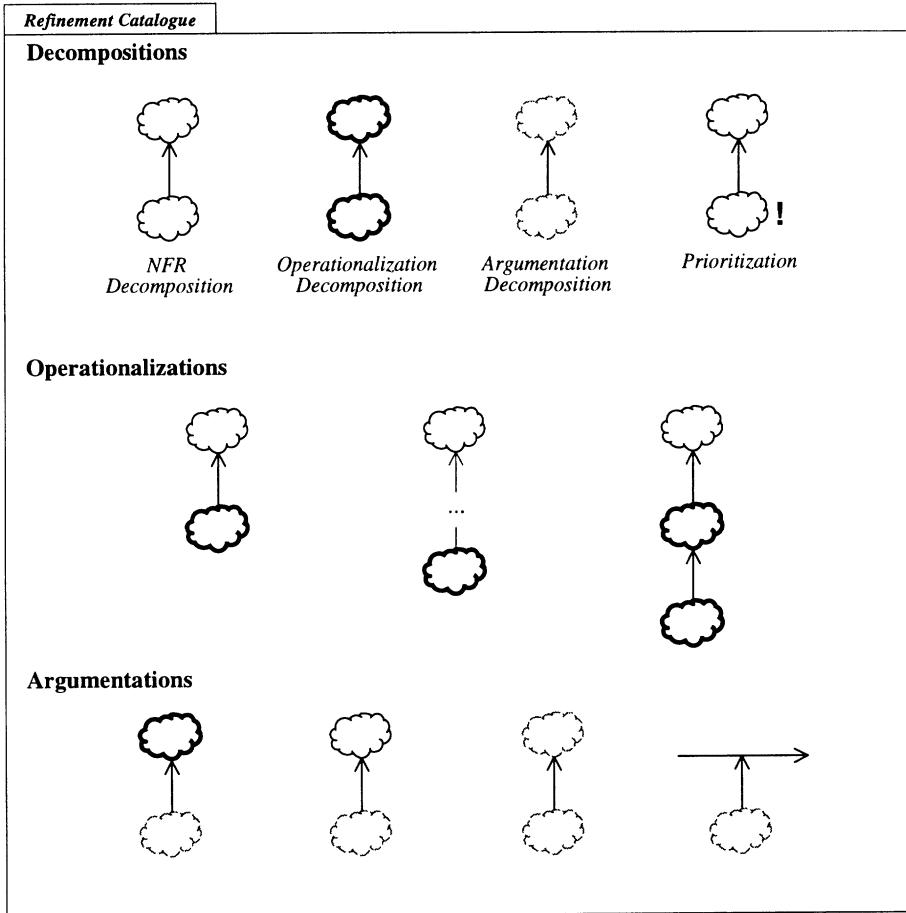
of softgoals: NFR softgoals (drawn as thin (light) clouds), operationalizing softgoals (thick (dark) clouds), and claims (dashed clouds).

Figure 3.5 shows several *NFR decompositions*, which refine NFR softgoals into other NFR softgoals. By breaking a “large” NFR softgoal into more specific ones, this kind of decomposition can help solve large problems in terms of smaller ones. Decomposition is also helpful for dealing with ambiguities, and dealing with priorities.

Operationalization decompositions refine operationalizing softgoals into other operationalizing softgoals. They are helpful for taking a general solution and breaking it (further operationalizing it) into more specific ones. Operationalization decompositions are illustrated in Chapter 4.

Argumentation decompositions refine claim softgoals into other claim softgoals. This is helpful for supporting or denying particular design rationale. Chapter 4 also illustrates argumentation decompositions.

Prioritization is a special kind of decomposition. It refines a softgoal into another softgoal with the same type and topics, but with an associated priority. For example, Figure 3.5 shows the prioritization of ResponseTime[GoldAccount.highSpending] to !ResponseTime

**Figure 3.6.** Kinds of Refinements.

`[GoldAccount.highSpending]{critical}`, and the prioritization of `!Accuracy` to `[GoldAccount.highSpending]{critical}`.

The prioritization can reflect the intention of the developer to spend more time and effort to satisfy these softgoals than the rest, and, as will be seen later, to use relative priority among softgoals to resolve conflicts and deal with priorities. In this book, prioritization is primarily applied to NFR softgoals.

Operationalization.

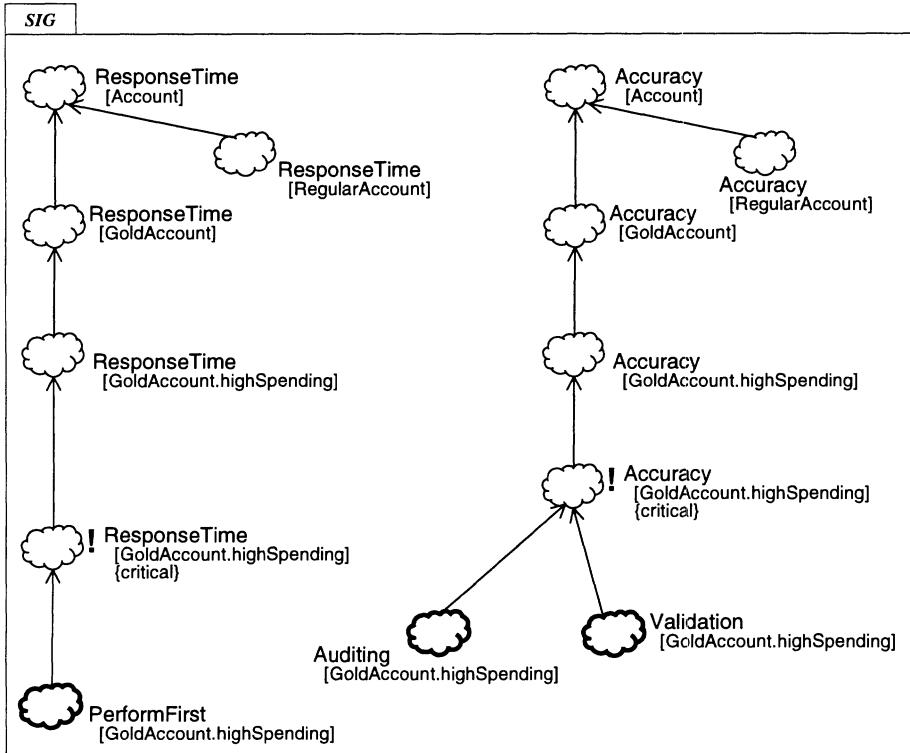


Figure 3.7. Refining NFR softgoals into operationalizing softgoals.

Operationalizations refine a softgoal into operationalizing softgoals.

The refinement of NFR softgoals into operationalizing softgoals represents a crucial transition, where we move from NFRs (which are to be achieved) to development techniques which can achieve the NFRs.

Figure 3.7 shows some operationalizations. `PerformFirst[GoldAccount.highSpending]` is an operationalization of `!ResponseTime[GoldAccount.highSpending]{critical}`. In addition, `Auditing[GoldAccount.highSpending]` and `Validation[GoldAccount.highSpending]` are operationalizations of `!Accuracy[GoldAccount.highSpending]{critical}`.

The term “operationalization” can also be used to refer to a *refinement* producing an operationalizing softgoal as an offspring. Observe that a solution can lead to a problem in its own right. A solution often needs to be “achieved” as well in terms of more concrete ones. Furthermore, there is usually more than one way to achieve a solution. By treating operationalizations as softgoals, we are able to aim to satifice operationalizing softgoals

and can refine them. For example, operationalization can refer to a refinement of `Validation[GoldAccount.highSpending]` to a specialized operationalizing softgoal, such as validation by a specific kind of employee. To avoid ambiguity, this may be called an “operationalization decomposition,” a “decomposition of an operationalization,” or a “further operationalization” of `Validation[GoldAccount.highSpending]`.

“Operationalization” can also be used to relate an operationalizing softgoal to one of its ancestors, typically an NFR softgoal somewhat “above” it in a SIG. For example, we can say that `PerformFirst[GoldAccount.highSpending]` operationalizes, or is an operationalization of, `ResponseTime[GoldAccount.highSpending]`. Likewise, `Validation[GoldAccount.highSpending]` is an operationalization of `Accuracy[GoldAccount.highSpending]`.

Argumentation.

Design rationale is noted via *argumentation*. This is recorded by refinements, typically involving claim softgoals. Examples are shown in Figure 3.8.

A few words of clarification are in order here. By treating design rationale as softgoals, the intention is to convey the idea that they in turn may need to be supported or denied by other claims, partially or fully, using the notion of “satisficing.” Typically, however, claims merely state facts or opinion about the domain, and developers usually do not attempt to achieve them the way they do for NFR softgoals or operationalizing softgoals. Thus refinements involving claim softgoals are somewhat different in nature than those involving NFR softgoals or operationalizing softgoals.

One form of argumentation has a claim softgoal as an “offspring” of any kind of *softgoal*. In Figure 3.8, `Claim[“Priority actions can be performed first.”]` is an offspring of the operationalizing softgoal `PerformFirst[GoldAccount.highSpending]`. The claim gives a reason for selecting the operationalization.

Another form of argumentation has a claim softgoal as an offspring of an *interdependency link*. In Figure 3.8, `Claim[“One of vital few: high spending in gold accounts.”]` is an offspring of the interdependency between `ResponseTime[GoldAccount.highSpending]` and `!ResponseTime[GoldAccount.highSpending]{critical}`. The claim is also an offspring of the interdependency link between `Accuracy[GoldAccount.highSpending]` and `!Accuracy[GoldAccount.highSpending]{critical}`. The claim gives a reason for making the refinements, here for prioritizations.

Claims can also be used to support other claims.

In the NFR Framework, other forms of argumentation are possible, albeit infrequent. One involves an interdependency link as the “offspring,” and a claim softgoal as the parent. The other involves interdependency links as both the parent and offspring. These two cases are not used in this book.

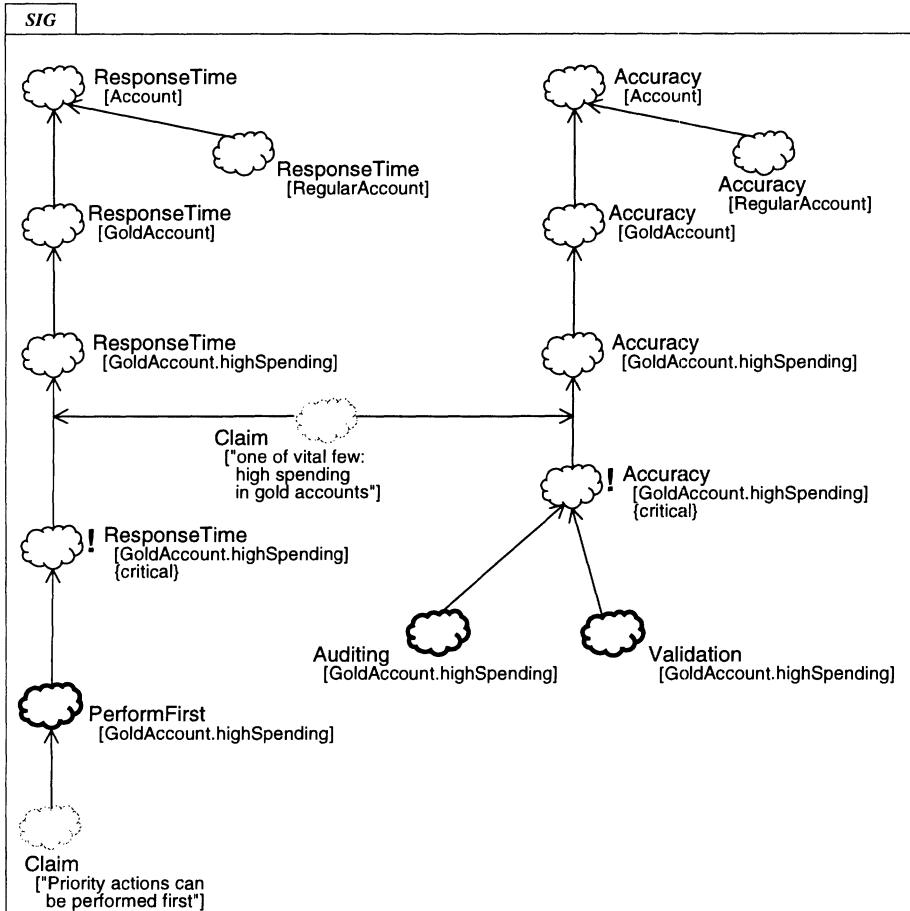


Figure 3.8. Recording design rationale using claim softgoals.

Prioritization.

Softgoal prioritization includes dealing with *criticality* and *dominance*. Critical softgoals have a high degree of importance, and dominant softgoals deal with frequent situations which dominate the workload of a system. Any softgoal or interdependency can be associated with a *priority*, indicating its relative importance. A priority softgoal can be thought of as one of the “vital few” [Juran79] softgoals, needed for success of the system, while non-priorities can be considered part of the “trivial many.” There are priority attributes:

critical and **dominant** (each with softgoal prefix “!”), **very critical** (“!!”) and **very dominant** (each with softgoal prefix “!!”), etc. The developer can prioritize a single softgoal or interdependency link, as well as assign relative importance to several softgoals or interdependencies.

In the example, `!Accuracy[GoldAccount.highSpending]{critical}` and `!ResponseTime[GoldAccount.highSpending]{critical}` are both critical. These prioritizations are supported by a claim softgoal. Prioritizations can be used to deal with tradeoffs and help with selection among alternative operationalizing softgoals.

Another use of prioritization concerns coping with resource limitations. Suppose developers are given a large number of softgoals. Should they put equal amount of effort into meeting each of them? Perhaps not, especially if they only have a limited amount of time available. But instead, they would want to prioritize the softgoals and spend more time for softgoals of high priority. For example, they may choose to spend most of the time to meet the priority softgoals `!Accuracy[GoldAccount.highSpending]{critical}` and `!ResponseTime[GoldAccount.highSpending]{critical}`, rather than trying to meet non-priority accuracy and response-time softgoals for regular accounts.

Up to this point the figures have left unspecified the contributions that offspring softgoals make to the satisficing of parent softgoals. This is the topic of the next section.

Types of Contributions

As illustrated in Chapter 2, development proceeds by repeatedly refining “parent” softgoals into “offspring” softgoals. In such refinements, the offspring can contribute fully or partially, and positively or negatively, towards the satisficing of the parent.

Recall that we speak of softgoal *satisficing* to suggest that generated software is expected to satisfy non-functional requirements within acceptable limits, rather than absolutely. Accordingly, in the NFR Framework, there can be several different types of *contributions* describing how the satisficing of the offspring (or failure thereof) contributes to the satisficing of the parent.

For example, *AND* is one of the contribution types. It means that if the offspring are all satisfied, so will their parent be. A more precise definition will be presented shortly. Suppose `Accuracy[Account]` has two offspring `Accuracy[RegularAccount]` and `Accuracy[GoldAccount]` which make an *AND* contribution to their parent.

Figure 3.9 illustrates this contribution in a softgoal interdependency graph. Offspring are shown underneath the parent softgoal. The *AND* contribution is shown by an arc connecting the lines from the offspring to the parent. For clarity of figures, arrowheads from offspring to parents are omitted from *AND* and *OR* contributions. Note that previous figures showed interdependencies between the parent and offspring, but did not show an *AND* contribution.

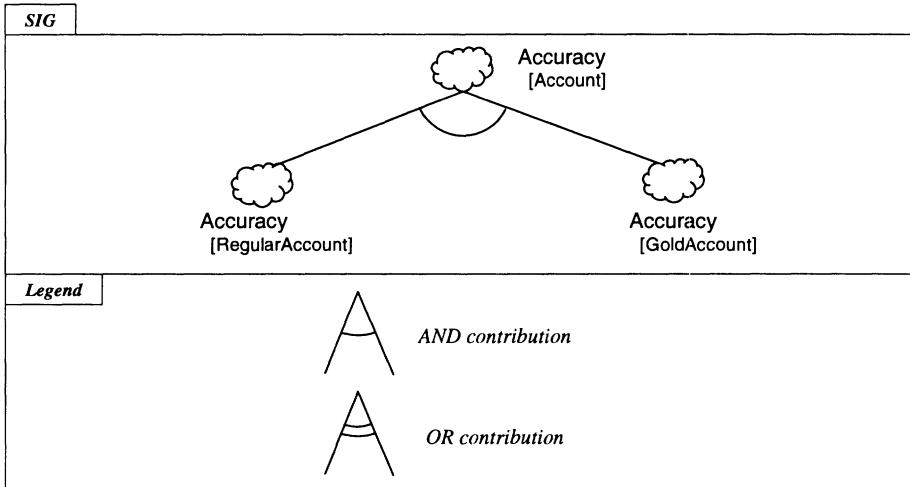


Figure 3.9. A decomposition with an *AND* contribution.

We can express the *AND* contribution of Figure 3.9 by writing:

Accuracy[RegularAccount] *AND* Accuracy[GoldAccount]
SATISFICE Accuracy[Account]

to state that if all offspring are satisfied, then the parent will be satisfied. Equivalently, this can also be written as:

AND({Accuracy[RegularAccount], Accuracy[GoldAccount]})
SATISFICE Accuracy[Account]

which shows the contribution of the *set* of offspring to the parent.

Figure 3.10 shows examples of several contribution types. Note that contributions were not shown in Figure 3.8 and earlier SIGs in this chapter.

For *OR* contributions, we have:

Accuracy[MasterFile] *OR* Accuracy[WorkingFile]
SATISFICES Accuracy[RegularAccount]

which states that if any offspring is satisfied, then the parent will be satisfied. This can also be written as:

OR({Accuracy[MasterFile], Accuracy[WorkingFile]})
SATISFICES Accuracy[RegularAccount]

Contribution Type Examples		Examples of contribution types in use
Contribution Types		
 AND	 OR	<p>Performance [RegularAccount] → Performance [Account]</p> <p>Accuracy [RegularAccount] → Accuracy [WorkingFile]</p>
 ++  --		<p>Validation [GoldAccount.highSpending] → Accuracy [GoldAccount.highSpending] {critical}</p> <p>Validation [GoldAccount.highSpending] → FlexibleUserInterface [InfrequentUser, GoldAccount.highSpending]</p>
 +  -		<p>Indexing [GoldAccount.highSpending] → ResponseTime [GoldAccount.highSpending]</p> <p>Indexing [GoldAccount.highSpending] → FlexibleUserInterface [InfrequentUser, GoldAccount.highSpending]</p>
 SOME+  SOME-		<p>Auditing [GoldAccount.highSpending] → Accuracy [GoldAccount.highSpending] {critical}</p> <p>CompressedFormat [GoldAccount.highSpending] → ResponseTime [GoldAccount.highSpending] {critical}</p>

Figure 3.10. Examples of several contribution types.

For *OR* contributions, a double arc connects the lines from the group of offspring to the parent.

A given contribution may, however, not be acceptable to different people. Thus, just like softgoals, contributions need to be satisfied either through a refinement process or through arguments provided by the developer. In this book, contributions are normally assumed to be satisfied, unless indicated otherwise.

AND and *OR* relate a *group* of offspring to a parent. Let's consider the other contribution types, which relate a *single* offspring to a parent.

One pair of contribution types provide *sufficient* (full) support, *MAKES* being positive, and *BREAKS*, being negative. *MAKES* provides sufficient positive support. It can be viewed as the single-offspring analogy of *AND*. With *MAKES*, if the one offspring is satisfied, the parent can be satisfied. *BREAKS* provides sufficient negative support: if the one offspring is satisfied, the parent can be denied.

Another pair of contribution types provide *partial* support, *HELPS* being positive and *HURTS*, being negative. *HELPS* provides partial positive support: if the one offspring is satisfied, partial positive support is given to the parent. For *HURTS*, if the one offspring is satisfied, partial negative support is given to the parent.

Note that we can write contributions fairly naturally in an infix form, e.g., “Validation *MAKES* Accuracy” and “Indexing *HELPS* ResponseTime”. Reading left-to-right, we have the offspring, the contribution, and the parent. Where unambiguous, softgoal topics may be omitted from such statements of contributions.

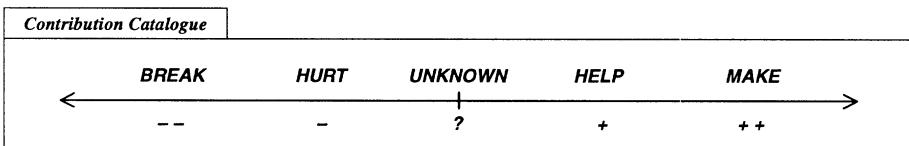


Figure 3.11. Intuitive distinction among contribution types.

In addition, the *UNKNOWN* contribution makes some contribution, but of unknown “sign” (positive or negative) and “extent” (partial or sufficient).

Figure 3.11 illustrates these *contribution types* in a spectrum, ranging from sufficient negative support (“--”) on the left to sufficient positive support (“++”) on the right. Note that the suffix “*S*” may be omitted from the names of contribution types *MAKES*, *BREAKS*, *HELPS*, *HURTS* and *EQUALS*, and the keyword *SATISFICES*.

MAKE represents the positive situation where the developer is sufficiently confident to think that the particular offspring is “good enough” towards meeting (“making”) the parent. On the other hand, when the developer is confident that the offspring will deny (“break”) the meeting of the parent softgoal, *BREAK* is used. Sufficient contributions are shown with doubled sign symbols, while the partial contributions, *HELP* (“+”) and *HURT* (“-”) are shown with single signs, towards the middle. In the middle is *UNKNOWN*, drawn as “?”.

Figure 3.12 is an elaboration of Figure 3.11. The *SOME+* contribution type represents *some positive* contribution, either *HELP* or *MAKE*. Likewise *SOME-* represents *some negative* contribution, either *HURT* or *BREAK*.

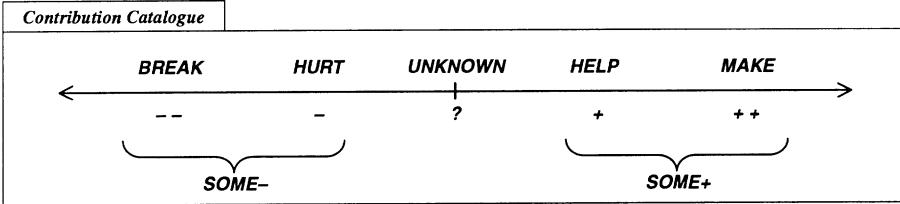


Figure 3.12. Grouping positive and negative contribution types.

Note that the informal contribution types of Chapter 2 have given way to a larger number of contribution types with more specific meanings. For example, in Chapter 2, “+” stood for all positive contributions. Now we have several types of positive contributions: “+”, “++”, and *SOME+*.

When the developer considers that the achievement of a softgoal is satisfactory, the softgoal is *satisficed*. However, when softgoal achievement is considered unsatisfactory, the softgoal is *denied*.

If achievement of a softgoal is considered to be potentially satisfactory, the softgoal is *satisficeable*. If, however, softgoal achievement is considered to be potentially unsatisfactory, the softgoal is *deniable*.

In Artificial Intelligence problem-solving terminology [Nilsson71], *satisficed* refers to “solvable” problems, while *denied* refers to “unsolvable” ones. In this sense, *satisficeable* refers to potentially solvable problems, and *deniable* refers to potentially unsolvable ones.

Suppose an offspring makes a sufficiently positive contribution to a parent. We say that the parent is *satisficeable*. Furthermore, if there is no counter-evidence, the parent is *satisficed*. However, if there is counter-evidence, the parent can become *unsatisficeable* (but not necessarily *denied*).

Likewise, if an offspring makes a sufficiently negative contribution to a parent, the parent is *deniable*. In the absence of counter-evidence, the parent is *denied*. With counter-evidence, the parent can become *undeniable* (but not necessarily *satisficed*).

Now what does it mean to say that a softgoal is *satisficeable* or *deniable*? Why can't a softgoal simply be considered either *satisficed* (satisfactory) or *denied* (unsatisfactory)?

The reason is that a softgoal will sometimes receive both positive and negative contributions from offspring.

For instance, the accuracy softgoal for high spending data may be *satisficed* — thanks to a validation procedure which, although imperfect, significantly enhances the accuracy of such data. However, the accuracy softgoal can also be *denied* later on — due to a flexible user interface which permits widespread access to infrequent users. On the other hand, performing opera-

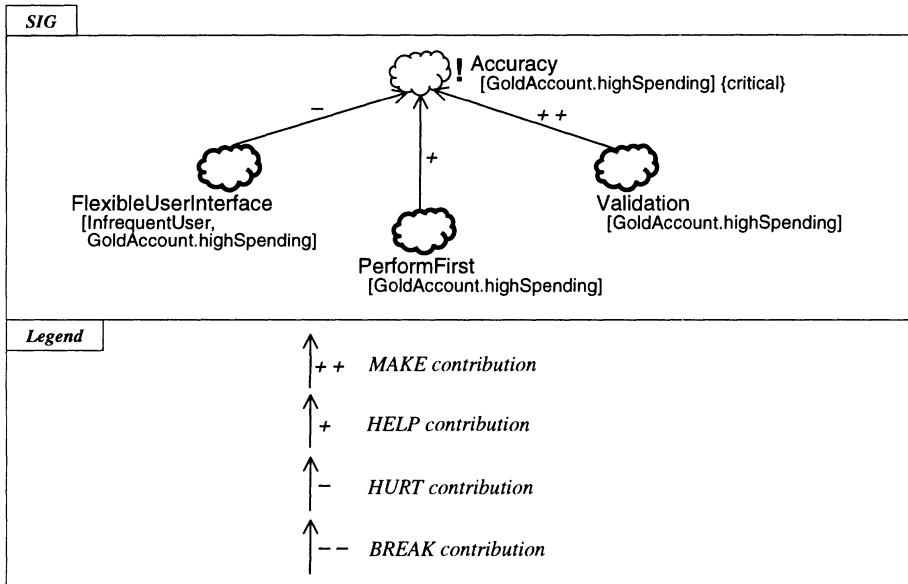


Figure 3.13. A conflict in contributions.

tions in a certain order may aid accuracy. These conflicting contributions are shown in Figure 3.13.

When a validation procedure is applied to the accuracy softgoal, the accuracy softgoal *may* be satisfied, i.e., satisficeable. In other words, the softgoal will remain satisfied if no other operationalizing softgoal with a negative effect is applied later on. Otherwise, the accuracy softgoal may no longer be considered satisfied or it may even become unsatisfied.

Thus, we need to distinguish a softgoal being *satisfied* or *denied* from one being *potentially* satisfied or denied, namely, *satisficeable* or *deniable*. Likewise, *interdependencies* can be *satisfied*, *denied*, *satisficeable* and *deniable*.

With these distinction in place, we now define the set of contribution types.

Definitions of Contribution Types.

The first two contribution types, *AND* and *OR*, relate a group of offspring to a parent (See Figure 3.9).

Let

$$\{ \text{offspring} \} = \{ \text{offspring}_1 \dots \text{offspring}_n \}$$

and note that variables are shown in sans serif italics.

We then let

$$offspring_1 \text{ AND } \dots \text{ AND } offspring_n \text{ SATISFICE } parent$$

be equivalent to:

$$\text{AND}(\{offspring\}) \text{ SATISFICE } parent$$

We define them both as follows:

If all the offspring are satisfied
when the interdependency itself is satisfied,
then the parent is satisficeable;
and
if any of the offspring is denied
when the interdependency itself is satisfied,
then the parent is deniable.

In other words, $offspring_1 \text{ AND } \dots \text{ AND } offspring_n \text{ SATISFICE } parent$ implies that the parent is satisfied if all the offspring are satisfied, and the offspring make an *AND* contribution to the parent. In our example, *Accuracy[Account]* will be satisficeable if all the offspring (here, *Accuracy[RegularAccount]* and *Accuracy[GoldAccount]*) are satisfied, and the *AND* interdependency itself is satisfied.

But what does it mean to say “the offspring make an *AND* contribution to the parent” and “the *AND* interdependency itself is satisfied”? A particular contribution type such as *AND* may sometimes need to be subject to validation, especially if the developer does not have confidence in it. After all, it may turn out that the offspring do not conjunctively satisfy the parent, but instead disjunctively or individually satisfy the parent. Thus, for an interdependency to be effective, it too needs to be satisfied.

Contributions can also be written in a frame-like notation. For example,

$$\begin{aligned} & \text{Accuracy[RegularAccount] AND Accuracy[GoldAccount]} \\ & \text{SATISFICE Accuracy[Account]} \end{aligned}$$

can be written as:

```
Contribution specialized-account-accuracy-to-account-accuracy
parent: Accuracy[Account]
offspring: Accuracy[RegularAccount],
           Accuracy[GoldAccount]
contribution: AND
```

If desired, the kind of softgoal can be indicated, e.g., *parent: NFR Softgoal Accuracy[Account]*.

As for *OR* contributions, the parent is satisfied if any of the offspring are satisfied and the offspring make an *OR* contribution to the parent.

We then let:

offspring₁ OR ... OR offspring_n SATISFICES parent

be equivalent to:

OR({offspring}) SATISFICES parent

We define them both as follows:

If any of the offspring is satisfied
when the interdependency itself is satisfied,
then the parent is satisficeable;
and
if all of the offspring are denied
when the interdependency itself is satisfied,
then the parent is deniable.

Let's turn to the contributions involving a single offspring.

MAKES and *BREAKS* provide sufficient support, *MAKES* being positive and *BREAKS* being negative.

offspring MAKES parent is defined as:

If the offspring is satisfied
when the interdependency itself is satisfied,
then the parent is satisficeable.

offspring BREAKS parent is defined as:

If the offspring is satisfied
when the interdependency itself is satisfied,
then the parent is deniable.

In Figure 3.10, the validation operationalization is sufficient to satisfy an accuracy softgoal, so we have *Validation[GoldAccount.highSpending] MAKES !Accuracy[GoldAccount.highSpending]{critical}*. On the other hand, a claim that

accuracy is more important than flexibility is sufficient to deny an operationalization using a flexible user interface. So we have Claim[“Accuracy more important than flexibility.”] BREAKS FlexibleUserInterface[InrequentUser,GoldAccount.highSpending].

HELPS and *HURTS* provide partial support, *HELPS* being positive, and *HURTS* being negative.

HURTS provides partial negative support. If *offspring HURTS parent* then denial of the offspring leads to the satisficing of the parent, and satisficing of the offspring contributes to the denial of the parent.

offspring HURTS parent is defined as:

If the offspring is denied
when the interdependency itself is satisficed,
then the parent is satisficeable.

HELPS provides partial positive support.

offspring HELPS parent is defined as:

If the offspring is denied
when the interdependency itself is satisficed,
then the parent is deniable.

As a consequence of *offspring₁ HELPS parent*, there are other offspring, *offspring₂*, ..., *offspring_n* which cannot achieve the satisficing of the parent without the contribution of *offspring₁*.

On the other hand, *offspring₁* is not the only partial contributor to the parent. In other words, besides *offspring₁*, there can be other offspring *offspring_{2'}*, ..., *offspring_{m'}* which also partially contribute to the parent. For example, in Figure 3.10, Indexing[GoldAccount.highSpending] HELPS ResponseTime[GoldAccount.highSpending], but additional operationalizations may be used to help satisfice the ResponseTime softgoal.

Another pair of contribution types are used when the “sign” of a contribution (i.e., positive or negative) is known, but the extent (i.e., partial or full) of support is not. When there is *some* positive contribution, but some uncertainty whether to use *HELP* or *MAKE*, the *SOME+* contribution type is used. Similarly, *SOME-* is used where there is some negative contribution, but some uncertainty whether to use *HURT* or *BREAK*. Figure 3.12 shows *SOME+* grouping the *HELP* and *MAKE* cases, while *SOME-* groups *HURT* and *BREAK*. *SOME+* is defined as follows:

offspring SOME+ parent \equiv *offspring HELPS parent or*
offspring MAKES parent

Now “*offspring SOME+ parent*” can be read as “the offspring makes *some positive* contribution to the parent.” *SOME-* is defined as:

$$\text{offspring } \text{SOME- parent} \equiv \text{offspring HURTS parent or} \\ \text{offspring BREAKS parent}$$

Here “*offspring SOME- parent*” can be read as “the offspring makes *some negative* contribution to the parent.”

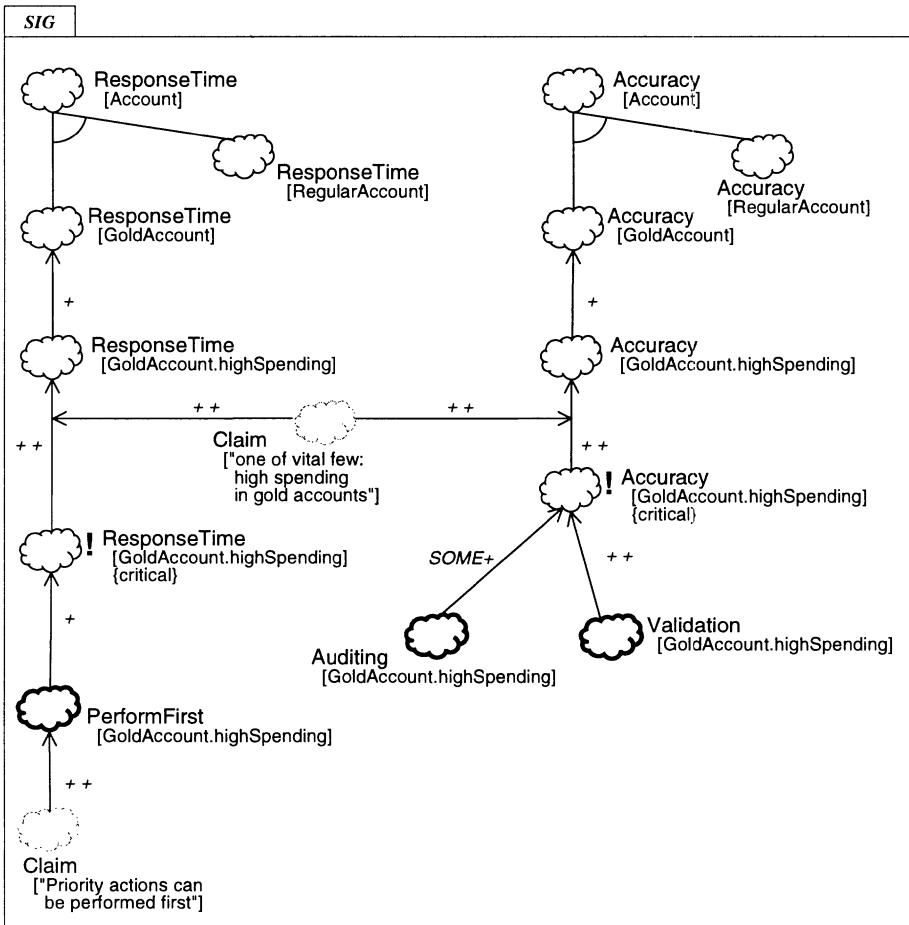


Figure 3.14. A softgoal interdependency graph with various contribution types.

In Figure 3.10, auditing gives *some positive* contribution to accuracy. In addition, the use of a compressed format gives *some negative* contribution

to response time (since extra time will be needed to uncompress data before usage).

It is useful to define the *EQUALS* (equivalent) contribution type. *EQUALS* means that the offspring is satisficeable if and only if the parent is, and the offspring is deniable if and only if its parent is too.

At times, it may be hard to determine *a priori* the precise relationship between a set of offspring and their parent softgoal without further expansion of the softgoal interdependency graph. This situation is accommodated through the *UNKNOWN* (“?”) contribution type. *UNKNOWN* presents some unknown interdependency between the parent and the offspring. More precisely:

$$\begin{aligned} \text{offspring UNKNOWN parent} &\equiv \text{offspring SOME- parent or} \\ &\quad \text{offspring EQUALS parent or} \\ &\quad \text{offspring SOME+ parent} \end{aligned}$$

Figure 3.14 is an elaboration of Figure 3.8, showing contribution types. Note that it shows *AND* contributions which were shown as pairs of interdependencies in Figures 3.5, 3.7 and 3.8. However, Figure 3.14 is still incomplete as it is missing labels which indicate the degree to which softgoals (and interdependencies) are satisficed. This is the topic of the next section.

3.3 THE EVALUATION PROCEDURE

Presented up to this point are two of the five main concepts of the NFR Framework: softgoals and interdependencies. These concepts enable developers to treat non-functional requirements as softgoals, and then satisfice them by repeatedly refining them into more specific ones. In developing a system, developers need to determine whether softgoals in a softgoal interdependency graph are satisficed, be they NFR softgoals, operationalizing softgoals or claim softgoals. Ultimately, they need to know if the initial (top) NFR softgoals for the system are met. *But how do they do that?*

The *evaluation procedure* is another concept of the NFR Framework which determines the degree to which non-functional requirements are achieved by a set of decisions. Given a softgoal interdependency graph, complete or incomplete, the evaluation procedure determines whether each softgoal or interdependency in the softgoal interdependency graph is satisficed. This is done through the assignment of a *label*.

Decisions to accept or reject alternatives provide an initial set of labels. These decisions are often “leaf” or bottom nodes of a softgoal interdependency graph. The evaluation procedure uses these labels to determine the impact of decisions on other softgoals, and ultimately, upon the top softgoals.

Thus, the evaluation procedure is useful in selecting among alternatives. In the presence of competing alternatives, developers can use the procedure to see what impact a particular selection has on the satisficing of their softgoals. If one selection hurts some important softgoals, developers can simply reject it, choose another and again use the procedure to see the impact of the new

choice. By repeating this process, developers can aim at making a choice which yields the most benefit with acceptable minimum sacrifices.

Using the notions of *satisficeable* and *deniable* from the previous section, a softgoal or interdependency of the graph is labelled:

- *satisficed* (\checkmark or S) if it is satisficeable and not deniable;
- *denied* (\times or D) if it is deniable but not satisficeable;
- *conflicting* (shown as a thunderbolt in figures, and “ \natural ” or C in the text) if it is both satisficeable and deniable; and
- *undetermined* (denoted by blank or U) if it is neither satisficeable nor deniable.

The U label represents situations where there is no positive or negative support. At the time a softgoal is newly introduced, it is given a U label by default. Interdependencies are normally given a satisficed (“ \checkmark ”) label when introduced. In figures, any unlabelled softgoal is undetermined (“U”) by default, and any unlabelled interdependency is satisficed (“ \checkmark ”) by default.

When a softgoal or interdependency in a softgoal interdependency graph is assigned a new label (including default values), the evaluation procedure is activated and propagates labels from offspring to parent. It consists of two basic steps. For each softgoal or interdependency in a SIG, the procedure first computes the *individual impact* of each satisficed interdependency. Secondly, the individual impacts of all interdependencies are combined into a single label. Details of the procedure will be described shortly.

The NFR Framework adopts a dialectical style of reasoning in which arguments are made to support or deny why non-functional requirements are considered fulfilled. This leads to a premise of the Framework that only some of the relevant knowledge is formally represented. The rest remains with developers, during the process of dealing with non-functional requirements.

Given this open-ended and argumentative nature of the development process, the NFR Framework calls for an *interactive* evaluation procedure where the developer may be asked to step in and determine the appropriate label for a particular softgoal or interdependency having supporting but inconclusive (partial) evidence.

For this reason, the labels characterizing the influence of one set of offspring towards a parent includes \checkmark , \times , \natural , and U, as mentioned before, but also:

- *weak positive* (W^+) representing inconclusive positive support for a parent, and
- *weak negative* (W^-) representing inconclusive negative support for a parent.

For any conflict (\natural) from the first step, the developer can determine the label that characterizes the contribution of an offspring towards its parent.

Figure 3.15 shows a *catalogue of label values*. The spectrum ranges from denied to satisficed. Unknown and conflicting values are shown in the middle.

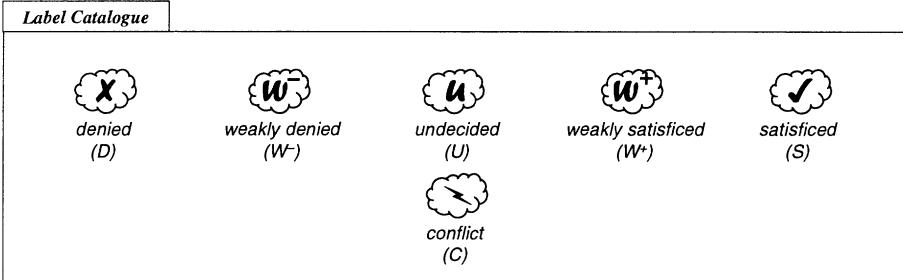


Figure 3.15. Catalogue of label values.

The First Step

The First Step determines the “individual impact” of an offspring’s contribution towards a parent.

For *AND* and *OR* contributions, we treat all of the offspring as *one* group, with a single “individual impact,” defined as follows:

If $offspring_1 \text{ AND } \dots \text{ AND } offspring_n \text{ SATISFICE parent}$
then $\text{label}(\text{parent}) = \min_i(\text{label}(offspring_i))$

If $offspring_1 \text{ OR } \dots \text{ OR } offspring_n \text{ SATISFICES parent}$
then $\text{label}(\text{parent}) = \max_i(\text{label}(offspring_i))$

where the labels are ordered in increasing order as follows:

$$\times \leq U \approx \nexists \leq \checkmark$$

The developer needs to provide input if either of U or \nexists is involved. For the purposes of the First Step, there is no specific ordering relationship between U and \nexists ; instead they are both in-between \times and \checkmark .

Let’s consider the individual impact for some of the other contribution types. Here are some rules for propagation of labels from an offspring to its parent:

- *MAKES* propagates \checkmark from offspring to parent; it also propagates \times .
- *BREAKS* “inverts” the “sign” of an offspring’s \checkmark label into \times for the parent.
- *HELPS* keeps the same direction, but “weakens” it. That is, \checkmark in the offspring is weakly propagated to W^+ in the parent, and \times in the offspring is weakly propagated to W^- in the parent.

- *HURTS* inverts the direction, and weakens it. That is, \checkmark in the offspring is weakly inverted to W^- in the parent, and \times in the offspring is weakly inverted to W^+ in the parent.
- *SOME+* considers the two parent labels that would result from using *HELP* and *MAKE* contributions for a given offspring, and uses the parent label that is weaker. By “weaker,” we mean that if the two parent labels are \checkmark and W^+ , the result is W^+ ; if the parent labels are \times and W^- , the result is W^- . Looking at it another way, *SOME+* labels the parent with a “weak” label (W^+ or W^-) with the same “direction” as the offspring’s label. An offspring label of \checkmark or W^+ results in W^+ for the parent; an offspring label of \times or W^- results in W^- in the parent.
- *SOME-* considers the two parent labels that would result from using *HURT* and *BREAK* contributions for a given offspring, and uses the parent label that is “weaker,” as explained above. Looking at it another way, *SOME-* labels the parent with a “weak” label that inverts the “direction” of the offspring’s label. An offspring label of \checkmark or W^+ results in W^- for the parent; an offspring label of \times or W^- results in W^+ in the parent.
- *UNKNOWN* (“?”) always contributes U .

In addition, there are rules for special offspring values:

- An undetermined offspring (U) always propagates U , regardless of the contribution type.
- A conflicting offspring (\ddagger) propagates \ddagger , unless the contribution type is *UNKNOWN*.

Let’s present the individual impact upon a parent, of a given offspring along a given contribution type. The entries in Tables 3.1 and 3.2 show the individual impact upon the *parent*.

Table 3.1 outlines an *evaluation catalogue* of individual impacts given selected contribution types, representing the “signs” of the contributions. Parent labels are shown in the table entries. As an example, if the offspring is satisfied (“ \checkmark ”), and the contribution type is *SOME-*, the individual impact on the parent is weak negative (W^-). To condense the size of Table 3.1, it uses grouped contribution types, *SOME+* and *SOME-*.

It is interesting to compare rows of entries of Table 3.1 with the label catalogue of Figure 3.15, which has a left-to-right ordering. In Table 3.1, looking left-to-right at the individual impacts for satisfied offspring (labelled \checkmark), there is a progression left-to-right through the label catalogue. For individual impacts of denied offspring (labelled \times), there is a progression *right-to-left* through the label catalogue.

We can now consider the individual impacts of more contribution types, in an extension of Table 3.1. Table 3.2 includes entries for contribution types *MAKES*, *BREAKS*, *HELPS* and *HURTS*.

<i>Evaluation Catalogue</i>				
<i>Individual Impact of offspring with label:</i>	<i>upon parent label, given offspring-parent contribution type:</i>			
	SOME-	?	(UNKNOWN)	SOME+
✗ (<i>Denied</i>)	W ⁺	U		W ⁻
✗ (<i>Conflict</i>)	✗	U		✗
U (<i>Undetermined</i>)	U	U		U
✓ (<i>Satisficed</i>)	W ⁻	U		W ⁺

Table 3.1. The “individual impact” of an offspring upon its parent for selected contribution types during the First Step. Parent labels are shown in the table entries.

<i>Evaluation Catalogue</i>								
<i>Individual Impact of offspring with label:</i>	<i>upon parent label, given offspring-parent contribution type:</i>							
	BREAK	SOME-	HURT	?	HELP	SOME+	MAKE	=
✗	W ⁺	W ⁺	W ⁺	U	W ⁻	W ⁻	✗	✗
✗	✗	✗	✗	U	✗	✗	✗	✗
U	U	U	U	U	U	U	U	U
✓	✗	W ⁻	W ⁻	U	W ⁺	W ⁺	✓	✓

Table 3.2. The “individual impact” of an offspring upon its parent during the First Step. Parent labels are shown in the table entries.

Let’s look again left-to-right at the individual impacts for offspring labelled ✓, but now in the larger Table 3.2. There is a progression left-to-right through the label catalogue of Figure 3.15. For individual impacts offspring labelled ✗ (denies), there is a progression right-to-left through the label catalogue.

In Table 3.2, it is interesting to note that the entries for the *SOME-* and *HURT* contributions are the same. Likewise, *SOME+* and *HELP* have identical entries.

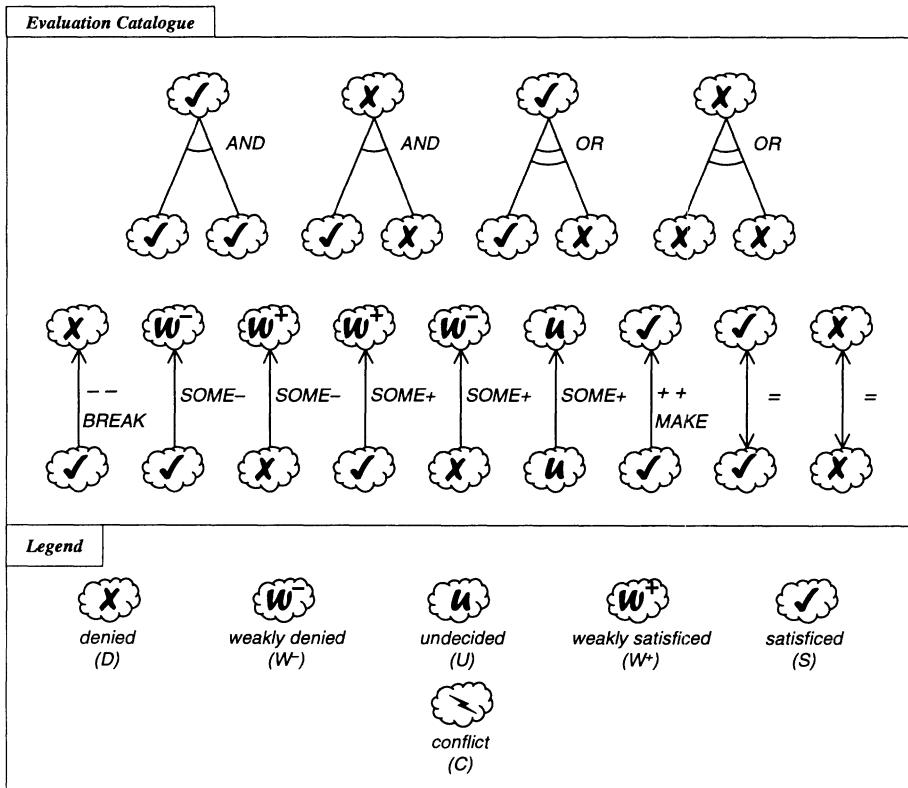


Figure 3.16. Label propagation for selected contribution types and offspring labels during the First Step.

The reader may note that the table does not have entries for other offspring, such as W^+ and W^- . As we will see later, the Second Step eliminates such values. We will also discuss a possible extension to retain such values as outputs of the Second Step, and inputs to the First Step.

Figure 3.16 illustrates how the first step of the evaluation procedure works on some selected contribution types and labels.

Consider the case of a denied offspring ("✗") and a *BREAKS* contribution type. It contributes W^+ to the parent. By reasons of symmetry, one might expect the entry to be ✓. Informally, the reason for using W^+ is that stopping a bad thing is helpful, but does not necessarily result in a good thing. In some cases, however, stopping a bad thing does result in a good thing. Thus, the developer may change the W^+ to ✓, depending on the circumstances. In

fact, this will be done frequently in subsequent chapters, and will not always be explicitly noted.

The Second Step

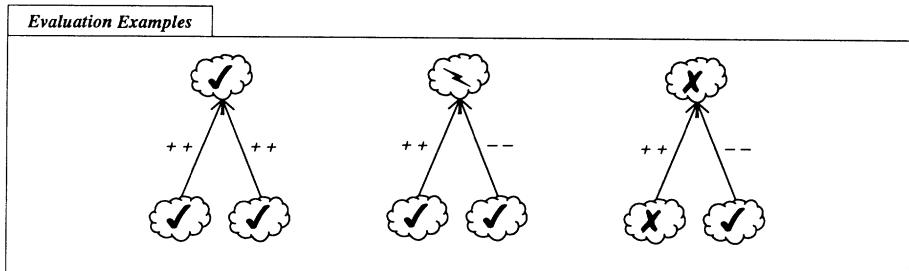


Figure 3.17. Examples of “automatic” label propagation during the Second Step.

Once all contributed labels have been collected for a given parent, the Second Step of the evaluation procedure combines them into a single label.

The labels contributed to a parent are collected in a “bag” (a collection which allows duplicate entries, unlike a set). The possible label values in the bag are \times , W^- , \natural , U , W^+ and \checkmark . A bag is used because duplicate labels are useful; for instance several positive supporting contributions indicated by several W^+ labels may be combined into a \checkmark label by the developer.

The W^+ and W^- labels in the bag are first combined by the developer into one or more \checkmark , \times , \natural , and U labels. Typically, W^+ values alone in a bag would result in \checkmark or U , and W^- values alone would result in \times or U . A mixture of W^+ and W^- values would typically result in \checkmark , \times or \natural .

The resulting set of labels is then combined into a single one, by choosing the minimal label of the bag, with a label ordering:

$$\natural \leq U \leq \times \approx \checkmark$$

If there are both \checkmark and \times values present, without *SOME+* or *SOME-*, the result will be a conflict (“ \natural ”), and the developer will have to deal with the conflict. For the purposes of the Second Step, there is no specific ordering relationship between \times and \checkmark ; instead they are both greater than the other label values.

Figures 3.17 and 3.18 illustrate the Second Step of evaluation. Note that these examples each show two individual contributions, *not* one *AND* contribution.

In the Second Step of evaluation, Figure 3.17 shows cases where the decisions can be made *automatically*, by following a procedure. This can be

done without developer intervention since no uncertainty is introduced during either the first or second steps. A straightforward application of the rules in the first and second steps suffices for automatically determining the label of the parent. In the first example, both individual contributions are satisfied (“ \checkmark ”), so the parent is also satisfied. In the second example, the left offspring contributes \checkmark , while the right one contributes \times , resulting in a conflict. If desired, the developer can resolve the conflict and change the parent’s label. In the third example, both offspring contribute \times , so the parent is also denied.

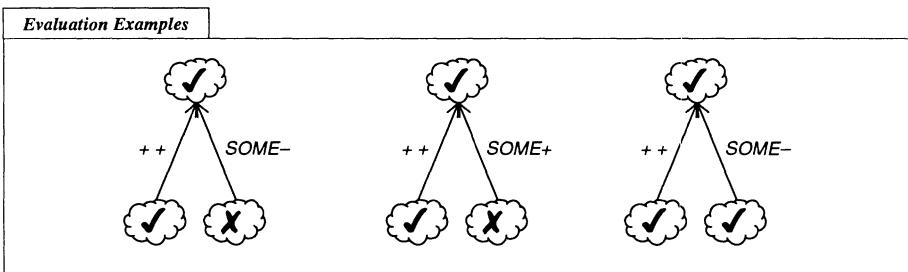


Figure 3.18. Examples of developer-directed label propagation during the Second Step.

The developer can use expertise about NFRs, the domain and development, as well as other knowledge, in order to resolve conflicts. Figure 3.18 illustrates the second step of evaluation, where the developer has stepped in to make some decisions. In each of the three examples, the left offspring is satisfied (“ \checkmark ”) and has a *MAKE* contribution to the parent. The right offspring in the first example contributes W^+ , which the developer must change to one of \times , \perp , U or \checkmark . If the developer chooses \checkmark , then the parent will be satisfied, receiving \checkmark from both offspring. The right offspring in the second example contributes W^- ; using expertise (e.g., that the left offspring is more important than the right one), the developer might change the right offspring label to \checkmark , resulting in the parent being satisfied. The third example is similar to the second one.

Generally, the interdependency is assumed satisfied, i.e., having \checkmark as its label. If the interdependency is not satisfied, the associated offspring has no effect on its parent.

The examples of Figure 3.18 show that developer input is important to the evaluation procedure. The developer combines the contributions into a single result. This “resolution” is done by choosing one of \times , \perp , U or \checkmark . However, after the Second Step is done, information about the resolution is not directly visible. Nonetheless, the developer may wish to take this resolution into account during subsequent evaluation, say of the parent’s parents. This may be done, for example, by changing the parent’s label to one of the weak label

values, W^+ or W^- . For instance, in each of the three examples of Figure 3.18, the parent's label could be changed from \checkmark to W^+ to indicate that there is some weak positive contribution arising from the combined offspring. In fact, this kind of change of labels is often done throughout this book, resulting in W^+ and W^- appearing throughout SIGs. Because of its frequency, this kind of change is not always explicitly mentioned.

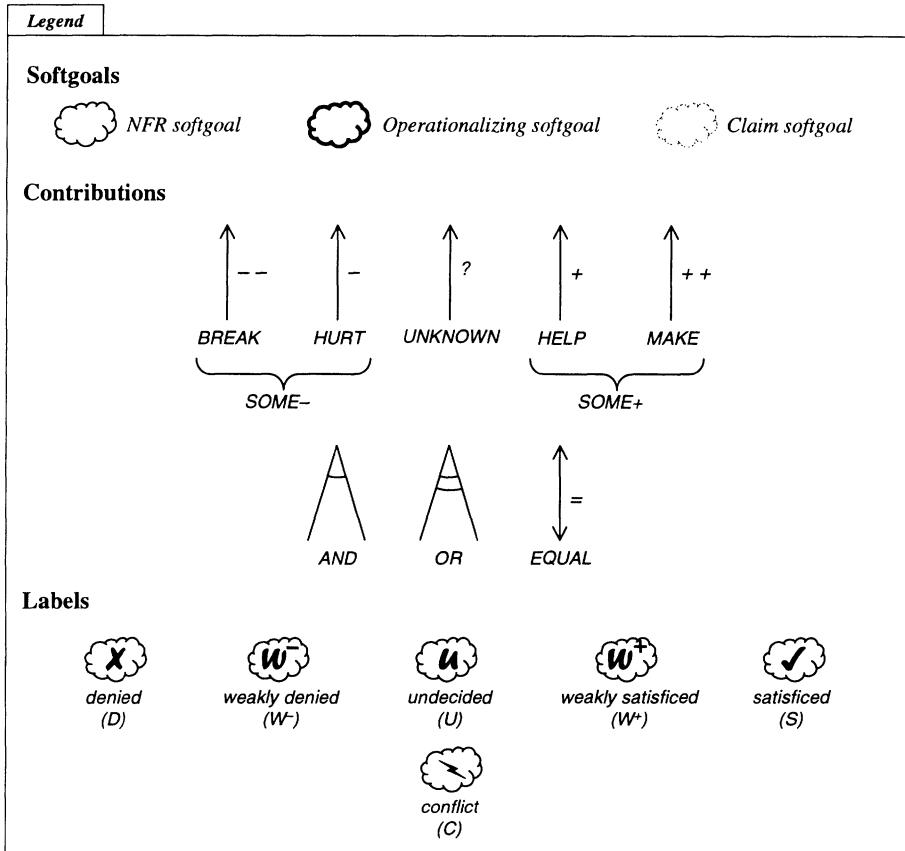


Figure 3.19. Meaning of symbols in softgoal interdependency graphs.

By having W^+ and W^- as output values after the Second Step, they would also have to be considered as inputs to subsequent First Steps. This would require extensions to Tables 3.1 and 3.2. Developers would be guided by domain information, and existing rules; for example, to infer possible propagation values for W^+ as offspring, they could look at entries for \checkmark as offspring,

but weaken the result. Informally, we can consider some possible propagation values:

- W^+ EQUALS W^+ .
- W^- EQUALS W^- .
- $W^+ OR W^+$ could result in W^+ for the parent.
- $W^- AND$ any value could result in denying the parent
- A W^+ offspring and a *SOME+* contribution could result in W^+ for the parent.
- A W^+ offspring and a *SOME-* contribution could result in W^- for the parent.

To summarize the notation, Figure 3.19 presents the symbols used for softgoals, contributions and labels.

We now continue with SIGs for our example. Figure 3.20 shows some of the decisions made by the developer to choose or reject some operationalizing softgoals and claim softgoals. In choosing an operationalizing softgoal, the developer considers it to be satisfied. In rejecting an operationalizing softgoal, the developer considers it to be denied. Such labellings are often applied to “leaf” operationalizing softgoals and claim softgoals, but in principle can be applied to any kind of softgoal, anywhere in a SIG.

Then the Framework’s evaluation procedure determines whether softgoals are satisfied or denied. It takes into account the developer’s decisions, labels of softgoals, and contribution types. It uses the rules introduced earlier in this chapter to assign labels to softgoals. The result for our example is shown in Figure 3.21. Here, the decisions at the bottom of Figure 3.20 have an impact on higher softgoals.

Let’s consider where the developer stepped in to assist the evaluation procedure. Selecting Validation contributed (“ \checkmark ”) to $!Accuracy[GoldAccount.highSpending]\{critical\}$. However, rejecting Auditing gave a weak negative contribution (“ W^- ”) to $!Accuracy[GoldAccount.highSpending]\{critical\}$. The developer felt that validation was sufficient to satisfy $!Accuracy[GoldAccount.highSpending]\{critical\}$, and declared it to be satisfied, by assigning it a label of “ \checkmark ”.

The developer also selected $PerformFirst[GoldAccount.highSpending]$. Performing operations related to gold accounts gave a weak positive contribution (“ W^+ ”) to $!ResponseTime[GoldAccount.highSpending]\{critical\}$, but the developer felt that it was sufficient to satisfy (“ \checkmark ”) the priority softgoal for response time. Similarly, the developer felt that satisfying $Accuracy[GoldAccount.highSpending]$ was sufficient to satisfy $Accuracy[GoldAccount]$. Likewise, satisfying $ResponseTime[GoldAccount.highSpending]$ was felt to be sufficient to satisfy $ResponseTime[GoldAccount]$.

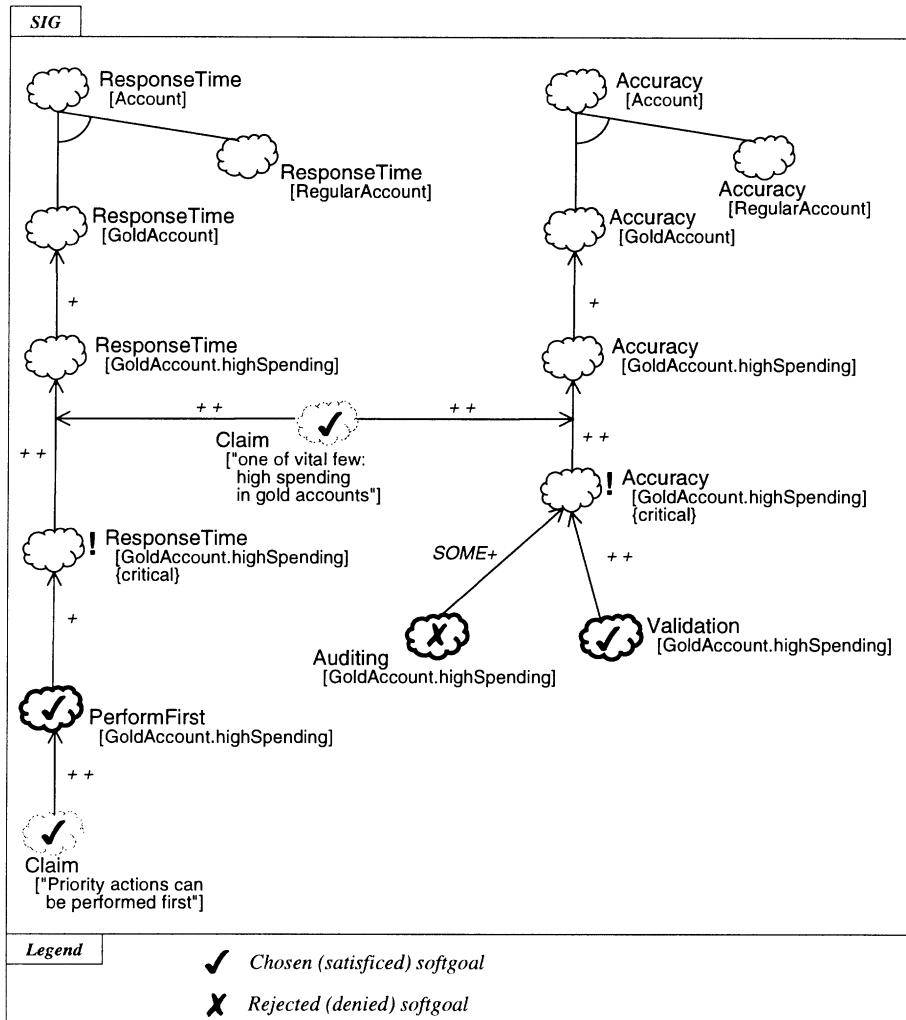


Figure 3.20. Recording developer's decisions to choose or reject softgoals.

3.4 COUPLING NFRs WITH FUNCTIONAL REQUIREMENTS

Let us consider the bigger picture of overall software development. A source specification with functional requirements (FRs) is dealt with, resulting in a description of the target system (e.g., a design or implementation [Chung91b, 91a]). There may be several possibilities for the target system. However, only

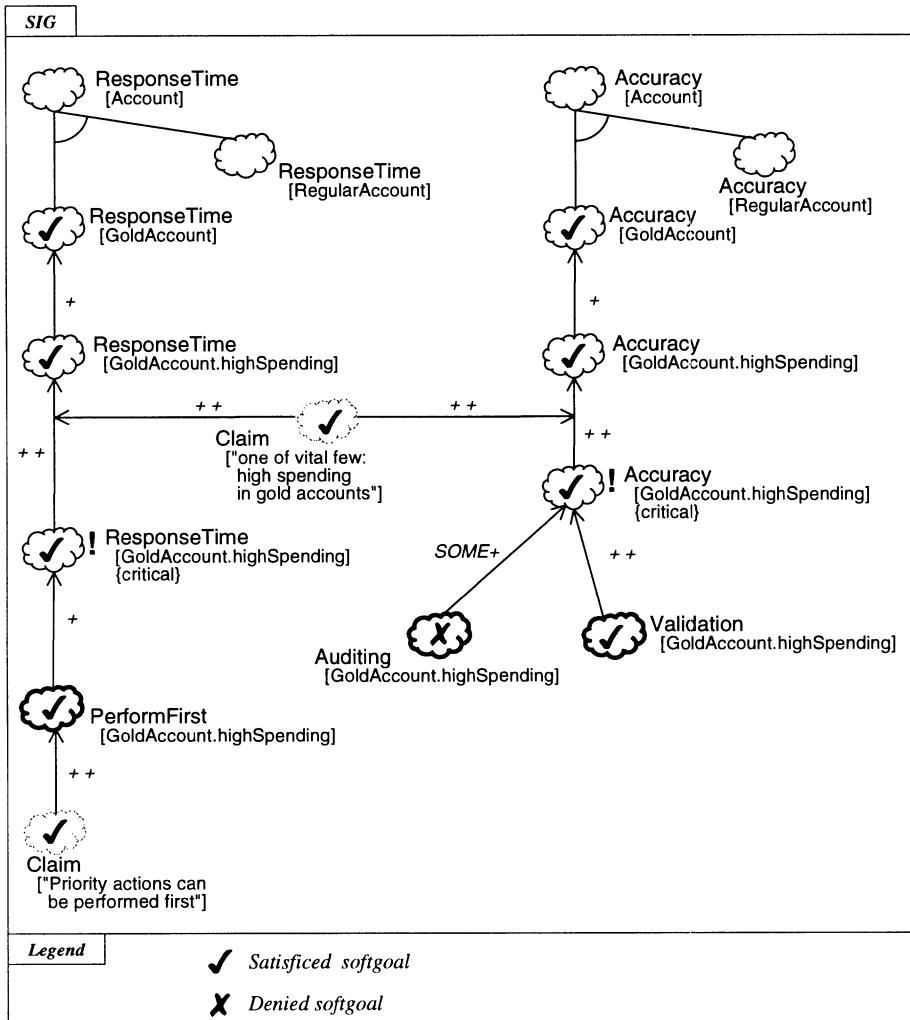


Figure 3.21. Determining the impact of developer's decisions, using the evaluation procedure.

one final system can be chosen, possibly involving several components. What alternative should the developer select?

The NFR Framework is intended to guide the developer to make such a selection. Take, for example, the functional requirement that “the system should maintain accounts.” For this source specification, there are several tar-

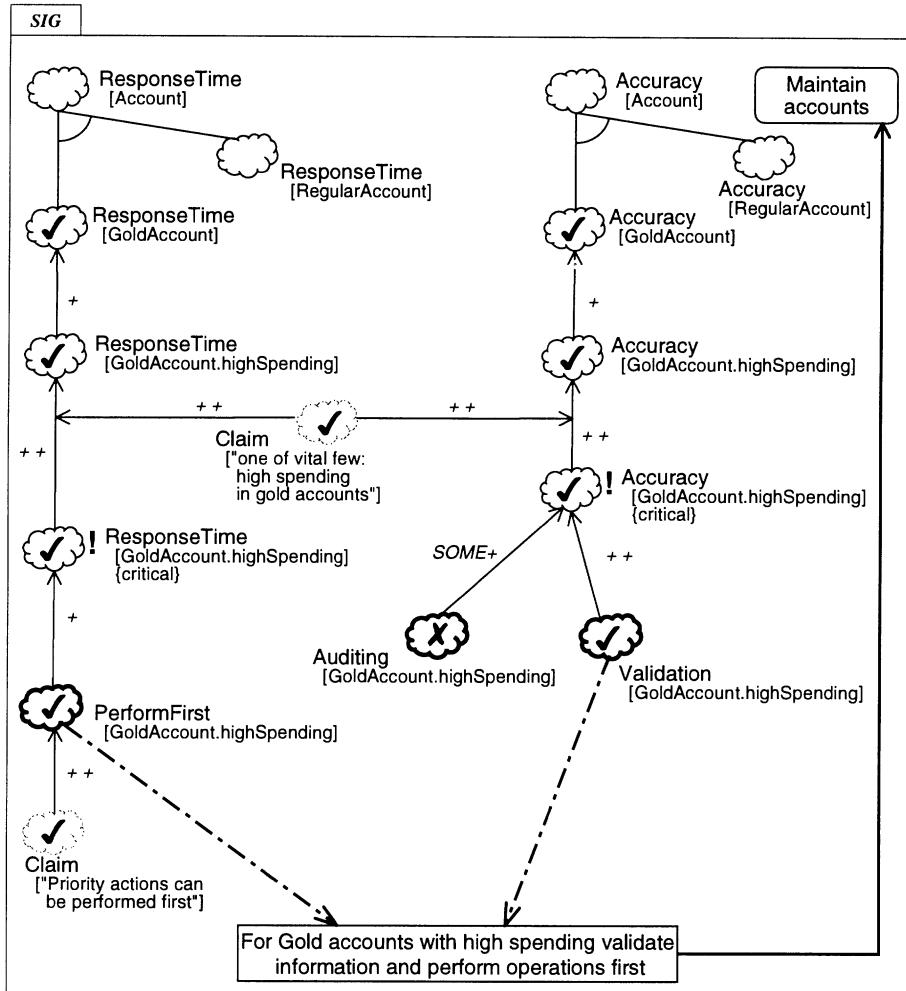


Figure 3.22. Relating functional requirements and the target system to a SIG.

get alternatives:

<u>Functional Requirements</u>	<u>Target Alternatives</u>
Maintain accounts	<ul style="list-style-type: none"> → – Validate information for all accounts. – Validate information for

- Maintain accounts → – Validate information for all accounts.
 – Validate information for

- gold accounts with high spending.
- Perform operations first for regular accounts.
- Perform operations first for
- gold accounts with high spending.
- Audit all accounts.
- Audit gold accounts with high spending.

For example, auditing all accounts may exhaust critical resources and prevent other NFRs from being met. As another example, if gold account customers get excellent response time, regular customers may become dissatisfied with poor service, and take their business elsewhere.

Figure 3.22, an elaboration of Figure 3.21, shows the selection of a target system, and relates it to the softgoal interdependency graph of our example. For gold accounts with high spending, information is validated, and associated operations are performed first. This is how the functional requirements of maintaining accounts are handled.

Figure 3.23 shows how (source) functional requirements and the selected target are related to each other and to the SIG of the NFR Framework. Functional requirements and the selected target are related via *design decision links*. NFR softgoals are related to the development process of taking functional requirements to a target system (or “designs”). In addition, operationalizing softgoals are related to the target specifications (“designs”) via *operationalization-target links*.

The figure shows how the development process is influenced and guided by NFR softgoals in general.

Now, where can the developer use the particular non-functional requirements at hand? Matters pertaining to system functionality may become topics of NFR softgoals. Account is one example. `InvoiceProcessingModule` could be another. Thus, when a functional requirement is being taken to a design, all those NFR softgoals having that requirement as a topic may act as criteria for selecting among target alternatives.

The NFR Framework makes it possible to systematically deal with non-functional requirements and ultimately use them as the criteria for making selections among competing target alternatives. Hence, it offers a goal-driven, process-oriented approach to dealing with non-functional requirements, complementary to other software engineering approaches which focus on products and their evaluation.

The relationship between the development phases of requirements specification, design specifications, and implementation is considered in the DAIDA framework for software development [Jarke92a, 93b]. DAIDA includes a requirements modelling language, RML [Greenspan82] [Greenspan94], a conceptual design specification language, TaxisDL [Borgida93], and a database implementation language, DBPL [Matthes93]. The case studies

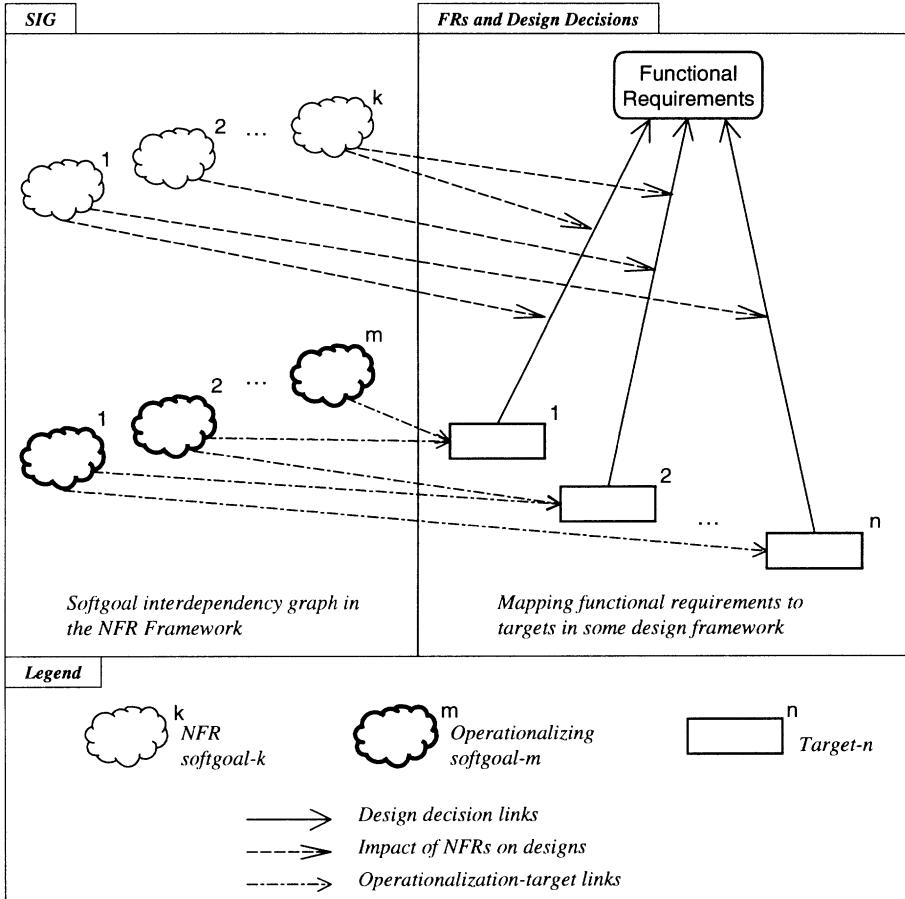


Figure 3.23. Dealing with functional requirements guided by considerations of NFRs.

done using the NFR Framework (See Part III) draw on the DAIDA framework, its languages, and the three phases of development which DAIDA considers.

We feel that the NFR Framework can be used in conjunction with a variety of frameworks for taking functional requirements to target systems. Section 3.5 outlines such frameworks, from areas including structured analysis, the Entity-Relationship approach, and object-oriented approaches.

3.5 DISCUSSION

Key Points

During the process of software development, softgoals and interdependencies are organized into a *softgoal interdependency graph (SIG)*. It is a record of what is addressed by the developer: NFRs, development alternatives and design rationale. All these are represented by softgoals. It also records design tradeoffs, which are represented as positive or negative contributions of operationalizing softgoals towards the satisficing of NFRs. Each softgoal is associated with a *label* which indicates the degree to which it is satisficed. Each interdependency has a *contribution*, which indicates the local impact of a softgoal. The *evaluation procedure* determines the overall impact of design decisions upon the satisficing of softgoals. This is done by considering labels (of both softgoals and contributions) and contribution values.

The NFR Framework records development decisions in *softgoal interdependency graphs*. The Framework offers five components which are used for dealing with non-functional requirements, *softgoals*, *interdependencies*, the *evaluation procedure*, *methods* and *correlations*. This chapter has presented the first three of them. The next chapter deals with methods and correlations.

The notion of *softgoal* enables non-functional requirements to be treated as softgoals which need to be *satisficed* instead of always being absolutely satisfied. Non-functional requirements are then refined via decompositions into other non-functional requirements. Refinements aid disambiguation and prioritization. These non-functional requirements are next satisficed by operationalizing softgoals, either design or implementation components, which can be used to meet functional requirements. Just like non-functional requirements, operationalizing softgoals can also be refined via decompositions into other operationalizing softgoals. In the presence of multiple competing techniques, tradeoffs are taken into consideration before some of them are finally chosen to be included in the target design or implementation. Throughout the refinements of non-functional requirements, their operationalization, prioritization and selection among competing techniques, arguments can be captured to support particular decisions.

The notion of *interdependency* enables softgoals to be interrelated. *Refinements* capture the refinement of softgoals into other softgoals. *Contributions* reflect the degree to which offspring softgoals satisfy the parent softgoal. Associated with each interdependency is one of several contribution types, indicating full or partial, positive or negative influence. Interdependencies themselves can contribute to other interdependencies, hence making it possible to talk about the relationship between a softgoal and an interdependency, etc.

The *evaluation procedure* determines the impact of design decisions upon the satisficing of softgoals and interdependencies. This is done by propagating labels of the offspring upward to the parents, while taking the contribution types into consideration.

The next chapter deals with *methods* and *correlations*. These help a developer to catalogue and systematically use knowledge when constructing *softgoal interdependency graphs*.

The concepts of the Framework help represent essential concepts of non-functional requirements and systematically deal with them. This is done by organizing, in a softgoal interdependency graph, NFRs and their associated alternatives, decisions and rationale. This organization then serves as criteria for selecting among target alternatives in meeting functional characteristics of a software system.

When addressing a particular type of non-functional requirement, one can use the representation and reasoning facilities of this chapter, without much change. The main work will be to catalogue the knowledge of the particular NFR type, using the counterpart of the next chapter to structure the knowledge.

Extensions

We can consider some extensions to the Framework. It might be helpful to have additional contribution types. One possibility would be a contribution which “inverts” the offspring’s label, taking \times in the offspring to \checkmark in parent, and \checkmark in the offspring to \times in the parent. Note however, that even if this were done, the rules for individual impacts for *SOME+* and *SOME-* would not change from what is shown in the tables.

We could consider modifying the evaluation procedure so that values such as W^+ and W^- could appear as outputs of the Second Step, instead of being eliminated. If this were the case, the Second Step would use the ordering

$$\emptyset \leq U \leq W^- \approx W^+ \leq \times \approx \checkmark$$

In addition, Tables 3.1 and 3.2 would have to be extended by adding rows for W^+ and W^- as *offspring*.

Literature Notes

This chapter is based on [Mylopoulos92a] [Chung93a].

Related literature for the NFR Framework is discussed at the end of Chapter 4.

The notion of “satisficing” was used by Herbert Simon, earlier in the context of economics, and later in the context of design [Simon81] where he actually used the term to refer to decision methods that look for satisfactory solutions rather than optimal ones. The term is adopted here in a broadened sense since in the context of non-functional requirements, the solution or optimality of a solution may be unclear. Since user requirements can be unclear, even the problem can be unclear.

Softgoal interdependency graphs are very much in the spirit of AND/OR trees used in problem-solving [Nilsson71]. Unlike AND/OR softgoal trees, where the relationship between a collection of offspring and their parent can only be *AND* or *OR*, in the NFR Framework there can be several different

types of relationships or *contribution types* describing how the satisficing (or denying) of offspring relates to the satisficing of the parent.

Contribution types are suggested by the literature, e.g., [Boehm78], which states that some quality characteristics are necessary, but not sufficient, for achieving others. A four-grade scale is then used to correlate each quality metric with quality attributes in the final product. Hauser et al. [Hauser88] use four types of values (strong positive, medium positive, medium negative and strong negative) to state how much each engineering characteristic affects each customer quality requirement. They are similar to those in [DiMarco90] and generally to those used in qualitative reasoning frameworks [AI84].

Leveson [Leveson86] presents a convincing argument for the need to explicitly capture tradeoffs, and an elegant discussion of various issues on safety. Leveson states that the effect of taking water is relative to the person who takes it. In other words, that water is good for human health is too coarse, but should be discussed, among other things, in terms of the average and threshold amount of water, as well as the age, body weight, health condition, etc., of the person who drinks it. This is, in spirit, also similar to the notions in [Garvin87] which describes a wide spectrum of different dimensions of quality. For instance, a quality concern such as performance, that lies on the one extreme, tends to be more objective than a concern on user perception, that lies on the other extreme, which is more subjective.

It is interesting to compare our evaluation procedure with those of truth maintenance systems (TMSs) [deKleer86] [Doyle79]. They record and maintain beliefs, their justifications and assumptions, while distinguishing facts from defeasible beliefs, which are either accepted or rejected. As with TMSs, our graph evaluation procedure recursively propagates values of offspring to parents. However, our procedure is not automatic, but interactively allows the developer to deal with inconclusive evidence. While we have *AND* and *OR*, comparable to TMS conjunction and disjunction, our contribution types have additional values, all of which are inputs to computing *individual impact* in our first step. In applying the propagation rules of Tables 3.1 and 3.2, *interdependencies*, which are not included in TMS beliefs, must be *satisficed*. Unlike TMSs, we then *combine* individual effects of label values including qualitative (*conflicting*) and open-ended (*undetermined*) ones, using a label ordering in the second step. In order to make sure that the evaluation procedure always terminates, techniques for cyclic graphs can be used, such as those used in constraint networks (e.g., [Dechter90]).

In the area of software engineering, the notion of operationalization has been around for quite some time. For example, in SADTTM [Ross77], management objectives are fulfilled through “operational concepts” which define system functions, which are then allocated as personnel functions, software functions and hardware functions. More recently, in KAOS [Dardenne93], goals of stakeholders are achieved through operationalization in terms of actions of agents or constraints.

There are a number of frameworks for taking functional requirements to target systems. We feel that the NFR Framework could be used in conjunction with many of them. Such frameworks could include Structured Analysis, e.g., Data Flow Diagrams (DFDs) [DeMarco78], Jackson System Development (JSD) [Jackson83], Structured Analysis and Design Technique (SADTTM) [Ross77], and the Entity-Relationship approach [Chen76]. More recent proposals for design frameworks, such as object-oriented analysis and design (e.g., [Booch94] [Booch97] [Coad90] [Coleman91] [Jacobson92] [Martin95] [Rumbaugh91] [Shlaer88] [Quatrani98]), ERAE [Dubois86], and KAOS [Dardenne93], could also be suitable for use with the NFR Framework.

4 CATALOGUING REFINEMENT METHODS AND CORRELATIONS

During the process of software development, developers use *softgoals* and *interdependencies* to analyze and record in a softgoal interdependency graph their intentions, design alternatives and tradeoffs and rationale. The *evaluation procedure* is then used to determine if their softgoals have been met.

In this chapter, we present the remaining components of the NFR Framework: *refinement methods* and *correlations*. These two components help the developer generate a softgoal interdependency graph, by allowing for knowledge about ways of softgoal refinements and their interactions to be captured, catalogued, tailored, reused and improved. They show the impact of one softgoal upon another.

Refinement methods and correlations can be defined, collected and organized into *catalogues*. Catalogues are then available for sharing, reuse and adaptation, within the same application contexts and across different ones. This helps alleviate time-consuming and often difficult searches for know-how.

Refinement methods and correlations elaborate existing graphs. This is done by using the developer's expertise and knowledge about the domain and functional requirements, while considering elements of the graph. Applicable patterns are examined, and appropriate methods and correlations are selected.

Refinement methods are used by the developer to refine and extend a particular portion of a graph. Methods can be taken from catalogues or from the developer's expertise.

Correlation rules, in contrast, may elaborate any part of an existing graph. The graph is examined for patterns which are matched with a catalogue of correlation rules. This matching can be done by a developer “by hand,” or can be done by a software tool.

The chapter concludes with a discussion of the overall process of using the NFR Framework, and a review of work which is related to the Framework.

4.1 REFINEMENT METHODS

Refinement methods (or *methods*) are generic procedures for refining a softgoal or interdependency into one or more offspring. When a number of methods are collected and catalogued, they can offer appropriate vocabulary and subject matter for dealing with NFRs.

Recall that there are three kinds of refinements used to relate softgoals to each other. Section 3.2 presented decomposition, operationalization and argumentation refinements. Three corresponding kinds of *refinement methods* are applied (used) to guide these refinements:

- *NFR decomposition methods*,
- *operationalization methods*, and
- *argumentation methods*.

Methods are used to systematically make refinements, using the three kinds of softgoals and several contribution types.

NFR decomposition methods iteratively refine NFR softgoals into more specific NFR softgoals. When NFRs are sufficiently refined, the developer then uses operationalization methods to satisfice such NFRs. Along the way, design decisions can be recorded using argumentation methods.

During the development process, softgoals may be refined by the developer in an *ad hoc* manner. However, this can be time-consuming, especially when the know-how for such a refinement is not readily available, as it takes time and effort to search for the source of the know-how. Even when available, the know-how might not be properly encoded, making it harder to be shared, extended, tailored, or reused. Refinement methods help to alleviate these difficulties.

Now we move onto the three major kinds of methods mentioned earlier, namely, NFR decomposition, operationalization, and argumentation methods. Let us start with NFR decomposition methods.

4.2 NFR DECOMPOSITION METHODS

Recall that initial NFR softgoals are often coarse-grained. Thus they do not permit the consideration of tradeoffs and design decisions which normally require more specific details. In addition, initial NFR softgoals can be ambiguous, and invite many interpretations from different groups of people.

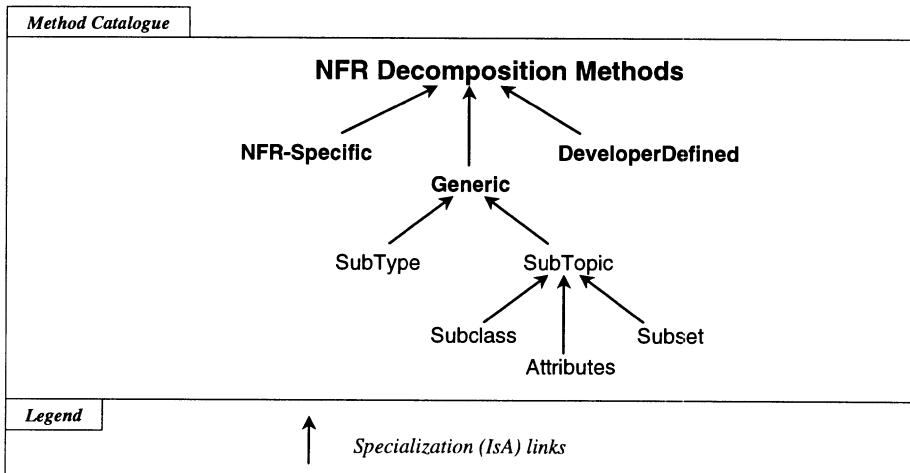


Figure 4.1. A catalogue of NFR decomposition methods.

NFR decomposition methods record the know-how for breaking NFR softgoals down into more detailed ones, and interpreting them in different (and possibly conflicting) ways. This allows multiple conflicting interpretations of non-functional requirements to coexist. The developer can choose the one that best suits the needs of the intended application domain.

NFR decomposition methods also allow for the “divide-and-conquer” paradigm to be exercised. This helps deal with as many NFRs as needed. It also helps divide a problem into components; when each component is solved, the overall problem is solved. Hence NFR Decomposition methods which break down a parent softgoal into a number of offspring softgoals often use *AND* contributions. If there is a single offspring, *HELP* and *MAKE* contributions are common.

Figure 4.1 shows a *catalogue of NFR decomposition methods*. More general methods are shown above more specific ones. We now discuss some of the groups of methods shown in the figure.

The *generic methods* include refinements on type and on topic. By refining on NFR type, we can address more specific aspects of NFRs. Refinements on topic include structural decomposition. For example, a softgoal about an organization can be decomposed into softgoals dealing with parts of the organization. Similarly, a softgoal about a software system can be decomposed into softgoals for software components. We will discuss some of these generic refinements in this chapter. These include decompositions on features common to several data models.

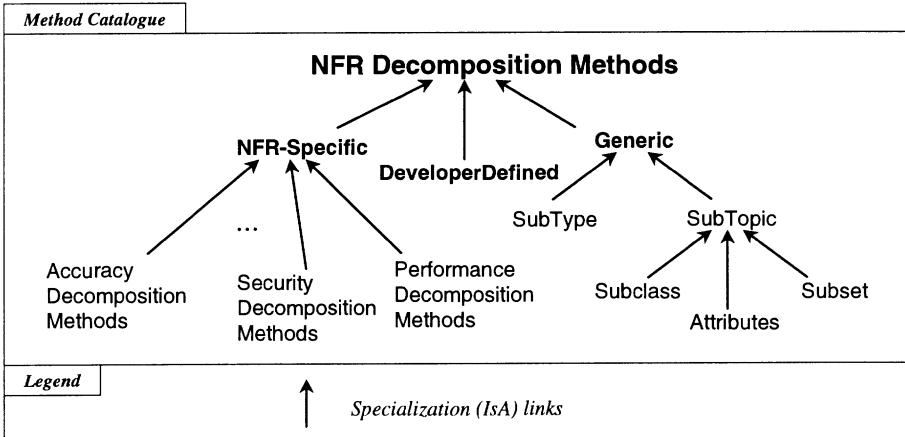


Figure 4.2. A catalogue of NFR decomposition methods, including those for specific NFRs.

Note that these generic decompositions can be specialized. For example, we can deal with features and structures of particular data models; this is illustrated in Chapters 8 and 9 on performance requirements. Likewise, the use of decomposition methods to deal with ambiguities is shown in detail in Chapters 6 and 7 on accuracy and security requirements.

NFR-specific decomposition methods apply to particular NFRs, such as accuracy, security and performance, outlined in Figure 4.2, an elaboration of Figure 4.1. Detailed method catalogues for particular NFRs are presented in Part II of this book.

Let's consider some simple refinement methods. For example,

“To have good response time for operations on credit card accounts, you need good response time for regular accounts and gold accounts.”

might come from experts in the credit card system domain. Once acquired, this know-how can be represented as a method as in Figure 4.3. The figure is broken into parts. Logos at the top left indicate the nature of each part of the figure.

The method definition is shown in the top of the figure. Logos of figures indicate the degree of parameterization of the definition. This definition is not parameterized, i.e., it is quite “concrete.”

An important part of methods is their use of *functional requirements* and *domain information*. Here, we use the fact that the *subclasses* of the *Account* class are *RegularAccount* and *GoldAccount*.

The bottom of this figure shows the method application. The left side shows the initial softgoal interdependency graph, here with the softgoal Re-

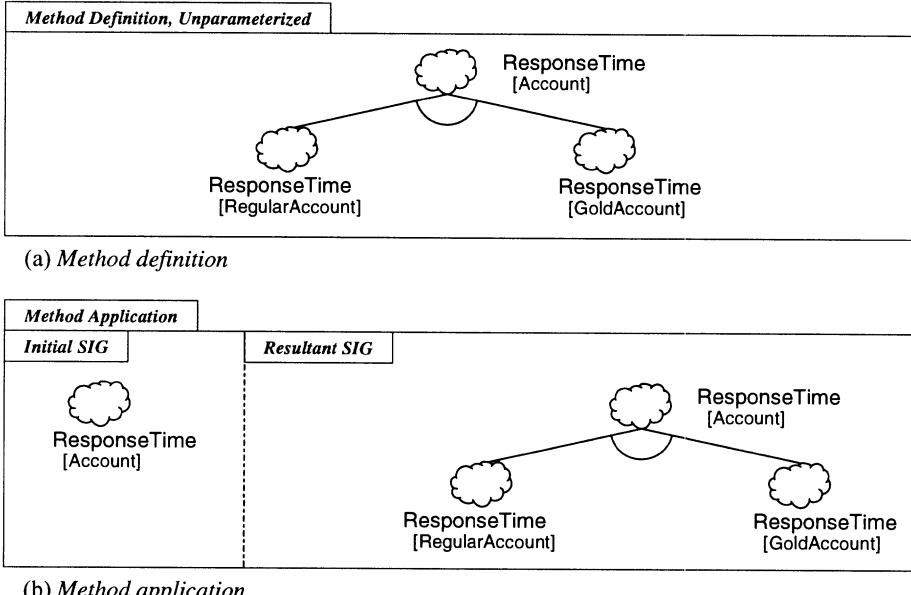


Figure 4.3. Definition and application of the `AccountResponseTimeViaSubclass` decomposition method.

`sponseTime[Account]`. The right side shows the SIG which results when we start with the initial SIG and apply the method. In this case, the right side contains the already-existing parent, its newly-created offspring, `ResponseTime[RegularAccount]` and `ResponseTime[GoldAccount]`, and the interdependency between parent and offspring. Here the result is an *AND* contribution of the offspring to the parent. That is, the parent will be satisfied if all the offspring are.

Methods can be summarized and identified by their contribution, here:

`ResponseTime[RegularAccount] AND ResponseTime[GoldAccount]`
`SATISFICE ResponseTime[Account]`

Where desired, methods can also be identified by a name, here `AccountResponseTimeViaSubclass`. The convention is that the name includes the parent softgoal's *topic* (here, `Account`), followed by its *type* (`ResponseTime`), followed by “*Via*” and the *kind of refinement* (here, `Subclass`). Methods which are more generic (parameterized) will have names which omit some of these components.

Method definitions can also be shown in a frame-like notation:

NFR DecompositionMethod AccountResponseTimeViaSubclass

parent: ResponseTime[Account]
 offspring: {ResponseTime [RegularAccount],
 ResponseTime[GoldAccount]}
 contribution: AND

The definition starts with keywords which identify the kind of method, here **NFR DecompositionMethod**.

Once a method is defined, it can be used to extend a portion of a softgoal interdependency graph. Given a particular parent in a softgoal interdependency graph, its offspring are generated according to the definition of the method being applied.

In order to be sure about this, let us see how a method application works. Suppose that the developer has selected **ResponseTime[Account]** as the softgoal to refine in an existing SIG. Also suppose that the developer has selected **AccountResponseTimeViaSubclass** as the method to apply to the softgoal. At this point, the chosen method is applicable to the chosen softgoal only if both the type in the method definition matches that of the softgoal and the topic list in the method definition matches that of the softgoal. If the method is indeed applicable, offspring of the softgoal are generated, along with a contribution, as specified in the method definition.

In the case of **AccountResponseTimeViaSubclass**, a method takes a parent having a fixed type (here, **ResponseTime**) and a fixed topic list (here, **Account**), and then generates offspring whose types and topics are again fixed.

While this kind of unparameterized method is useful for capturing domain-specific refinement knowledge, a method becomes less domain-dependent, more widely applicable and more amenable to tool support when the type and topics are parameterized, using variables. This is fairly easy to do, using slight extensions to the mechanisms we have already introduced.

For example, the following is a method which has a variable as topic:

“To have good response time for class *cl*, you need to have good response time for all relevant subclasses of *cl*.”

Here the softgoal topic is a variable. The above know-how is represented in Figure 4.4.

Part (a) of the figure has the method definition, which is parameterized. Parameters (variables) are shown in sans serif italics. We can *parameterize* on the *type* or *topic* of a softgoal. Here the topic is a parameter, and the parameter must be a class.

When a method involves a variable, an application of the method can benefit from the *functional requirements* description. The description usually includes *structural requirements* which model information structures and rules which determine allowable states of these structures, and *behavioural requirements* which model activities that operate on such structures and events that trigger these activities.

Part (b) of the figure shows a portion of the *functional requirements* (*FRs* in the logo). Here we have the subclasses of the class **Account**, namely,

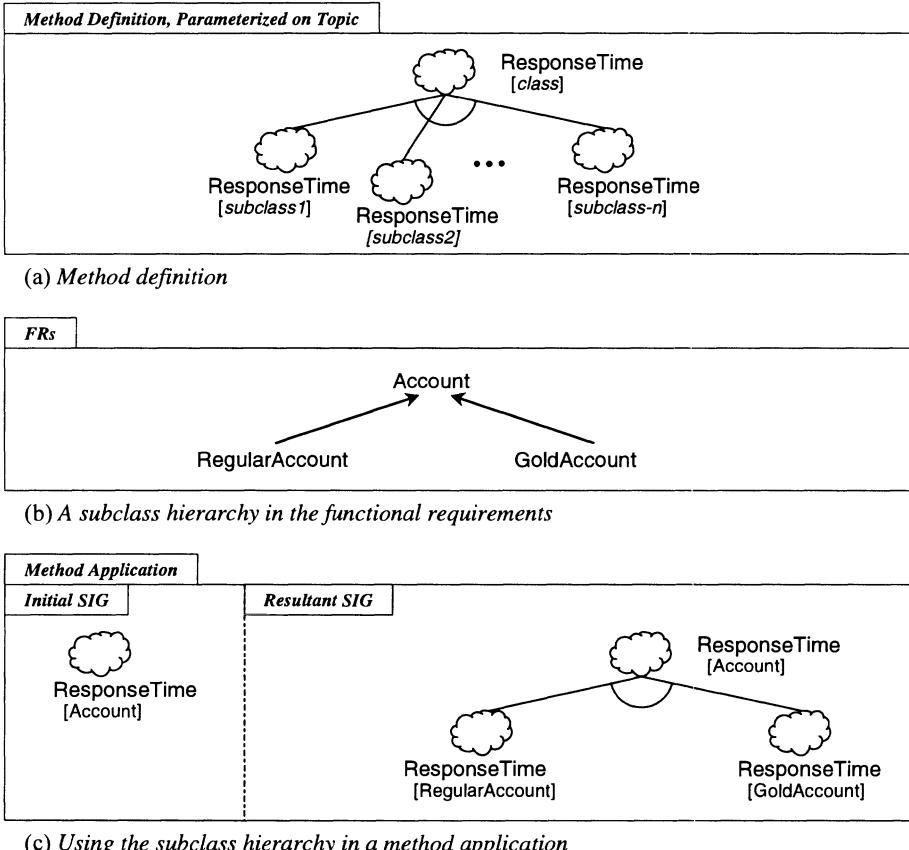


Figure 4.4. Definition and application of `ResponseTimeViaSubclass`, a parameterized decomposition method.

the class of regular accounts, and the class of gold accounts. They are arranged in a specialization (IsA) hierarchy, where general classes are above specialized ones.

Part (c) of the figure shows the method application. Here, the functional requirements from part (b) are used (substituted) as values for the parameters of part (a). The application of the `ResponseTimeViaSubclass` method to `ResponseTime[Account]` will result in the generation of two offspring softgoals.

Figure 4.4 as a whole, thus, illustrates the definition of a method involving a variable, the use of functional requirements, and finally an application of the method using the functional requirements. This three-part figure show-

ing a parameterized method definition, functional requirements, and a method application, will be frequently used in this chapter.

Note that the method name here, `ResponseTimeViaSubclass` is shorter than the previous one, `AccountResponseTimeViaSubclass`. The topic at the front (`Account`) is omitted, because it is a parameter.

A frame-like definition can also be given for this kind of method. Parameters are in sans serif italics.

NFR DecompositionMethod `ResponseTimeViaSubclass`

```

parent: ResponseTime[cl]
offspring: {ResponseTime[cl1], ..., ResponseTime[cln]}
contribution: AND
applicabilityCondition: cl: InformationClass
constraint: forAll i: cli isA cl
    and /* set up one offspring for every subclass of cl */

```

Each method definition specifies the parent softgoal (its type and topic(s)) followed by a list of offspring (with type and topic(s) for each). The **contribution** of the offspring to the parent is also indicated.

The **applicabilityCondition** describes the kind of parent in a softgoal interdependency graph to which the method can be applied. Here the topic of the parent must be an information class.

In contrast, the **constraint** describes the kind of offspring that should be generated when the method is applied to the parent. Here, one offspring should be generated for every subclass of the parent's topic.

The **applicabilityCondition** and **constraint** can often be omitted. When included, they can be described formally, informally, or using a combination of styles.

Now let's consider the impact of the method application. Note that the net result of all this is identical with the application of the earlier `AccountResponseTimeViaSubclass` method. Why do we now bother with the `ResponseTime`

`ViaSubclass` method, when the use of the `AccountResponseTimeViaSubclass` method does not even need a class hierarchy from the functional requirements?

It's because `AccountResponseTimeViaSubclass` can *only* be applied to the `ResponseTime[Account]` softgoal, and none other. In contrast, for the `ResponseTimeViaSubclass` method, the topic can be just about anything pertaining to information. In other words, the method can be applied not only to the response time for all account information but also to the response time for very specialized fields. For example, the method can be applied to medical test information, which should be of concern to patients and practitioners alike in the context of a health care system. Now suppose test information can be categorized into blood, heart- kidney and x-ray information. Then the method application results in the generation of four offspring, each having as its topic one of the four kinds of test information. Figure 4.5 illustrates this.

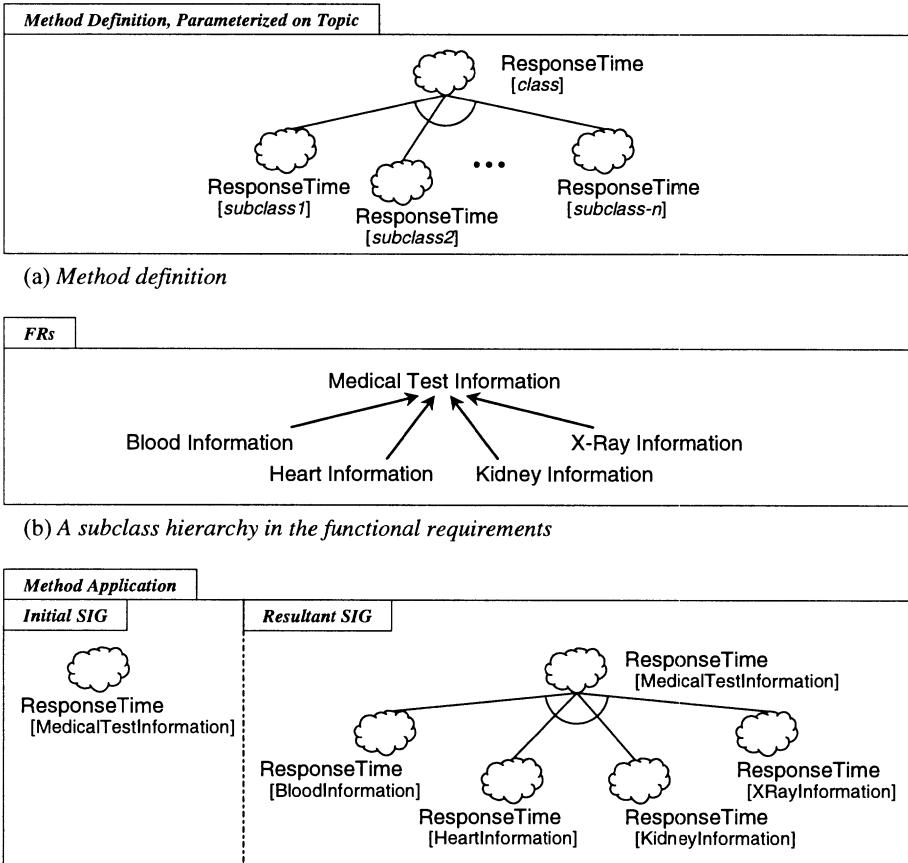


Figure 4.5. Another application of the ResponseTimeViaSubclass method.

Note that the method definition is the same as in Figure 4.4. However, Figure 4.5 also shows a different class hierarchy from the functional requirements. This is used to replace the variable topic in the method application.

Softgoals with multiple topics are refined just the way softgoals with a single topic are. This is illustrated in Figure 4.6 for an operating cost softgoal with two topics. Here, both topics are refined. In the subclass hierarchy, Staffing is refined into full-time and part-time employees, while the amount is refined into applicable dollar limits for each staff category. Here, a limit of \$50 000 is a specialization of a limit of \$100 000.

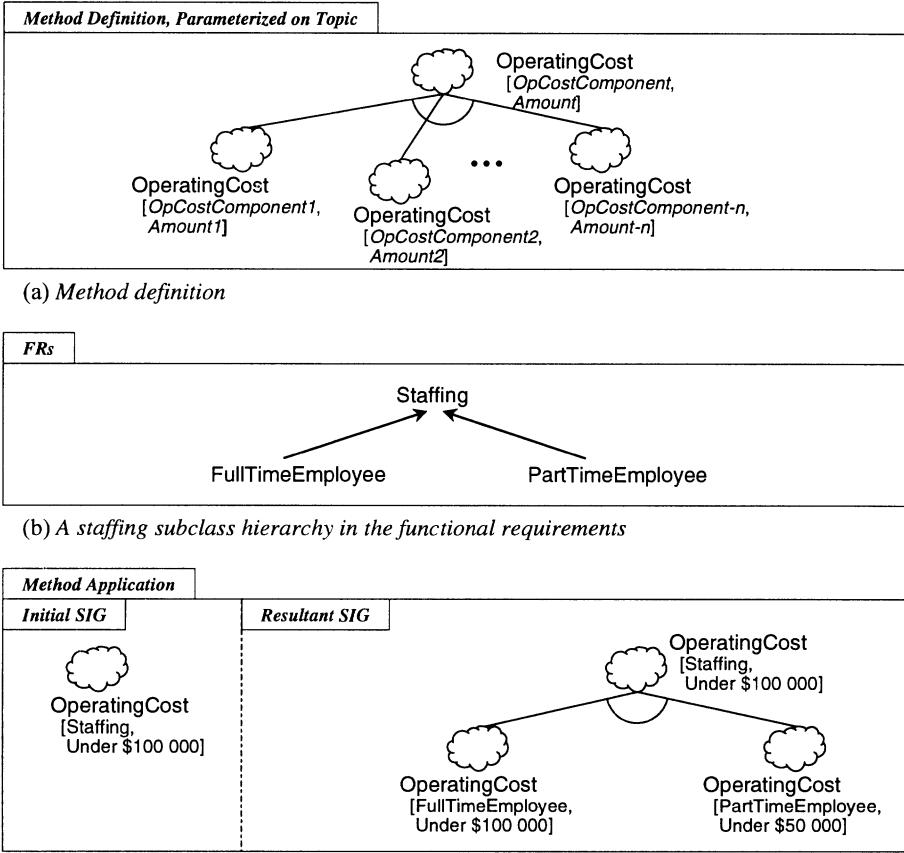


Figure 4.6. The StaffingAndAmountForOperatingCostViaSubclassAndDollarLimit method, parameterized on two topics.

At this point we should note that we may need to address the correctness of a refinement method. For generic methods, we assume that they have been defined carefully and correctly. However, the correctness of *ad hoc* refinements must be considered by the developer on a case-by-case basis. We handle this by representing the refinement itself as an interdependency. In the case of generic methods, the refinement is assumed to be satisfied. For *ad hoc* refinements, the developer must satisfy the interdependency.

Let's discuss in more detail the two important kinds of NFR decomposition methods: *decompositions on NFR type* and *decompositions on topic*.

Decomposition on NFR Type

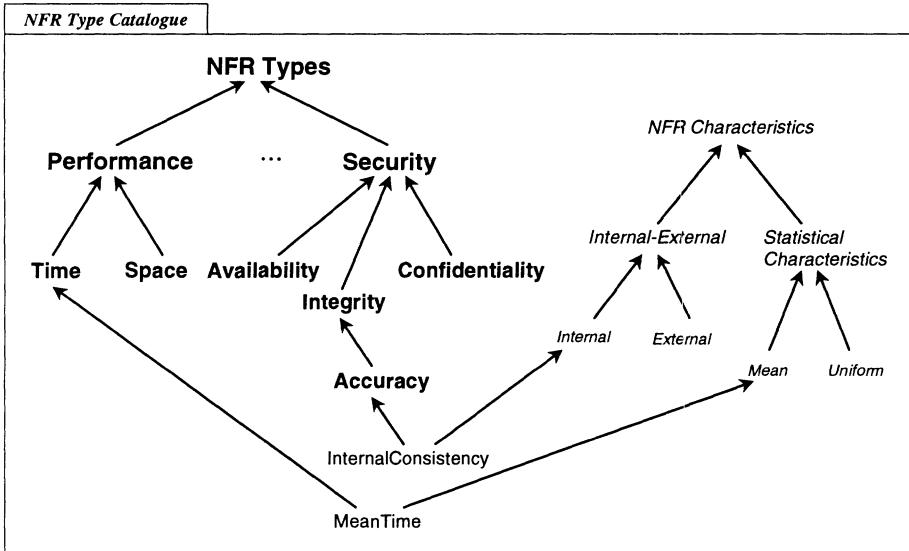


Figure 4.7. A catalogue of NFR Types.

NFR type decomposition methods refine and relate the types of NFR softgoals which represent the types of system characteristics, such as accuracy, response time, operating cost, modifiability and traceability.

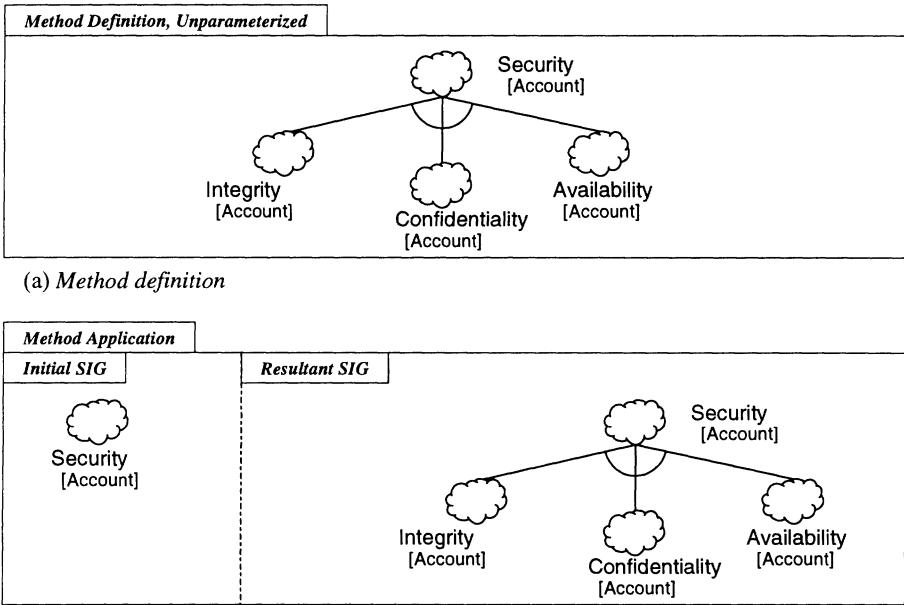
NFR type decomposition methods are particular kinds of decomposition methods which in turn are particular kinds of refinement methods. As such, NFR type decomposition methods inherit the nature and benefits of NFR decomposition methods as well as refinement methods.

An NFR type decomposition method allows for the refinement of a parent on its type in terms of offspring, each with a subtype of the parent type. Each subtype can be viewed as representing special cases for each softgoal class.

Figure 4.7 shows a *catalogue of NFR types*. Specialized types are shown underneath general ones. The entries in bold face include the NFRs that are discussed in detail in this book.

Characteristics of NFRs are shown in small italics at the right of Figure 4.7. They are not NFR types in their own right; however they are characteristics which can modify and specialize the meaning of a type. Specialized NFR types can then be formed from combinations of type and characteristics. For example, *internal* consistency (within the system) and *external* consistency (outside the system) are two kinds of consistency. As another example, we can

refer to *mean (average)* response time, and *uniform* response time (i.e., with small deviation from the mean).



(a) *Method definition*

(b) *Method application*

Figure 4.8. Definition and application of the `AccountSecurityViaSubType` method.

Let us consider a SubType refinement. Consider the `Security` type with subtypes `Integrity`, `Confidentiality` and `Availability`. Figure 4.8 defines and applies the unparameterized `AccountSecurityViaSubType` method.

NFR Type decomposition methods also allow for variables to be used in place of the type.

Figure 4.9 defines and applies the `AccountQualityViaSubType` method, which is parameterized on NFR type. It takes a parent softgoal dealing with accounts for a given NFR (quality attribute). It produces offspring dealing with accounts for sub-types of the given NFR type.

This is more general than the `AccountSecurityViaSubType` method of Figure 4.8, as it can deal with various NFR types. In Figure 4.9, the `AccountQualityViaSubType` method is applied to `Security`, which has subtypes `Integrity`, `Confidentiality`, and `Availability`. Note that the topic of the parent (i.e., `Account`) is copied down to the offspring; only the type of the offspring differ from that of the parent. In the method name, `Quality` is a placeholder for the NFR type (quality attribute), which is a variable.

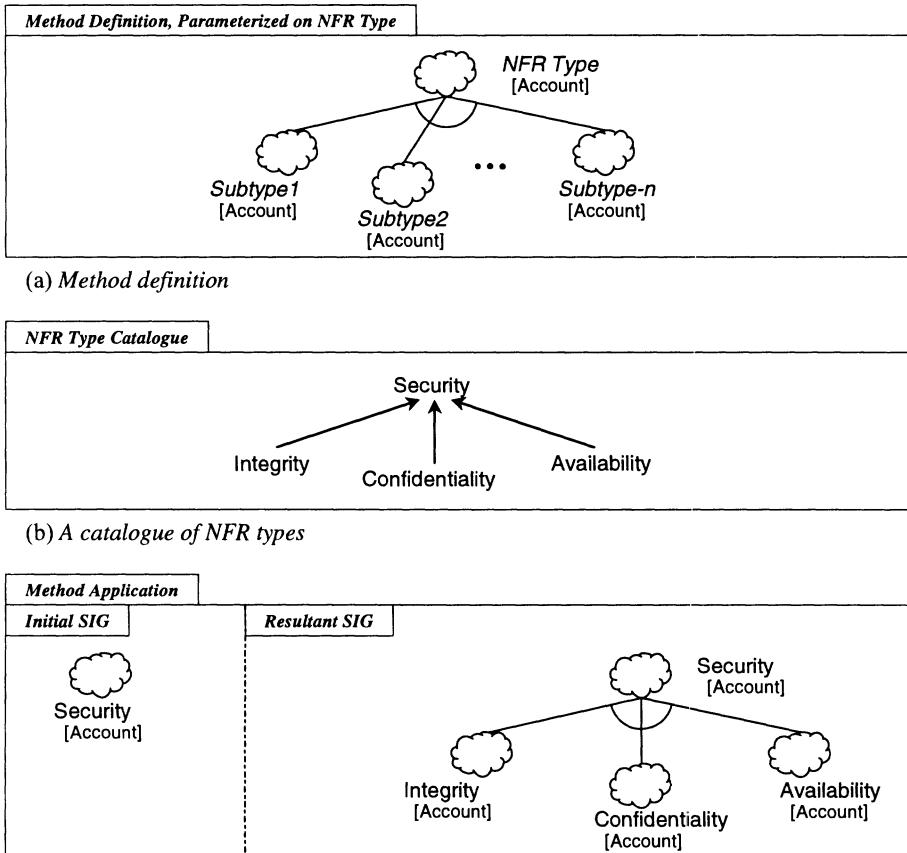


Figure 4.9. The AccountQualityViaSubType method, parameterized on NFR type.

The method can also be written as:

NFR DecompositionMethod AccountQualityViaSubType

parent: $ty[\text{Account}]$

offspring: $ty_1[\text{Account}], \dots, ty_n[\text{Account}]$

contribution: AND

applicabilityCondition: $ty: \text{NFR Type}$

constraint: forAll $i: ty_i \text{ isA } ty$

and /* set up one offspring for every subclass of ty */

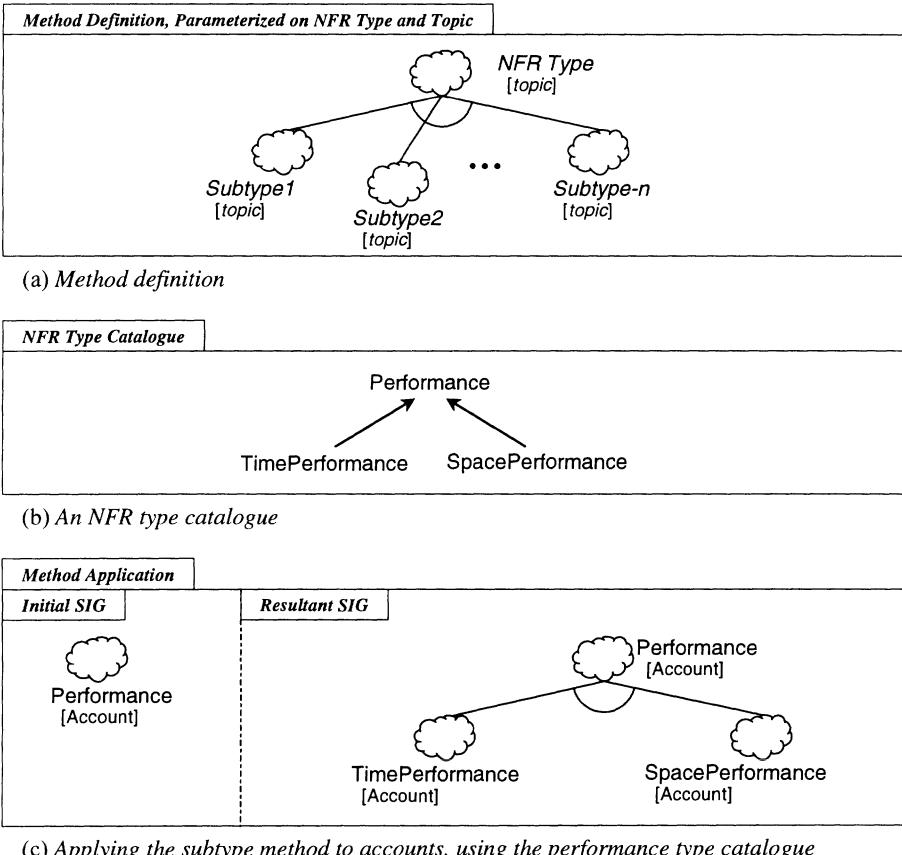


Figure 4.10. The SubType method, parameterized on NFR Type and Topic, and applied to performance of accounts.

Of course, the **AccountQualityViaSubType** method can take any softgoal as the parent, as long as the softgoal has accounts as its topic. For example, the method can be applied to **Adaptability[Account]** and result in the generation of **DetectabilityOfChange[Account]** and **Modifiability[Account]**, where the type **Adaptability** has two subtypes **DetectabilityOfChange** and **Modifiability**. Thus, the **AccountQualityViaSubType** method may be viewed as being more generic than the **AccountSecurityViaSubType** method.

An even more generic method can be defined which can be applied to virtually any softgoal having variables for both its type and topics. Figure 4.10 shows the **SubType** method. Given a parent, it produces offspring which have

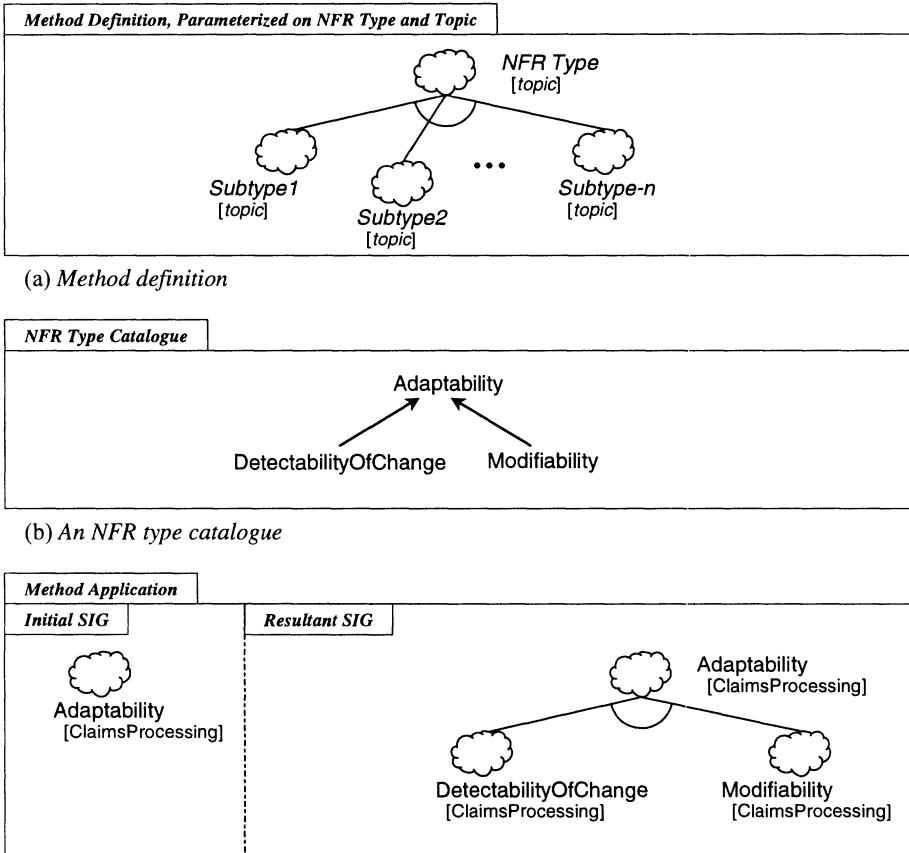


Figure 4.11. The SubType method, applied to adaptability of insurance claims processing.

the same topic as the parent. In addition, the subtypes of the parent's type are used as the types of the offspring. This method is parameterized on both type and topic. It is applied to performance of accounts. The offspring deal with time performance and space performance of accounts.

This method can be summarized as:

$$\begin{aligned}
 & ty_1[to] \text{ AND } \dots \text{ AND } ty_n[to] \\
 & \text{SATISFICE } ty[to]
 \end{aligned}$$

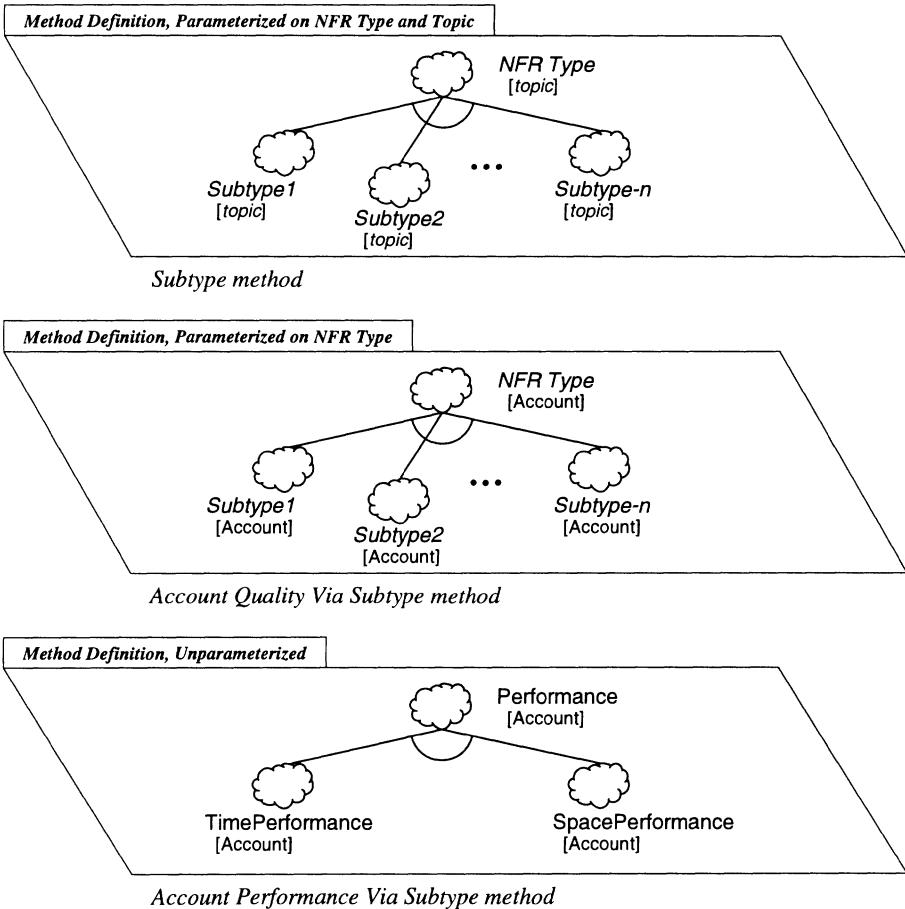


Figure 4.12. The SubType type decomposition methods, with varying degrees of parameterization.

Here, ty is a NFR type, and each ty_i is a sub-type of ty . One offspring is set up for each subtype of ty .

Now, this method can be applied not only to $Security[Account]$ and $Performance[Account]$, but also $Security[Customer]$, $Security[Address]$, $Performance[Customer]$, $Performance[Address]$, etc. This method is quite domain-independent, and as such it can be used in many contexts: systems for banking, credit card, insurance, health, military, etc.

To illustrate this domain independence, consider another application of the method, to the insurance domain. In Figure 4.11 the method is applied to

an adaptability softgoal for claims processing, **Adaptability[ClaimsProcessing]**. Note that this figure has the same method definition as Figure 4.10, but applies the method to a different type and a different topic. Now the type **Adaptability** has two subtypes **DetectabilityOfChange** and **Modifiability**. The figure illustrates the application of the **SubType** method to the adaptability softgoal. This results in the generation of **DetectabilityOfChange[ClaimsProcessing]** and **Modifiability[ClaimsProcessing]** as offspring.

Figure 4.12 shows a family of **SubType** methods. The three type decomposition methods have an increasing degree of parameterization. The bottom layer shows the unparameterized **AccountPerformanceViaSubType** method. The middle layer shows **AccountQualityViaSubType** parameterized on type. The top layer shows the **SubType** method, parameterized on both type and topic.

It is also possible to have parameterization on topic only. This will be shown next.

Decomposition on Softgoal Topic

Topic decomposition methods relate and refine topics of softgoals.

Like type decomposition methods, topic decomposition methods are specializations of decomposition methods which are in turn specializations of methods. As such, topic decomposition methods inherit all the benefits of decomposition methods and methods in general.

A topic of a softgoal can refer to a collection of items, such as information, processes, functions, events, interfaces, time, monetary amount, etc. Used for describing functional aspects of the system, these items are usually organized along structural modelling primitives of functional requirements, system and software architectures, or process architectures. Such primitives include classes, super-classes and sub-classes, attributes, views, contexts, perspectives, programs, packages and components, etc.

Whatever a topic may refer to, it can be decomposed along a structural dimension. For instance, the **StaffingAndAmountForOperatingCostViaSubclassAndDollarLimit** method shown in Figure 4.6 is a topic decomposition method which decomposes topics [**Staffing**, Under \$100 000] into [**FullTimeEmployee**, Under \$100 000] and [**PartTimeEmployee**, Under \$50 000]. In other words, a general limit of \$100 000 for staffing is also used for full-time employees, but specialized to a limit of \$50 000 for part-time employees.

Recall the varying degrees of type parameterization in Figure 4.12, where the **SubType** method is more generic than the **AccountQualityViaSubType** method, which in turn is more generic than the **AccountPerformanceViaSubType** method.

Similarly, topic decomposition methods can have varying degrees of parameterization. For example, the **AccountResponseTimeViaSubclass** and more generic **ResponseTimeViaSubclass** methods of Section 4.2 are also topic decomposition methods which decompose the topic of a response-time softgoal in terms of subclasses of the topic of the softgoal. We can define an even more generic method which can be applied to virtually any softgoal, because it has variables for both the type and topics. The **Subclass** method of Figure 4.13 de-

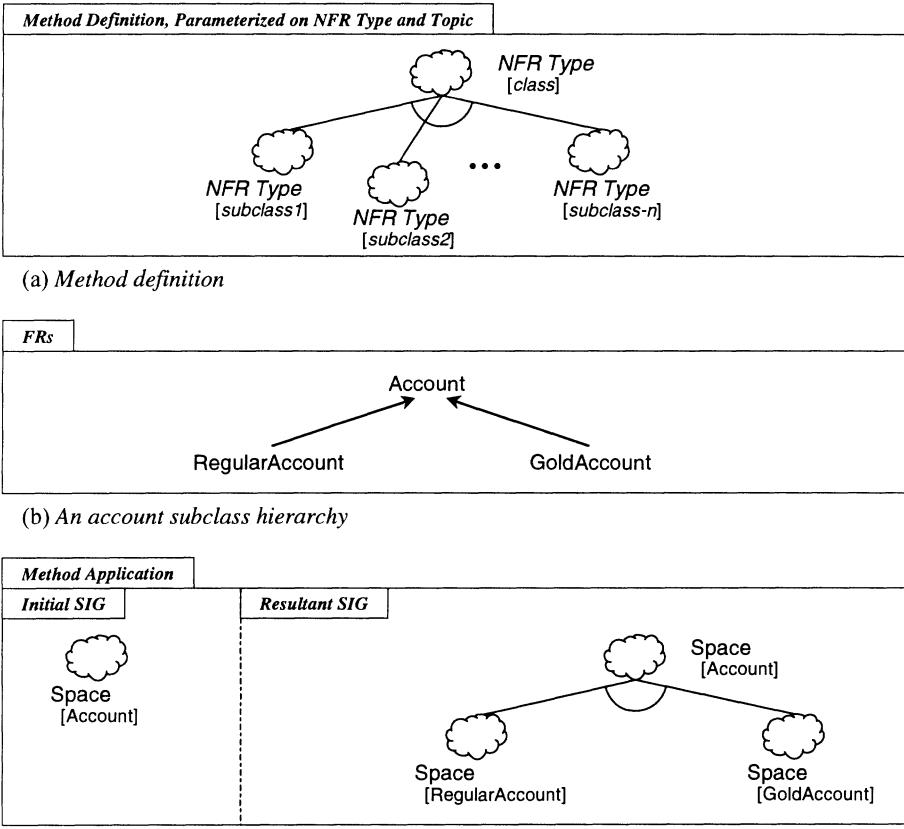


Figure 4.13. The Subclass method, applied to space performance of accounts.

composes any softgoal with a class as the topic into softgoals having as topics the immediate subclasses (specializations) of the class.

Note that the type of the parent is copied down to the offspring; only the topics of the offspring differ from those of the parent. In the figure, we apply the generic method to deal with space requirements for accounts. We use the account *subclass hierarchy* from the functional requirements to determine the offspring topics, RegularAccount and GoldAccount.

NFR DecompositionMethod Subclass

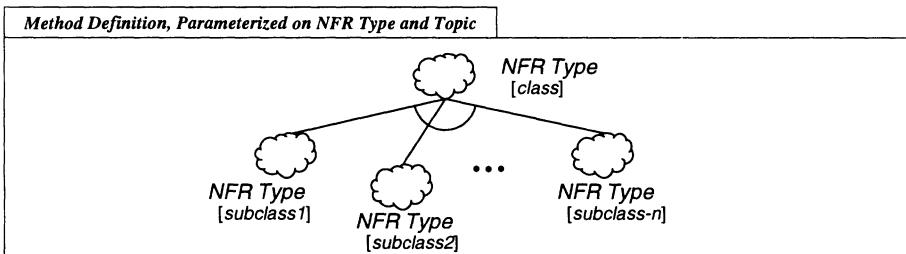
```

parent: ty [to]
offspring: {ty [to1], ty [to2], ..., ty [ton]}
```

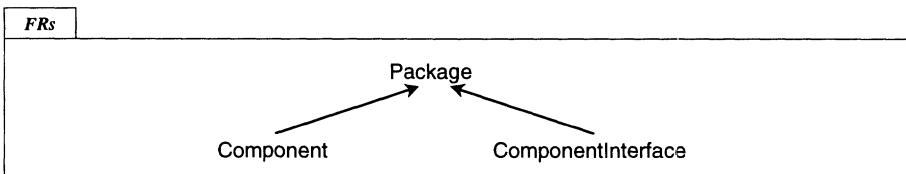
contribution: *AND*

applicabilityCondition: *ty : NFR Type*
to: Information Item

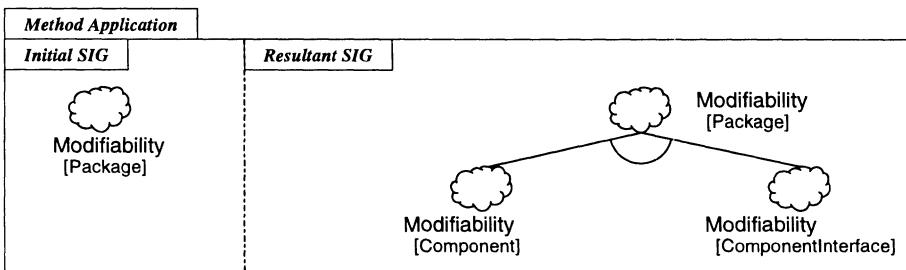
constraint: *forAll i: to; isA to*
and / set up one offspring for every subclass of to */*



(a) *Method definition*



(b) *A package subclass hierarchy*



(c) *Applying the subclass method to the package subclasses, for modifiability requirements*

Figure 4.14. The Subclass method, applied to modifiability of packages.

Importantly, the type and the topics of the parent are not limited to **Space** and **Accounts**. Instead the method can be invoked for any meaningful type and topics. For example, Figure 4.14 shows how the method can be applied to a modifiability softgoal for software packages. It uses the same method definition as Figure 4.13, but applies it to a different type and topic.

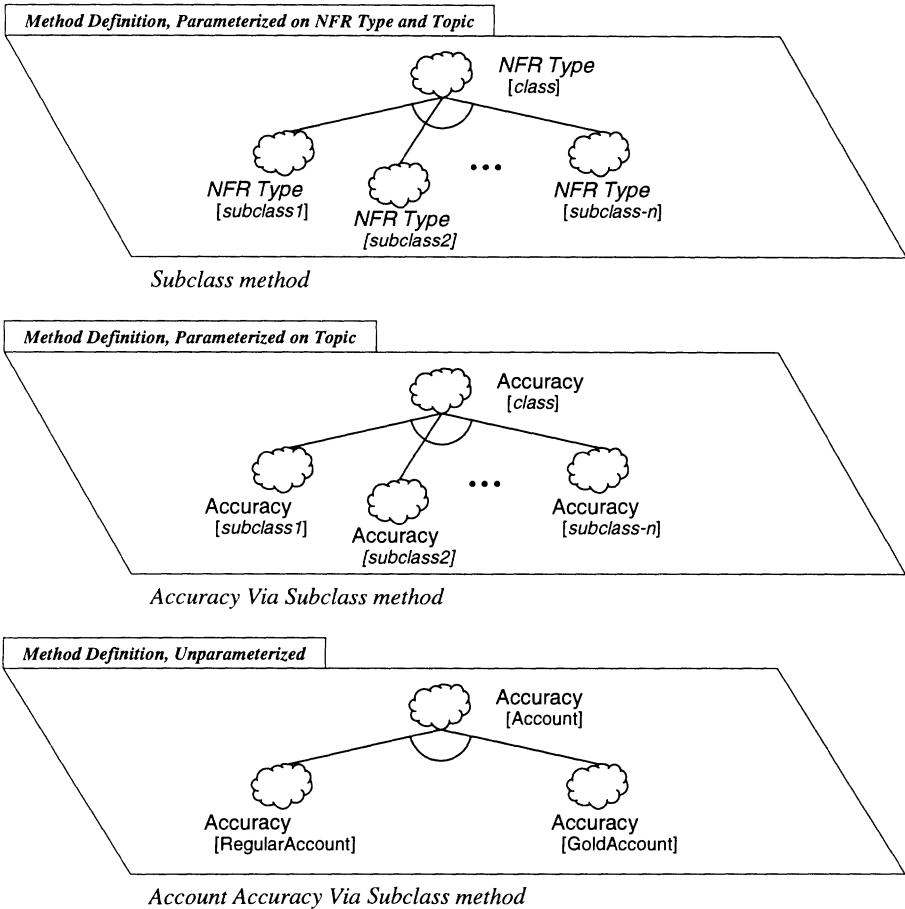


Figure 4.15. The Subclass topic decomposition methods, with varying degrees of parameterization.

Figure 4.15 shows a family of Subclass methods. The three topic decomposition methods have an increasing degree of parameterization. At the top layer is the very generic Subclass method, which is parameterized on both type and topic. This is specialized in the middle layer by fixing the type as Accuracy to produce the AccuracyViaSubclass method, which is parameterized on topic. In turn, this is specialized by fixing the topic as Account to produce the unparameterized AccountAccuracyViaSubclass method at the bottom layer.

Another example of a fairly generic topic decomposition method is the Attribute method, shown in Figure 4.16. This draws on data models having

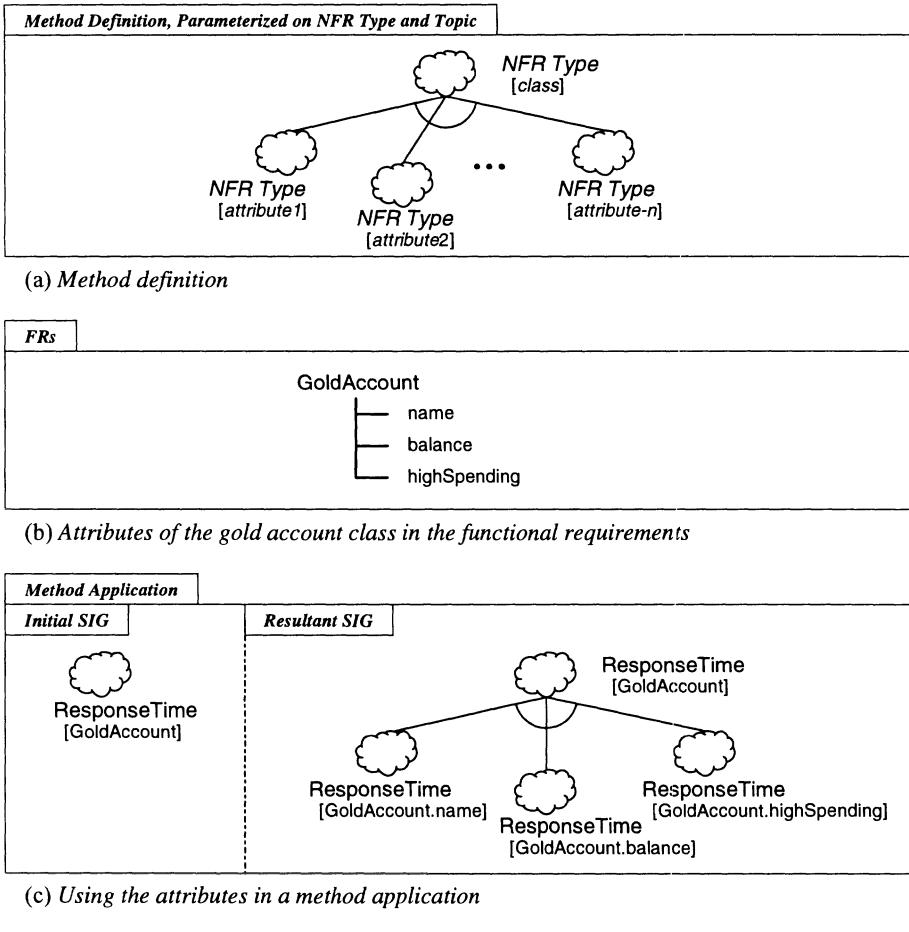


Figure 4.16. The Attribute decomposition method and one of its applications.

classes with attributes. This method takes a parent of any type, with a class as topic, and generates a set of offspring, each having as its topic one of the attributes of the parent topic. The figure shows how to accomplish this by an application of the method to response time for gold accounts.

Gold card accounts have an attribute `highSpending`, in addition to `name` and `address` which `GoldAccount` inherits from `RegularAccount`. In the functional requirements in Figure 4.16, attributes are shown underneath and to the right of their class. Attributes of a class can also be written using a dot notation in the form `class.attribute`, e.g., `GoldAccount.balance`.

The **Attribute** method can be applied to any softgoal having any type and any topic, provided the topic has attributes. For example, the method could be applied to softgoals such as **Flexibility[UserInterface]**, **Space[RegularAccount]**, **Accuracy[GoldAccount]** or **OperatingCost[Staffing, Under \$100 000]**. Applying the **Attribute** method to **Accuracy[GoldAccount]** would require that all attributes of gold accounts be maintained accurately, in order for **Accuracy[GoldAccount]** to be satisfied.

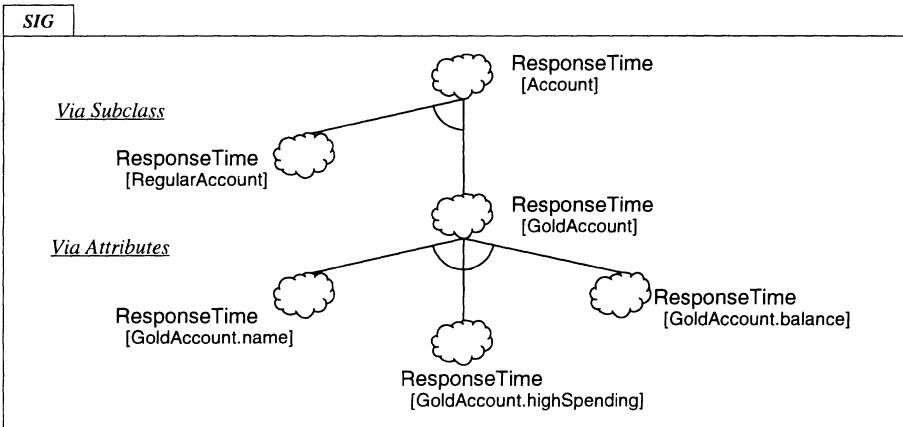


Figure 4.17. A Softgoal Interdependency Graph with two method applications.

Figure 4.17 illustrates how the developer could use two decomposition methods, first the subclass method, followed by the attribute method, to refine the top-level softgoal **ResponseTime[Account]**. In SIGs, method names (possibly abbreviated) are in italics and underlined.

In using a method, whether decomposing on type or topic, it should be noted that there may be more than one decomposition possible. For example, if the **Subclass** method is applied to a softgoal whose topic involve students, the result could be two subgoals involving graduate and undergraduate students, or it could be two subgoals involving full-time and part-time students. Multiple decompositions are also possible for **SubType**, **Subset** and other methods. In such cases, developers would have to choose what is suitable, using domain information and their expertise.

We have described NFR decomposition methods, which decompose NFR softgoals into other NFR softgoals. The same basic mechanisms are also used for operationalization decomposition methods (which refine operationalizing softgoals to operationalizing softgoals) and argumentation decomposition methods (which refine claim softgoals to claim softgoals). However, the meaning of the

methods will differ, reflecting the different kinds of softgoals that are decomposed.

4.3 OPERATIONALIZATION METHODS

In due course the developer will have satisfactorily refined the initial non-functional requirements in terms of more detailed ones, using a catalogue of methods, and the developer's expertise and own methods.

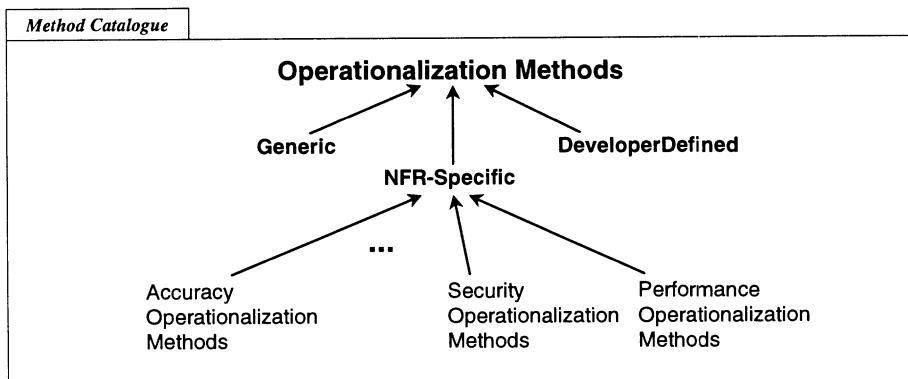


Figure 4.18. A catalogue of operationalization methods.

The developer can now move on to trying to satisfice the NFR softgoals. As with decomposition methods, the know-how for satisficing non-functional requirements can also be captured and encoded as methods, and compiled into a catalogue. This way, the time and effort to search for relevant know-how can be reduced, and a catalogue of such know-how is made widely available and reusable.

Figure 4.18 shows a *catalogue of operationalization methods*. More general methods are shown above more specific ones. As with NFR decomposition methods (Figure 4.2), operationalization methods include generic, developer-defined, and NFR-specific methods. Operationalization methods for specific NFRs will be presented in Part II.

Operationalization methods refine softgoals into operationalizing softgoals. Operationalizations are candidates for representing entities, activities and constraints. They represent possible design or implementation components.

One group of operationalization methods refines NFR softgoals into operationalizing softgoals. Another group, operationalization decomposition methods, decompose operationalizing softgoals into more specific operationalizing softgoals. This later kind of method is helpful because operationalizing soft-

goals can be coarse-grained and there can be more than one way to realize them in the target.

Thus, operationalization methods take either an NFR softgoal or an operationalizing softgoal, and generate one or more operationalizing softgoals. We now discuss these two kinds of operationalization methods.

Operationalization of NFR Softgoals

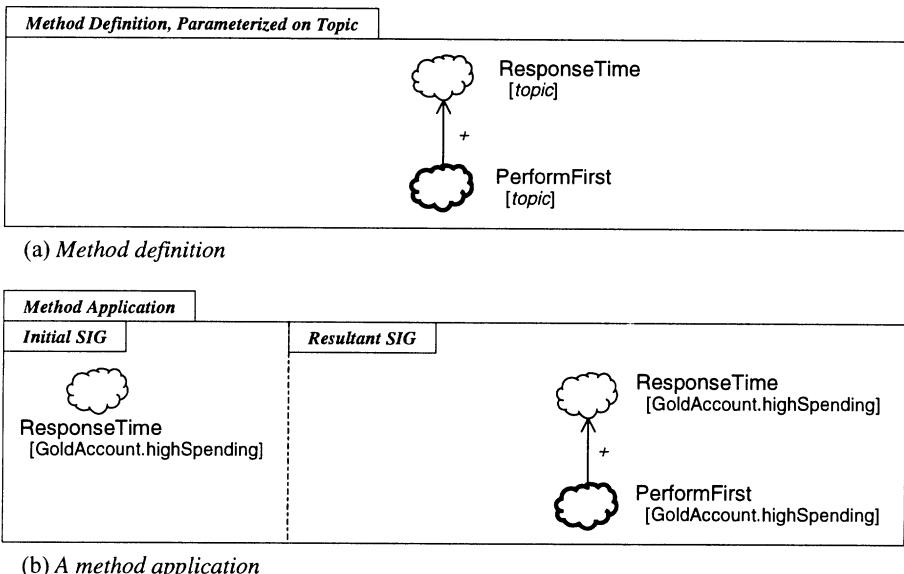


Figure 4.19. The `PerformFirst` method, applied to an operation on an attribute.

Returning to our credit card example, we started with NFRs for accuracy and performance of accounts. After decomposition, we have some specific NFR softgoals, such as ones for the accuracy and response time for the `highSpending` attribute of gold accounts. We now want to operationalize these NFR softgoals.

Let's consider the NFR softgoal `ResponseTime[GoldAccount.highSpending]`. One method of providing good response time is to order operations so that operations relating to high spending are performed first. This operationalization method is shown in Figure 4.19. The operationalizing softgoal is `PerformFirst[GoldAccount.highSpending]`. It *HELPS* satisfy the response time softgoal. Note that the type of an operationalizing softgoal is not an NFR type, such as `Accuracy` or `Performance`. Instead, it represents a development technique used in the target design or implementation to help achieve NFRs. Here the

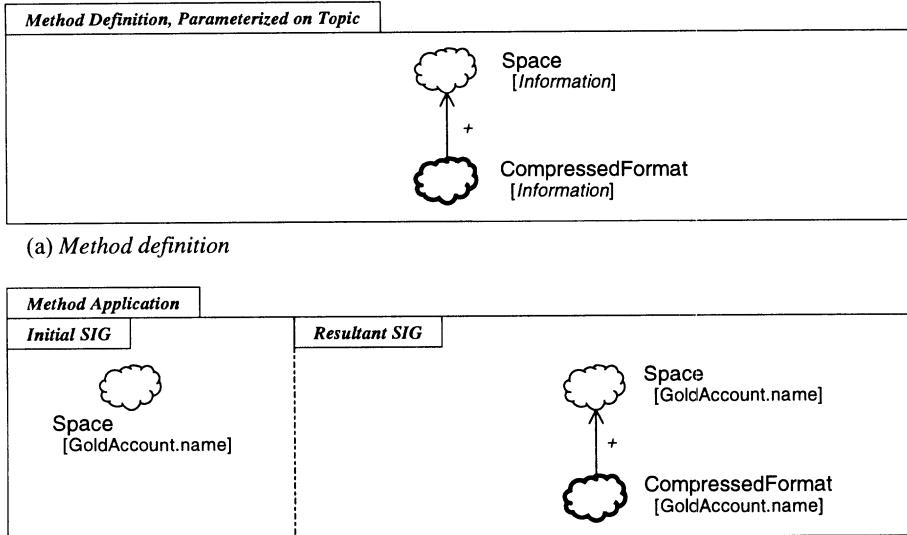


Figure 4.20. The CompressedFormat operationalization method and an application.

operationalization type is **PerformFirst**. Often the type of the operationalizing softgoal is used to name the operationalization method.

The method application can be summarized by stating the contribution of the operationalizing softgoal (the offspring) to the NFR softgoal (the parent):

PerformFirst[GoldAccount.highSpending] *HELPS*
ResponseTime[GoldAccount.highSpending]

Similarly, the method definition can be summarized:

PerformFirst[topic] *HELPS* **ResponseTime[topic]**

Here *topic* is a parameter, shown in sans serif italics. Where there is no ambiguity, softgoal topics may be omitted, to provide a briefer summary of the method:

PerformFirst *HELPS* **ResponseTime**

In this example, we have not specified a particular operation to be performed on the **highSpending** attribute, and to be performed first. This can also

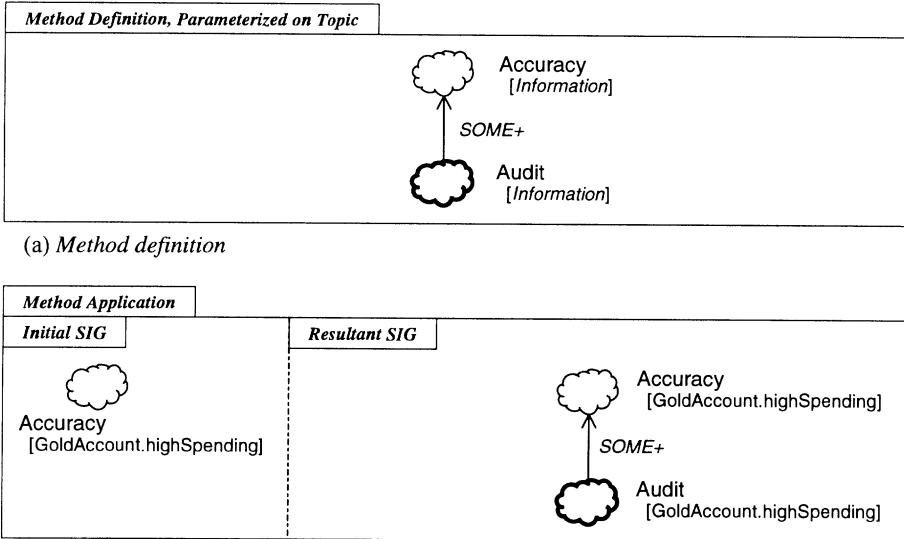


Figure 4.21. The Auditing method and one of its applications.

be done by specifying information in the topics of softgoals. For example:

```
PerformFirst[retrieve(GoldAccount.highSpending)] HELPS
ResponseTime[retrieve(GoldAccount.highSpending)]
```

Let's consider how we could help meet a requirement for good space performance for the accountholder's **name**. To help reduce the space usage, we can use a compressed format for storing the name. This is shown in Figure 4.20. While **CompressedFormat** **HELPS Space**, it will also hurt response time, because the information will have to be uncompressed before being used. We will deal with this kind of tradeoff later.

Continuing with our example, let's consider the accuracy of information about various high spending in gold card accounts, i.e., **Accuracy [GoldAccount.highSpending]**. Periodic auditing of those databases will make *some positive contribution* to ensuring the accuracy of this information, if the high spending record of each gold account is obtained from existing databases. This knowledge can be encoded as an operationalization method, shown in Figure 4.21 and summarized as **Audit SOME+ Accuracy**.

Given an accuracy softgoal, the **Audit** method will generate one offspring softgoal whose topic is identical with the parent's topic. The offspring's type is

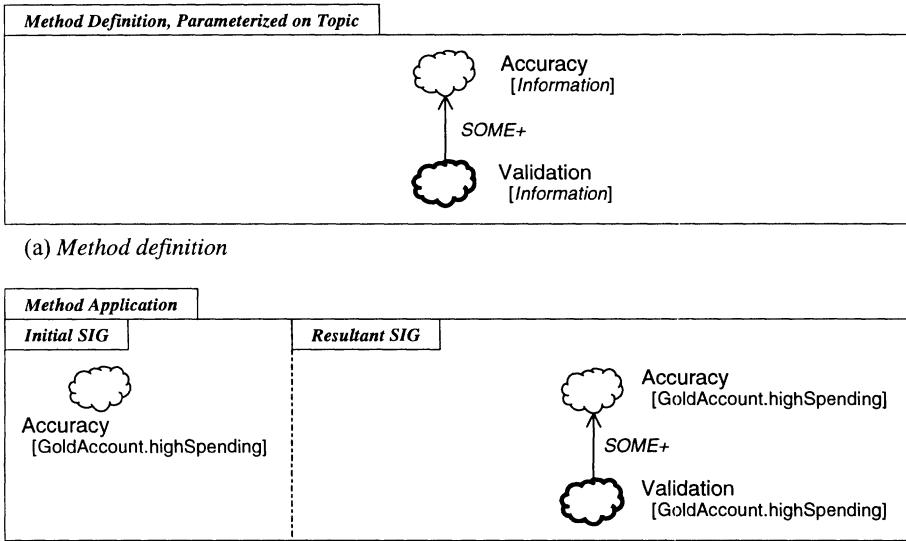


Figure 4.22. The Validation method and one of its applications.

Audit. The contribution type is *SOME+* which either *HELPS* or *MAKES* the parent.

There are some restrictions for this method. As an **applicabilityCondition**, the topic must be an information item. As a constraint, the topic must be in the database.

If, on the other hand, information about various high spending reports are fed directly by an employee of the credit card organization in question, a method may call for the validation of the information by the employee's manager. Figure 4.22 shows the definition of this method and one of its applications. Validation gives some positive contribution to Accuracy.

Later, the developer may change the contribution types. For instance, the contribution of Validation is changed from *SOME+* to *MAKES* in Figure 4.25, when the design has proceeded further and it can be determined that validation indeed leads to more accurate high spending data.

As a result of applying the Audit and Validation operationalization methods, each operationalization would give some positive contribution to Accuracy [GoldAccount.highSpending].

Clearly, selection of one of the two alternatives leads to very different kinds of user interfaces for the system under development. In particular, if validation is selected, all high spending information will have to be confirmed

by another person, while auditing calls for the inclusion of an audit requirement on the database from which high spending data are imported.

Decomposition of Operationalizing Softgoals

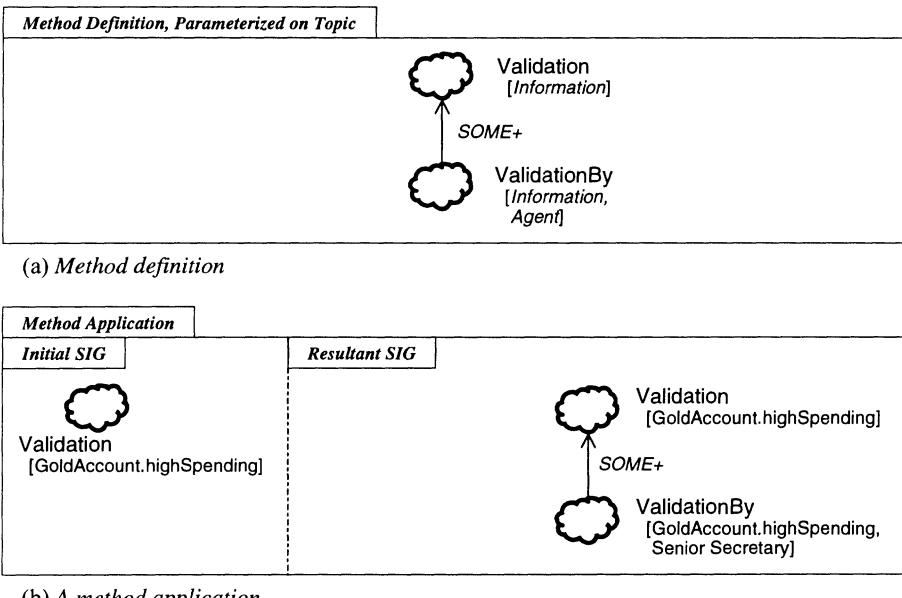


Figure 4.23. Refining an operationalizing softgoal and adding a topic.

Let's turn to operationalization decompositions, which refine operationalizing softgoals into other operationalizing softgoals. This kind of refinement can be used to add detail, to focus on a particular aspect, or to disambiguate notions. This kind of method can refine the type or topic of an operationalizing softgoal.

In our example, we continue considering validation of high spending for gold accounts. So far, the validation task has not been assigned to appropriate people. This is done by using the `ValidationBy` method (Figure 4.23). This method is a kind of topic refinement. It adds a second topic to the softgoal, which indicates who will do the validation. (The name of the type is slightly changed from `Validation` to `ValidationBy` to indicate that the later uses a validator.)

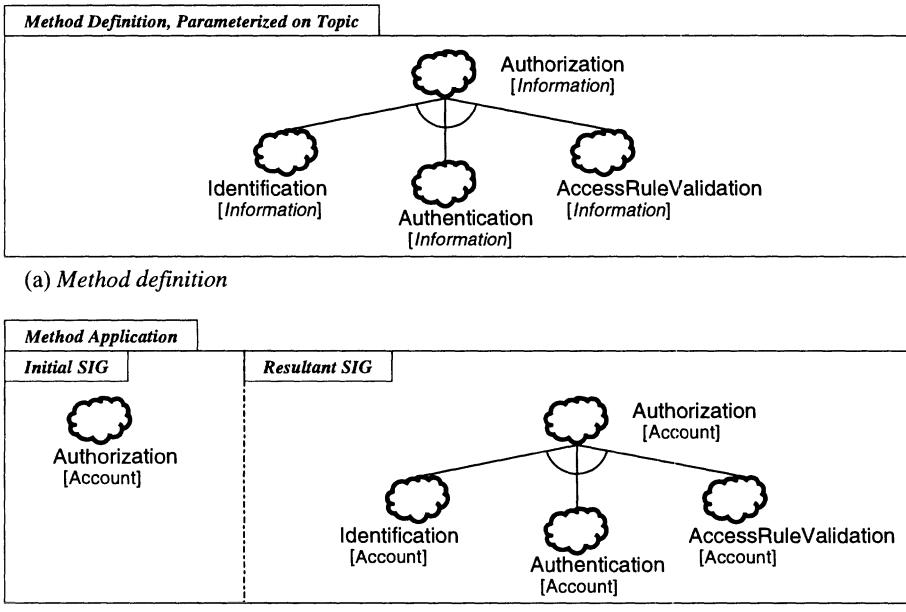


Figure 4.24. Applying the AuthorizationViaSubType method to accounts, resulting in several offspring of an operationalizing softgoal.

The method can be summarized as:

ValidationBy[*Information, Agent*] *SOME+* Validation[*Information*]

It can also be defined in a frame-like notation:

```

OperationalizationMethod ValidationBy
  parent: Validation[1]
  offspring: ValidationBy[i', Agent]
  contribution: HELP
  applicabilityCondition: i subsetOf i'
  constraint: /* Developer specifies Agent */
  
```

The **constraint** has the developer specify a particular agent, such as a senior secretary, to perform the validation. Note that this is an example of the developer's involvement in generating offspring softgoals *even after* a method has been selected. Before proceeding to other refinements, the developer could consult domain experts to assign an appropriate agent.

Operationalization decomposition methods can also refine the type of an operationalizing softgoal. Figure 4.24 shows the refinement of an Authorization operationalizing softgoal into more specific aspects of authorization. Here the types of the offspring are different from each other and from the parent. The types of the offspring are taken from a type catalogue. All three operationalizations must be done in order to satisfy Authorization.

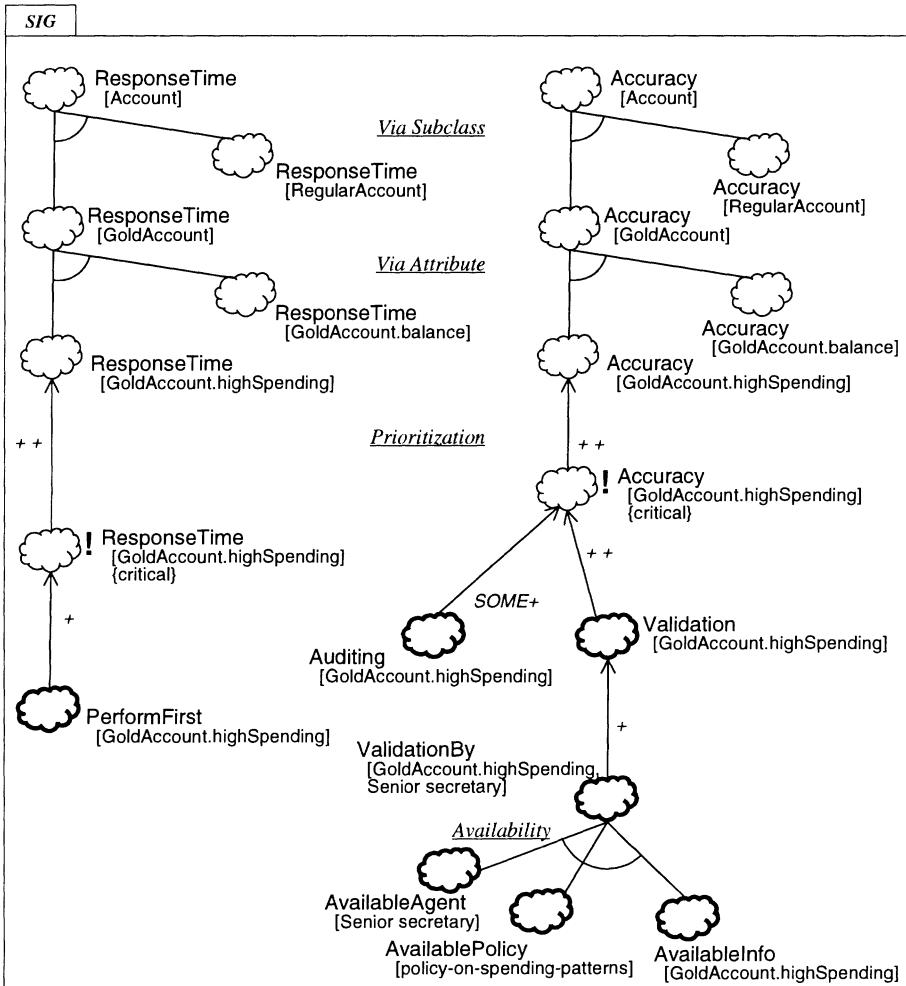


Figure 4.25. A Softgoal Interdependency Graph with applications of various decomposition and operationalization methods.

As another example, the bottom of Figure 4.25 shows the application of the Availability method to *ValidationBy[GoldAccount.highSpending, SeniorSecretary]*. To perform the validation, three resources should be available: information about the topic, a policy which states predetermined standards on permissible values of the topic, and an agent to perform the task.

Figure 4.25 illustrates a softgoal interdependency graph that could be generated by applying some of the operationalization methods we have presented. Method names are shown in SIGs in underlined italics. However, method names are omitted when they are similar to the type of the offspring operationalizing softgoal.

Note also that the developer has changed the contribution of *Validation* from *SOME+* to *MAKES*. Here, the developer feels that validation will suitably ensure accuracy. This decision could be made, for example, on the basis of the information which should be accurate, and the needs and characteristics of the particular domain for which the system is being built. This is an example of developers stepping in and using their expertise and knowledge of the domain.

In the current example, only one of auditing and validation is likely to be adopted for the satisficing of *Accuracy[GoldAccount.highSpending]*. However, there can be a large number of operationalization methods that are applicable to a non-functional requirement, all at the same time. If so, it is up to the developer to examine what impact such methods have on other types of non-functional requirements (e.g., cost and usability) and decide on what and how many operationalizing methods to apply.

4.4 ARGUMENTATION METHODS AND TEMPLATES

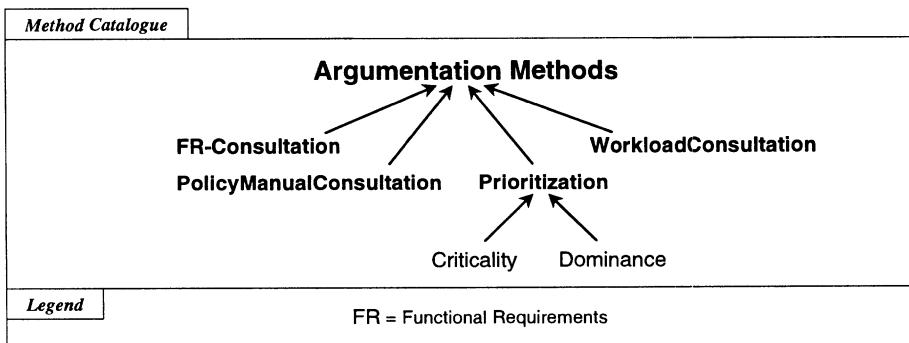


Figure 4.26. A catalogue of argumentation methods.

Claims are used to incorporate *domain characteristics* into the decision-making process. This includes information about the organization which will use the system, such as organizational priorities (e.g., “we value confidentiality and speed”), and organizational workload (e.g., the number of credit card transactions per day). It also includes information about the functionality of the system. Claims also incorporate *developers’ expertise*, including knowledge of the domain, particular NFRs, and particular development techniques. Claims help justify a variety of design decisions, including the way the initial NFR softgoals are elaborated, prioritized, and satisfied.

Argumentation methods refine a softgoal into a claim softgoal. They frequently also refine an interdependency into a claim softgoal. Whether the parent is a softgoal or interdependency, argumentation methods are used to indicate evidence, or counter-evidence, for the satisficing of a parent.

As with NFR refinement methods and operationalization methods, we can catalogue know-how in argumentation methods. Once catalogued, such know-how for making and justifying design decisions can be made available for re-use.

Figure 4.26 shows a *catalogue* of general kinds of *argumentation methods*. Several of them involve consulting other documents, which can be used as a reference source for claims that are made. For example, claims referring to the functionality of the system can involve consultation of the functional requirements. Workload-based claims can also be made, by consulting appropriate information. Claims referring to how users will use the system can involve consultation of operations policy manuals. Finally, argumentation is frequently used for prioritization, which we will discuss shortly.

Because argumentation often draws on domain information, it is not always possible to provide complete argumentation method definitions which capture domain details. Therefore, we use *claim templates* to provide outlines of argumentation methods. However, the developer must provide some details for the offspring. Unlike complete method definitions, these details cannot be determined automatically, by examining the parent and the template (or definition) and then using syntactic substitution.

Argumentation methods and templates help the developer construct an *argumentative structure* as part of a softgoal interdependency graph. Arguments are applied to softgoals and contributions. In an argumentative structure, an argument can offer positive or negative support for a particular softgoal or interdependency. In turn, an argument can be supported or denied by other softgoals or interdependencies. This kind of power to organize design rationale stems from the ability of interdependencies to interrelate softgoals and interdependencies. This allows nested and complex interrelationships to be stated.

The simplest case of argumentation is where a claim is associated with a softgoal. For example, `PerformFirst[GoldAccount.highSpending]` is an operationalization for a priority NFR, providing good response time for the `high-Spending` attribute. By examining domain information and implementation techniques, the developer could argue that `PerformFirst`

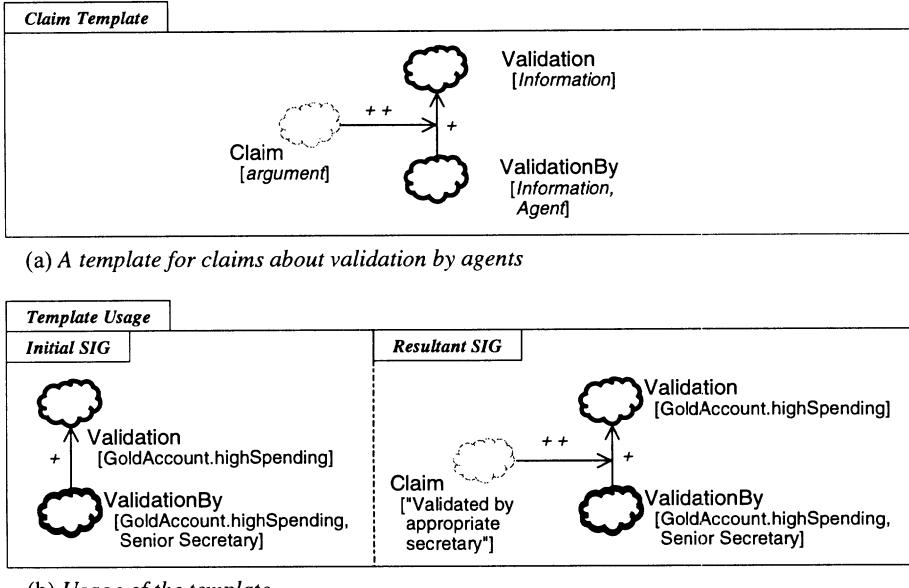


Figure 4.27. A template for claims about validation.

[GoldAccount.highSpending] is suitable because there is an implementation which allows priority arguments to be performed first. This claim provides a *MAKES* contribution to PerformFirst[GoldAccount.highSpending], because if the rationale of the claim is true, the operationalization will be suitable. This is shown at the left of Figure 4.28, later in this section.

More complex cases of argumentation associate a claim with an *interdependency link*. Let's consider the rationale for assigning a particular kind of employee to validating high spending in gold accounts. The developer assigned senior secretaries to this task. Now this is justified by claiming that validation should be performed by employees with appropriate seniority (Figure 4.27).

This is done by refining a parent *interdependency* to an offspring. Here the parent is the interdependency between ValidationBy [GoldAccount.highSpending, SeniorSecretary] and Validation [GoldAccount.highSpending]. This interdependency can be summarized as:

ValidationBy[GoldAccount.highSpending, SeniorSecretary]
HELPS Validation[GoldAccount.highSpending]

Here again, the developer has changed the contribution. Here, *SOME+* in the definition of the ValidationBy method is changed to *HELPS* here.

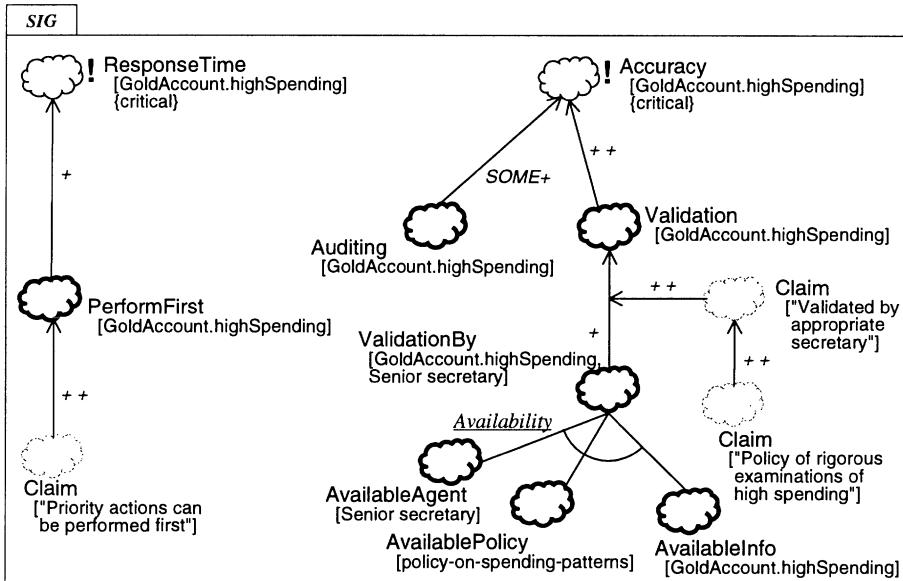


Figure 4.28. Claims about operationalization methods.

The offspring is *Claim*[“Validated by appropriate secretary”]. This provides a *MAKES* contribution to the parent interdependency. This means that if the claim is substantiated, it will justify making the refinement from *Validation*[*GoldAccount*.*highSpending*] to *ValidationBy*[*GoldAccount*.*highSpending*, *SeniorSecretary*]. This can be written as:

```

Claim      [“Validated by appropriate secretary”]
MAKES
  ( ValidationBy[GoldAccount.highSpending, SeniorSecretary]
    HELPS Validation[GoldAccount.highSpending])

```

This template can also be written in the frame-like notation:

```

ArgumentationTemplate ValidatorAssignmentJustification
  parent: ValidationBy[Information, Agent] HELPS
    Validation[Information]
  offspring: Claim[argument]
  contribution: MAKES
  applicabilityCondition: Information: InformationItem
  constraint: /* argument is about the seniority of the Agent

```

performing validation of Information */

Note that the parent is an interdependency. Also note that this is a template, not a full method, because the developer must provide the details of the argument.

This claim about who does validation can in turn be supported by argumentation. A company policy of carefully examining high spending is used to support the claim about validation being done by an appropriate secretary. This is an example of a claim supporting another claim. This *argumentation decomposition* is shown at the right side of Figure 4.28, which illustrates some of the argumentation used in this section.

Prioritization

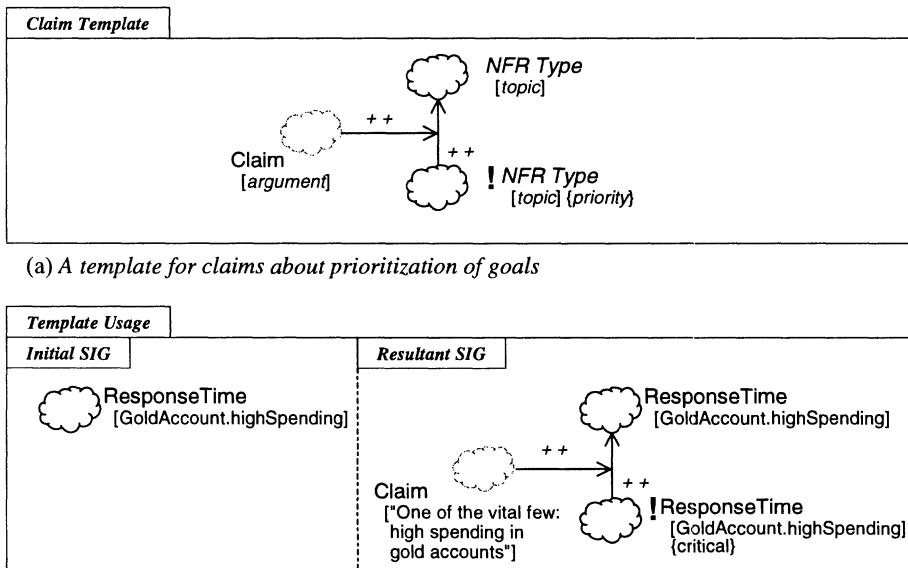


Figure 4.29. A template for the VitalFew argumentation method for prioritization, and an example of its usage in a claim.

An argumentation method can be used to capture generic knowledge about setting priorities among alternatives. One such *prioritization method*, is the **VitalFewTrivialMany** method. This argumentation method allows for putting more emphasis on important softgoals (the “vital few”) during refinement, conflict resolution, and selection among competing alternatives.

Figure 4.29 shows a template for the **VitalFew** argumentation method, which is used to identify a softgoal as a priority softgoal. It also shows a sample usage of the template. We start with a parent softgoal, such as **ResponseTime[GoldAccount.highSpending]**, and produce an offspring with the same type and topic, but identified as a priority using an exclamation mark. Since satisfying the priority offspring will satisfy the parent, they are inter-related by a **MAKES** contribution. For example,

```
!ResponseTime[GoldAccount.highSpending]    MAKES
ResponseTime[GoldAccount.highSpending]
```

To justify this decomposition, a claim is made. For example, the claim could explain why high spending in gold accounts is a priority item. The claim provides a **MAKES** contribution to the *interdependency* between the parent and offspring softgoals. This means that *if* the claim is satisfied, then the *interdependency link* (here, from **!ResponseTime[GoldAccount.highSpending]** to **ResponseTime[GoldAccount.highSpending]**) itself becomes satisfied. In this way, claims can support *softgoal refinements*.

Note that in using the template, we can in part use automatic substitution: the type and topic of the parent given the type and topic of the offspring; this is like the substitution used in methods. However, the details of the *argument* of the **Claim** are not given by the template. Instead, they must be provided by the developer, using domain information or other expertise.

The result of all this is that the developer will give higher priority to **!ResponseTime[GoldAccount.highSpending]**. Other softgoals will not receive such treatment. There is also a **TrivialMany** method which indicates that a softgoal is not a priority. These prioritization methods are used throughout this book.

Let's look at prioritization in more detail, especially when parts of an **AND** decomposition are prioritized. When priority offspring have been satisfied, but the non-priorities are denied, we may want to consider the parent to be satisfied. To do so, we present mechanisms which in effect modify **AND** contributions.

Consider the “Initial SIG” of Figure 4.30. We have refined an NFR softgoal for response time of accounts into two offspring, one for gold accounts, the other for regular accounts. The offspring make an **AND** contribution to the parent. Suppose that the gold account softgoal can be satisfied, and is a priority. On the other hand, the regular account softgoal is not a priority, and can't be satisfied. In cases where we have met the priorities, we may want to consider the overall requirement met [Juran64] [C. Smith86].

We need to deal with a technical problem. Using the evaluation rules for **AND** contributions (Figure 3.16), if **ResponseTime[RegularAccount]** is denied, the parent **ResponseTime[Account]** will always be denied.

To avoid this, we apply a prioritization method to each offspring. On the left sides of the “Claim Template” and the “Resultant SIG” of Figure 4.30, we apply the **VitalFew** method to the priority offspring, and provide an appropriate claim. As a result, we have a priority offspring, e.g., **!ResponseTime**

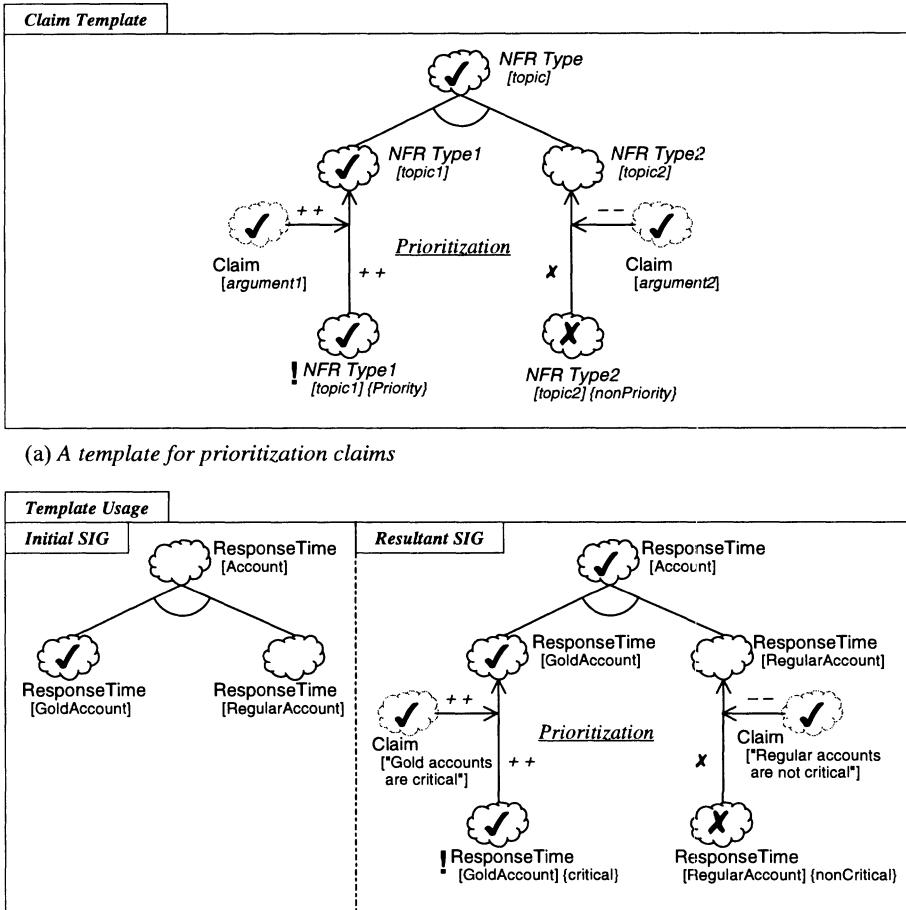


Figure 4.30. A template for claims about prioritization.

[GoldAccount]{critical}. As before, the priority softgoal *MAKES* its parent, and the claim *MAKES* the interdependency, which goes from the priority softgoal to its parent. The result of satisfying the claim is to justify the refinement from the NFR softgoal to the prioritized one.

On the right side, we apply the TrivialMany method. An offspring is generated which is identified as being a non-priority. In the example, we produce ResponseTime[RegularAccount]{nonCritical}. In addition, a claim is provided to explain why the softgoal is not a priority. In the example, we argue that regular

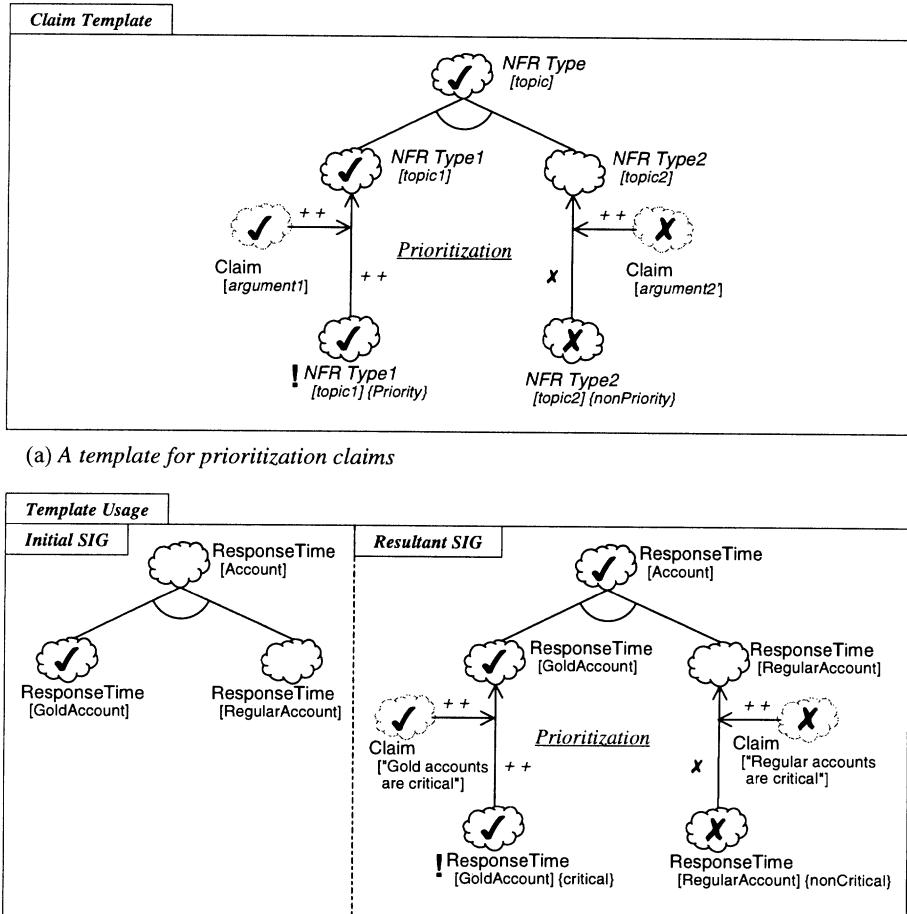


Figure 4.31. Another template for claims about prioritization.

accounts are not critical. Unlike the VitalFew method, the claim *BREAKS* the interdependency, which has a contribution from the non-priority softgoal to its parent. The result of denying the claim is to *deny the interdependency*. This is indicated by a small “ \times ” on the right interdependency link. As a result, the offspring, here $\text{ResponseTime}[\text{RegularAccount}]\{\text{nonCritical}\}$, makes *no contribution* to the parent. Interestingly, this is true regardless of value of the offspring; here it is denied, as would often be the case for non-priorities. Importantly, the parent, here $\text{ResponseTime}[\text{RegularAccount}]$, does not make any contribution to its

parent, here **ResponseTime[Account]**. As a result, the satisficing of the overall softgoal, here **ResponseTime[Account]**, will depend only on its other offspring. If that other offspring is satisficed (and here **ResponseTime[GoldAccount]** is), then the overall softgoal will also be satisficed.

This is consistent with the principle that satisficing the priorities may be enough to satisfy the overall requirement. This template for prioritization among *AND* offspring will be frequently used in this book.

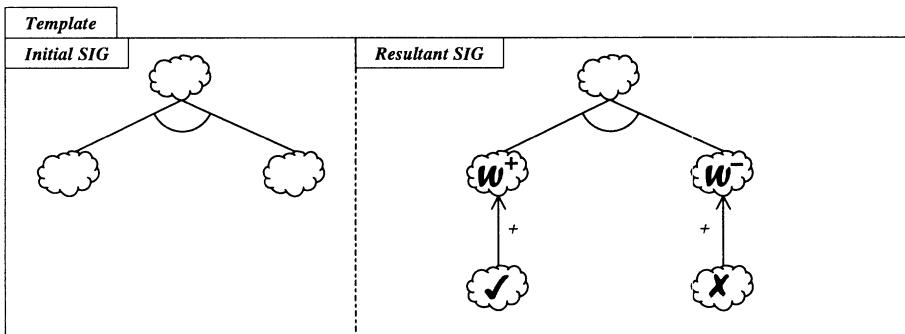


Figure 4.32. Modifying the offspring labels of an *AND* interdependency.

Figure 4.31 provides another template for prioritization of *AND* offspring. It obtains the same net effect as Figure 4.30, but accomplishes it in a slightly different way. The **VitalFew** prioritization on the left works exactly the same way as in the previous figure. However, the claim for the “trivial many” case is the opposite of the previous figure. Instead of satisficing a claim that the right offspring is not critical, we *deny* a claim that it *is* critical. In the “Claim Template” of Figure 4.31, *argument2*’ does *not* hold; it is the opposite of *argument2* of Figure 4.30. In the example, regular accounts are still not critical. In Figure 4.31, we make the opposite claim, i.e., “**Regular accounts are critical**” and then immediately deny the claim (indicated by “ \times ” in the claim softgoal on the right). This claim *MAKES* the interdependency (which runs from the non-priority softgoal to its parent). The result of denying the claim, and its *MAKES* contribution to the interdependency, is exactly the same as in Figure 4.30: the interdependency is denied, and we proceed as before.

The developer can use similar techniques to obtain flexibility to adjust labels and contribution values to deal with particular situations. This can be done, for example, to strengthen or weaken particular values. This can also be useful when combining values. We describe some templates to achieve this kind of flexibility.

In Figure 4.32, for example, we start with an *AND* interdependency. The developer may wish to adjust label values coming into the offspring. This can

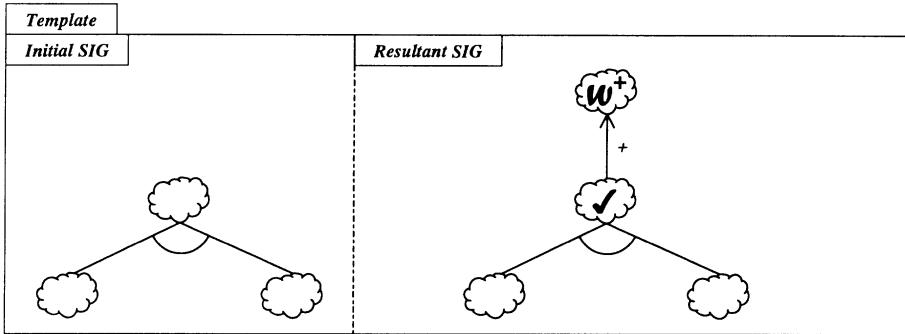


Figure 4.33. Modifying the parent label of an *AND* interdependency.

be done by giving each offspring its own offspring, each connected by *HELPS* contributions. With this contribution type, the effect is to weaken inputs. Here a satisfied offspring-of-an-offspring results in a weak positive (W^+) offspring. Similarly, a denied offspring-of-an-offspring results in a weak negative (W^-) offspring. Other effects can be achieved using other contribution types. The overall effect is to allow “inputs” to *AND* interdependencies to be adjusted. We say this two-step interdependency has the *HELPS-AND* combined contribution.

Likewise the “output” of an *AND* interdependency can be adjusted. In Figure 4.33, the parent of the *AND* interdependency is given a parent. Here it is given a *HELPS* contribution, which generally weakens the values. We say that this two-step interdependency has the *AND-HELPS* contribution type. Again, other contribution types will have different effects.

These types of adjustments can be useful, for example, when the developer does not have full confidence in the use of a method, or in the satisficing of offspring softgoals. In such cases it might be helpful to weaken output or input values.

Figure 4.34 shows a SIG with some examples of the argumentation we have been discussing. Note that the claim about high spending is used to prioritize two NFR softgoals, involving **ResponseTime** and **Accuracy**. The figure is an extension of Figure 4.25, which shows the usage of operationalization methods.

In SIGs, if the contribution of a claim is not indicated, it is *MAKES* by default. For other interdependencies, the default contribution is *HELPS*.

We have discussed prioritization of NFR softgoals. However, it is also possible to prioritize operationalizing softgoals and claim softgoals. Prioritization of operationalizing softgoals helps a developer indicate that some are more important than others, and should be given more attention than low-priority

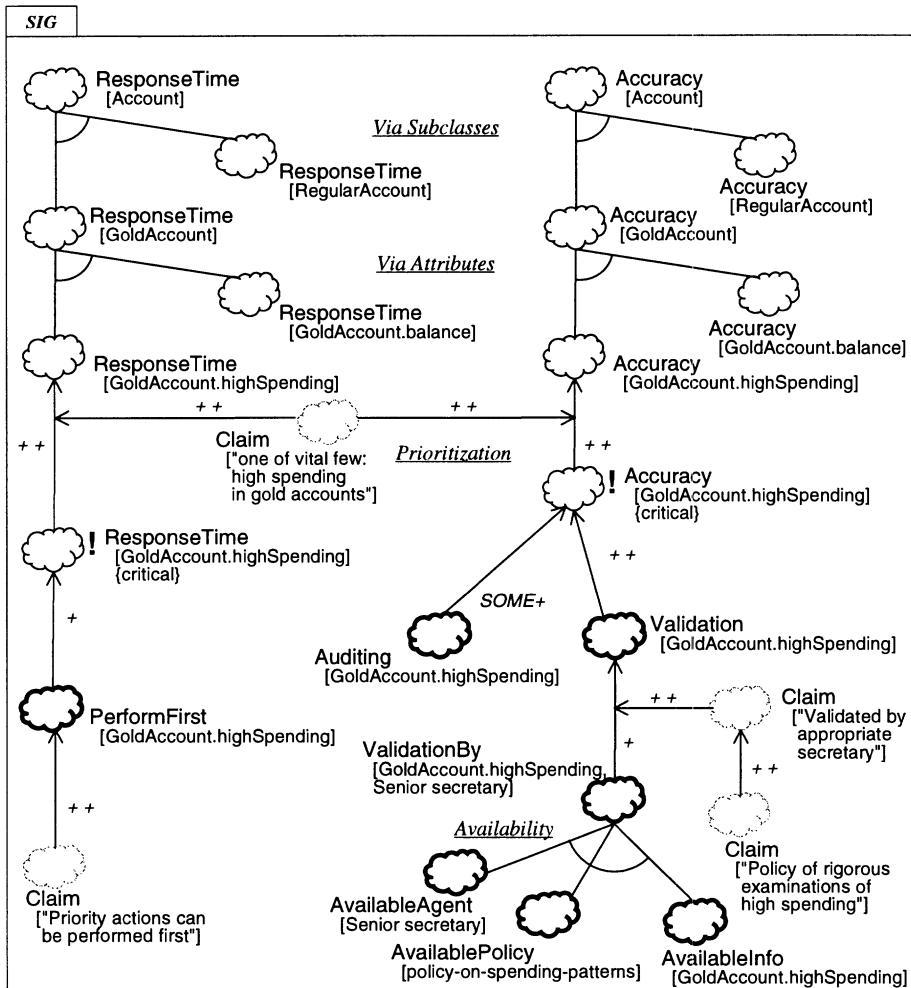


Figure 4.34. A SIG with operationalization and argumentation methods.

ones. Similarly, prioritization of claim softgoals can be used to indicate that some evidence is considered more important than others.

4.5 CORRELATIONS

The non-functional requirements set down for a particular system may be in conflict or in harmony with each other. For instance, “faster” (production time)

may be in conflict with “cheaper” (cost), if it involves hiring more software engineers. On the other hand, “better” (software) may be harmonious with “faster,” if “better” (software) significantly reduces the time needed for testing and correction of defects.

In dealing with non-functional requirements in software engineering, it is important to address this kind of conflict and harmony. This helps us deal with tradeoffs as well as synergy (mutual helpfulness) during system development.

However, we have to *detect* conflict and harmony. To do so, we introduce an additional mechanism. The reason is that for a given set of softgoals, there can be more interdependencies than those that have already been explicitly stated, e.g., via usage of methods.

In the NFR Framework, conflict is represented by *negative correlations* and harmony is represented by *positive correlations*.

During the process of software development, *interactions* between softgoals are discovered and noted in a softgoal interdependency graph. Such discoveries can be made “on-the-fly” or with the assistance of *correlation rules*.

Just like methods, correlations allow for capturing knowledge about generic interactions between softgoals, for encoding such knowledge, and for compiling it into a catalogue. Initially collected from the literature and industry experience, correlation rules then can easily be shared, extended, tailored, and reused.

Unlike methods, however, correlations are not usually selected and applied explicitly by the developer during the development process. Instead, they are usually detected *implicitly*. They typically arise from changes in SIGs, such as the addition of a softgoal. These changes can be thought of as “triggering” a correlation.

Thus an interdependency generated as the result of a method application is termed an *explicit interdependency*. On the other hand, an interdependency contribution inferred from a correlation rule is termed an *implicit interdependency*.

Correlations can be detected either by hand, or with the aid of a tool. In either case, this detection is done by comparing portions of a SIG to patterns in a *catalogue of correlations*. It is then up to the developer to decide whether to incorporate the detected interdependencies into the softgoal interdependency graph.

Detecting negative correlations can be viewed as revealing the (hidden) tradeoff of a development technique — it can be good for one softgoal but at the same time bad for another.

When selecting among competing operationalizing softgoals, correlation rules about interactions between such softgoals and NFR softgoals can serve as a basis for performing tradeoff analysis. Naturally, the developer might want to select those operationalizing softgoals that yield the most benefits and the least sacrifices, where possible.

Returning to our credit card account management system, let’s consider the use of a compressed format to store information, such as the `highSpending`

attribute of gold accounts. Unfortunately, using this operationalizing softgoal causes response time to deteriorate. This is because information must not only be retrieved, but also uncompressed before usage. This can be expressed by a *correlation rule*:

CompressedFormat[Information] HURTS ResponseTime[Information]

where *Information* is an information item (such as a class, or an attribute of a class). The gist of correlation rules can often be given by omitting the softgoal topics, where unambiguous, e.g.: **CompressedFormat HURTS ResponseTime**.

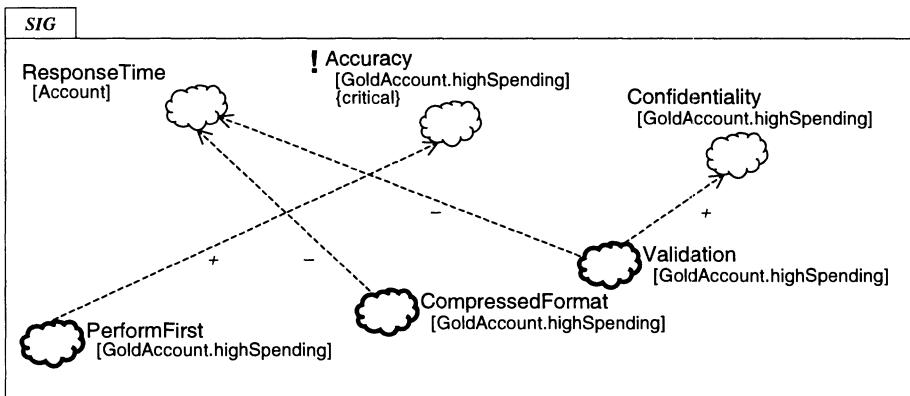


Figure 4.35. Sample correlations.

The effect of applying the correlation rule to the *highSpending* attribute would be a *negative implicit interdependency*:

**CompressedFormat[GoldAccount.highSpending] HURTS
ResponseTime[GoldAccount.highSpending]**

Figure 4.35 illustrates the use of some of the correlation rules in this section. Correlation links are shown as dashed lines.

As another example, a developer might allow access to a database using a flexible user interface. This could hurt accuracy, which is a priority NFR softgoal, especially if there are several infrequent users. Suppose that 5 employees is the acceptable limit. This can be written by attaching a *condition* to the correlation rule:

**FlexibleUserInterface[Employee, Information] HURTS Accuracy[Information]
WHEN cardinality(Employee) > 5**

Here, `Employee` is a class of employees. The *condition* follows the keyword `WHEN`.

When we apply the rule to infrequent users accessing the `highSpending` attribute, we have the following *negative implicit contribution*:

```
FlexibleUserInterface[InfrequentUser, GoldAccount.highSpending] HURTS
!Accuracy[GoldAccount.highSpending]{critical}
```

Correlation rules can also be written in a frame-like notation. For example, the rule concerning flexible user interfaces can be written:

```
CorrelationRule FlexibleUserInterfaceHURTSAccuracy
  parent: Accuracy[Information]
  offspring: FlexibleUserInterface[Employee, Information]
  contribution: HURTS
  condition: cardinality(Employee) > 5
```

A rule without a condition, such as the previous rule for compressed formats, is equivalent to a condition of “True”.

Usage of these and other correlations is shown in Figure 4.35. Note that correlations can be positive or negative. For example,

```
Validation[GoldAccount.highSpending] HELPS
Confidentiality[GoldAccount.highSpending]
```

is a positive correlation. In addition, one softgoal can participate in several correlations. Later figures in this chapter show a correlation as part of a larger SIG.

Let’s consider in more detail how correlations are used. Interestingly, one correlation rule can be used to make *inferences* in several ways. Let’s return to the correlation rule `CompressedFormat HURTS ResponseTime`. Figure 4.36 shows the correlation rule definition, followed by three different kinds of applications of the rule to accounts.

In part (b) of the figure, we start with two softgoals, `ResponseTime[Account]` and `CompressedFormat[Account]`, which are not connected. By comparing this “Initial SIG” with the Rule Definition in part (a) of the figure, we see that a match is possible, by substituting “`Account`” in the SIG for “`Information`” in the rule definition. As a result, the two softgoals, initially not connected, are now connected by an interdependency, here a *HURTS* contribution. The developer can now decide whether to use this interdependency.

In addition to generating interdependencies, correlation rules can also generate new softgoals. Consider the Initial SIG of part (c) of the figure, which has one softgoal, `Response Time[Account]`. Again using pattern matching, now with this SIG and the rule definition, `Response Time[Account]` is matched with a rule (pattern) where it is a “parent.” Then an offspring is inferred, along with its contribution to the initial softgoal. Here we infer `CompressedFormat[Account]` and its contribution of *HURTS*. We call this *downward inference*.

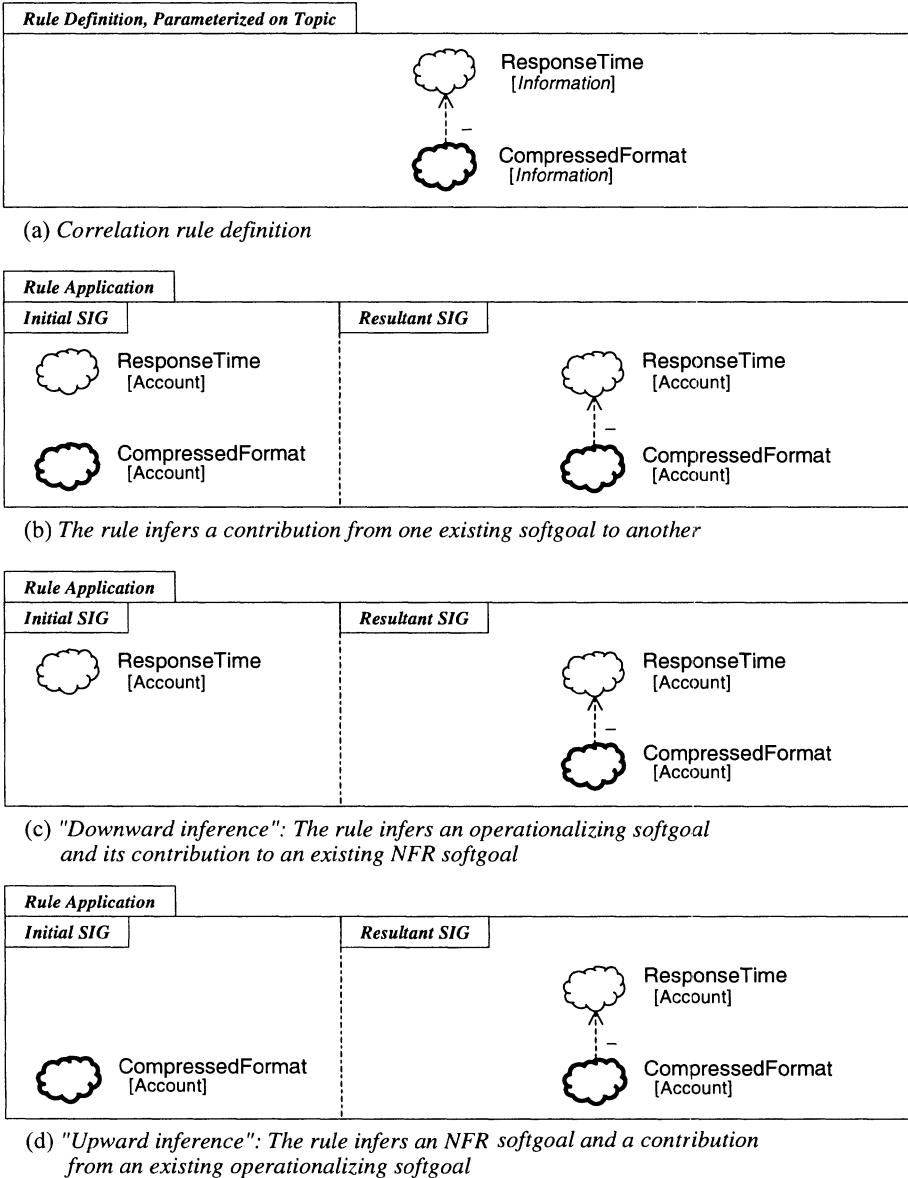


Figure 4.36. Different kinds of inferences which can be made by using a correlation rule.

We can also have inference in the other direction. In part (d) of the figure, the Initial SIG has one softgoal, `CompressedFormat[Account]`. This time, `CompressedFormat[Account]` is matched with a rule where it is an “offspring.” Then a parent is inferred, along with a contribution from the offspring (here, the initial softgoal) to the parent. Here we infer `ResponseTime[Account]` which receives a contribution of `HURTS` from `CompressedFormat[Account]`. We call this *upward inference*.

These kinds of uses of correlations are helpful in not only detecting negative correlations between softgoals that are already in the softgoal interdependency graph but also providing a warning to the developer against omission of those NFRs that the developer might not have addressed. These inferences can be confirmed by a domain expert, an analyst, or the developer.

Fairly general expressions can be used in a *condition* of a rule. This gives the flexibility to specify narrow or broad conditions for a rule to be applicable. For example, the relationships between different softgoal topics could be constrained, e.g., one information item would have to be a subset of another.

As another example, constraints in the domain may be used. For instance, certain classes of employees may be allowed access only to certain types of information. This could be written in a condition as: `accessAllowed(Information, Employee)`, where `accessAllowed` is not a softgoal, but a *descriptor* of situations in the domain. Suppose only senior secretaries had access to high spending information (written `accessAllowed(GoldAccount.highSpending, SeniorSecretary)`). Then a SIG involving senior secretaries might match the correlation rule with this condition, but a SIG involving junior secretaries would not.

Similarly, conditions could include descriptors of situations in the *system*, e.g., `infrequentReorganizations(Database)`. In this case, for example, a correlation rule involving implementation techniques might be matched only when the database is infrequently reorganized. Likewise, conditions could include descriptors of *workload*, e.g., `frequentAccess(Information)`.

Such *situation descriptors* can also be used in argumentation, e.g., `Claim[accessAllowed(GoldAccount.highSpending, SeniorSecretary)]`.

Correlation rules can be catalogued. Figure 4.37 shows a *correlation catalogue*. It summarizes in tabular form some of the correlation rules described in this section. The operationalizing softgoal (offspring) is in the left column, and the NFR softgoal (parent) is shown in the top row. The individual table entries show the *contribution* of the offspring to the parent, along with the *condition*, if any. One sample entry is `CompressedFormat[Information] HELPS Space[Information]`. This has no condition attached. Another sample entry, with a condition, is `FlexibleUserInterface[Employee, Information] HURTS Accuracy[Information]` WHEN `cardinality(Employee) > 5`. Details of conditions, often written using situation descriptors, are shown at the bottom of the figure.

The correlation catalogue helps the developer to examine cross-impacts among softgoals, during tradeoff analysis and subsequently during selection

Correlation Catalogue		to parent <i>NFR Softgoal</i>			
Contribution of offspring <i>Operationalizing Softgoal</i>		Accuracy [Info]	Confidentiality [Info]	Response Time [Info]	Space [Info]
Compressed-Format [Info]				HURTS	HELPS
Validation [Info]			HELPS	HURTS	
FlexibleUser-Interface [Employee, Info]		HURTS WHEN cond1			
PerformFirst [Info]		HELPS			

cond1: cardinality(Employee) > 5

Figure 4.37. A correlation catalogue.

among competing target alternatives. The table format helps keep things organized, even as the number of correlations grows.

We have only shown correlations involving a single offspring. It is also possible to have correlations involving *AND* and *OR* contributions. In this case, the table format for correlation catalogues would have to be modified to allow for more than one offspring.

Correlations can exist not only between an NFR softgoal and an operationalizing softgoal, but between two operationalizing softgoals as well.

Let's consider an operationalization which has two parts. Each part has several components. If the two parts share a common component, there may be a correlation which facilitates the overall operationalization.

To support security, account holders may be identified by fingerprinting and other biometrics. Biometrics is the identification of a human through the examination of physiological signs. These include eyeball-scanning, voice authentication, body temperature measurement, body contour measurement, etc.

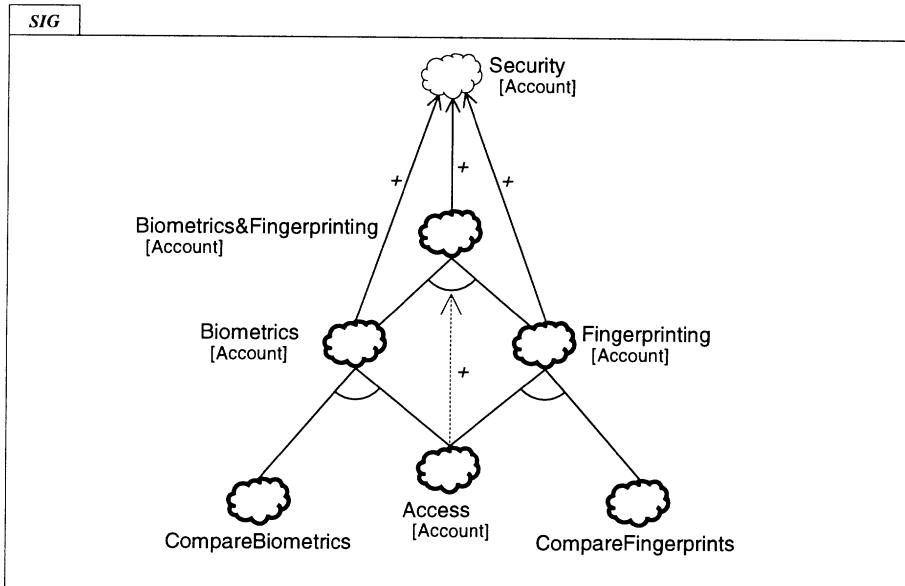


Figure 4.38. Benefits of using a common component in refinements of an operationalization.

Figure 4.38 shows the use of the operationalization **Biometrics&Fingerprinting[Account]**. It is refined into two parts, **Fingerprinting[Account]** and, to handle other biometrics, **Biometrics[Account]**.

Now **Fingerprinting[Account]** could be implemented by two components. The first, **Access[Account]**, accesses the account information and retrieves a stored fingerprint. The second, **CompareFingerprints[Account, currentClient]**, compares the retrieved fingerprint with the current client's fingerprint.

Likewise, **Biometrics[Account]** could be implemented by retrieving the account information and stored biometric information via **Access[Account]**, followed by comparing the retrieved biometrics with those of the current client.

Notice that both **Fingerprinting[Account]** and **Biometrics[Account]** require retrieval of stored information. By having **Access[Account]** retrieve the account information, stored fingerprints *and* stored biometrics, we retrieve more information at once, and only do one retrieval instead of two. This “family plan” (combined) retrieval [C. Smith90] facilitates the combined operation and offers efficiency. Thus there is a positive correlation from **Access[Account]** to **Biometrics&Fingerprinting[Account]**.

In passing we note that `Access[Account]` has two parents. This is an example of why the structure of softgoals and interdependencies is not a tree but a graph.

4.6 PUTTING THEM ALL TOGETHER: THE GOAL-DRIVEN PROCESS

In the previous and current chapters, the five components of the NFR Framework have been presented. For each component, some illustrations have been provided, with results shown in softgoal interdependency graphs.

We now elaborate upon the process of dealing with non-functional requirements, in the context of system development. The process was introduced in Chapter 2. We now present more detail on the management of SIGs.

The process starts with an initial set of high-level non-functional requirements as NFR softgoals, along with an initial set of functional requirements. It iteratively refines the NFR softgoals into more specific ones, while establishing interdependencies. These include relationships among softgoals, and relationships between softgoals and interdependencies. Priorities are identified, operationalizations are considered, tradeoffs are made, and design rationale is provided.

Figure 4.39 illustrates a softgoal interdependency graph that could be generated from our credit card example. It also shows a correlation, and the *selection* of operationalizations.

Figure 4.40 continues Figure 4.39 by showing the evaluation of the SIG. This shows the *impact of decisions* upon top NFRs.

Functional requirements serve as topics of softgoals, and can be used in claims. Figure 4.40 also shows the use of functional requirements in a SIG, and relates them to target decisions. We note that during development, functional requirements might need to be refined.

Throughout this process, softgoals and interdependencies are organized in a softgoal interdependency graph, and partitioned into two lists. *Open* is a list of all softgoals and interdependencies that are to be refined. *Closed* is a list of softgoals and interdependencies that have been completely refined.

In the process, the developer is in full control, selectively focussing attention and determining what softgoal or interdependency to refine next. The developer is also in full control, determining how to refine the chosen object and how much to refine.

After selecting a parent softgoal or interdependency from *Open* for refinement, developers can apply one of the pre-defined refinement methods; they can also propose and use refinements of their own making. Carrying out the chosen refinement on the selected parent results in the generation of its offspring, as well as a contribution of the offspring to the parent. The newly-created offspring and interdependency are then added to *Open*.

The use of pre-defined methods involves several steps. First is browsing a catalogue for methods which are applicable to the parent. Then is selecting a particular method. This is followed by ensuring that the type and topics

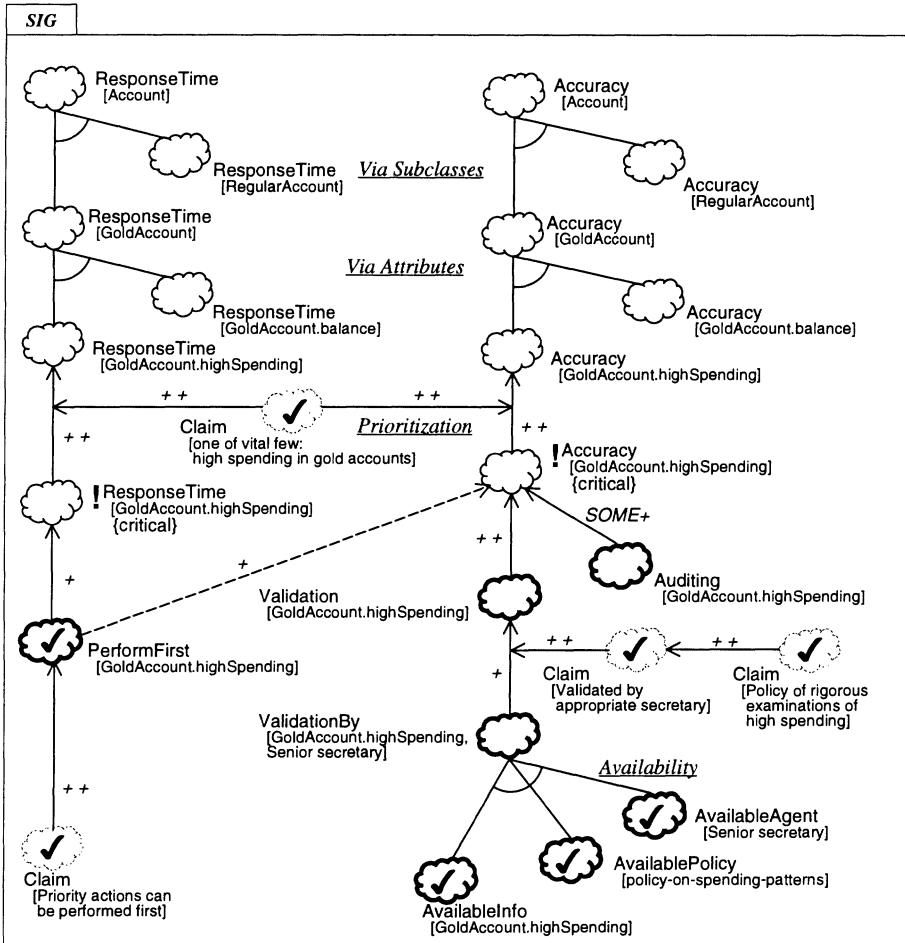


Figure 4.39. Selection of operationalizing softgoals and argumentation softgoals.

of the chosen softgoal or interdependency match (or are related to) those in the parent part of the definition of the selected method; in addition, one must ensure that the current situation meets the criteria given in the definition of the method. Finally is generating the offspring and contribution, according to the offspring and condition parts of the definition of the chosen method.

Due to design tradeoffs, correlations are introduced between softgoals. This involves using both the developer's judgement and correlation rules, and

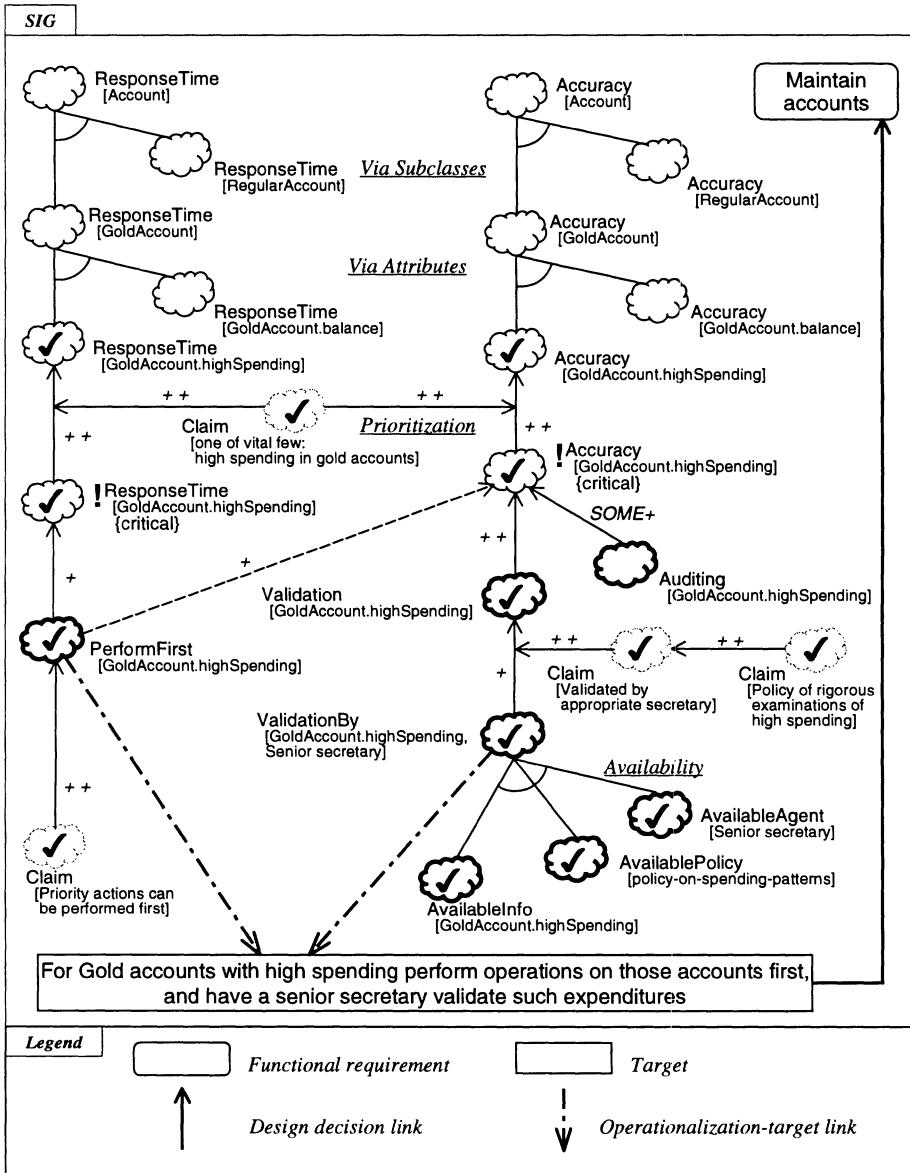


Figure 4.40. The impact of decisions on NFRs and Functional Requirements.

results in the creation of new interdependencies, and sometimes new softgoals, which are added to *Open*.

The use of pre-defined correlation rules involves several steps. First is matching parts of a SIG against patterns in the correlation catalogue. Then there is testing the conditions associated with the rules to see if any correlation rules are applicable to the newly-added softgoals or interdependencies of *Open*. Then there is inferring new contributions (and possibly new softgoals) to connect the existing SIG and the new softgoals or interdependencies. Finally the interdependencies and softgoals are proposed for addition to the SIG, subject to confirmation or rejection by the developer.

At any point during this process, the developer may prioritize the softgoals and interdependencies, in either *Open* or *Closed*, by assigning values to their criticality or priority attribute.

Prioritization may be needed due to resource limitation, bounded rationality [Simon81], domain constraints, etc. For example, due to limited development time available, the developer may need to spend more effort and time on priority softgoals than on less-important ones (e.g., more effort could be spent on an accuracy softgoal for large amounts than an accuracy softgoal on insignificant amounts).

Prioritization can also help resolve conflicts. In Figure 4.35, for example, **Validation** is good for **Confidentiality** but hurts **ResponseTime**. If **ResponseTime** were a priority (not shown in the figure), then **Validation** might be rejected, due to its negative contribution. Additionally, selection among alternative techniques may also be based on their relative importance, and analysis of arguments on their relative weights (e.g., distinguishing major policies from minor ones), etc.

This process of refinement, selection among alternatives, analysis of tradeoffs and prioritization is repeated for the chosen softgoal or interdependency until there are no more refinements the system or the developer can offer. At this time, the softgoal or interdependency is placed on the *Closed* list and another open softgoal or interdependency is selected.

A second alternative for softgoal or interdependency refinement is to simply label the softgoal or interdependency *satisficed* or *denied*. This may come from developer input or from the use of a method or correlation during refinement. Upon the change in any label, the interactive evaluation procedure determines, with the input from the developer, all those softgoals and interdependencies that are affected by such a change, and assigns new labels to them appropriately. A closed softgoal or interdependency should have as its label “ \checkmark ” or “ \times ”, while an open one can have any of the other label values.

The development process and SIGs can be further structured using a layering process. This is done for performance requirements in Chapters 8 and 9.

By using the components and development process, the NFR Framework takes a goal-oriented approach to dealing with non-functional requirements. This approach makes it possible to systematically treat non-functional require-

ments and ultimately use them as the criteria for making selections among competing target alternatives. This is complementary to approaches which focus on the evaluation of products.

4.7 DISCUSSION

Key Points

The NFR Framework takes a *knowledge-based* approach. First, the history of design decisions is recorded in graphs. Second, development knowledge is organized in catalogues. In both cases, there is a cost associated with collecting and recording the knowledge. However, there are also payoffs associated with this approach.

Softgoal Interdependency Graphs concisely record decisions. Our experience has been that a subsequent review of graphs brings key points to a developer's mind, even if the graph is reviewed a long time after it was prepared. In addition, graphs record patterns of commonly-occurring situations, and can be used and re-used in similar situations. Graphs have also been found useful for dealing with a variety of changes [Chung95a,96].

Of the five components which the NFR Framework offers, this chapter has presented *methods* and *correlations* which Chapter 3 did not cover.

These two components help construct a softgoal interdependency graph in which softgoals and interdependencies, together with their labels, are organized as a record of design intentions, alternatives, tradeoffs, decisions and rationale — the focus of Chapter 3.

Methods are a means to collect, encode and catalogue knowledge about clarifying the meaning of softgoals, satisficing them and providing arguments in support or denial of development decisions. Correspondingly, the NFR Framework offers three kinds of methods: NFR decomposition methods, operationalization methods and argumentation methods. Once catalogued, methods can be tailored, improved, extended and reused.

One group of methods provide facilities for *prioritizing* softgoals. This can be done on the basis of workload, other domain information, and the developer's expertise.

Similarly, correlation rules are a means to collect, encode and catalogue knowledge about interactions, synergistic or harmonious, between and among softgoals and interdependencies. Correlations are useful particularly during the analysis of tradeoffs among alternative techniques and discovery of omission of softgoals (NFR softgoals or operationalizing softgoals). Just like methods, correlations can be tailored, improved, extended and reused.

The Framework can be applied to a variety of NFRs (e.g., accuracy and performance) by providing *catalogues* of NFR-specific knowledge. When adapting the Framework to handle an additional NFR, the representation and reasoning facilities of Chapter 3 can be incorporated, essentially as-is. That is, the kinds of softgoals, interdependencies and refinements, as well as contribution types, label values and the evaluation procedure, can be viewed as being

“fixed.” Furthermore, the structure for defining methods and correlations is provided. In addition, the generic refinement methods can be viewed as given, subject to modification by the developer if desired. The main work, then, will be to add the counterpart of Chapter 4, by cataloguing concepts and methods applicable to the particular NFR. While there is a cost to preparing catalogues, our experience in several case studies (Part III of this book) is that there is a payoff when the catalogues are subsequently used.

Such catalogues are presented in Part II for a variety of NFRs. It’s important to observe that different NFRs have often been addressed with completely different models and means of analysis; for example, [ITSEC89] for security, and [C. Smith90] for performance). Yet the Framework can handle a variety of NFRs, both individually and in combination, in a goal-oriented manner.

Literature Notes

This chapter is adapted from [Mylopoulos92a] [Chung93a].

The representation of security requirements is adopted from the area of computer security (e.g., [Hartson76]).

The **VitalFewTrivialMany** prioritization method has its origin from Juran [Juran64] who coined the terms *vital few* and *trivial many* as applied to the Pareto principle, after the Italian economist, Vilfredo Pareto, who quantified the extent of inequality or nonuniformity of the distribution of wealth. Juran later noted in [Juran79] (the first edition came out in 1951) that it was a mistake to name it the Pareto principle, and that the principle should have been identified with M. O. Lorenz, who had used curves to depict concentration of wealth in graphic forms. McCabe et al. [McCabe87] applied this principle to software quality activities. This also reflects the 80–20 rule that 80% of development effort or software improvement results from 20% of the software under development or in the system [Boehm87]. Prioritization is also used in principles for building performance into software systems [C. Smith90].

The correlation catalogue is similar in spirit, to the “relationship matrix” in [Hauser88], which indicates, in terms of types and four kinds of values (strong positive, medium positive, medium negative, and strong negative), how much each engineering characteristic affects each customer quality requirement. Unlike the NFR Framework, the House of Quality by Hauser et al. deals with only actual design instances, rather than offering generic methods with topics or generic correlation rules with conditions which facilitate reuse of design knowledge. Another difference is that, via a softgoal interdependency graph, the NFR Framework allows for justifications of design decisions to be tightly associated with decision points.

An earlier, less formal description of some correlations among accuracy, security, cost, and user-friendliness can be found in [Chung91a].

4.8 RELATED LITERATURE FOR THE FRAMEWORK

The characteristics of the NFR Framework include:

- The focus of the work is *large systems* development rather than programming-in-the-small;
- Requirements specifications consist of *both functional and non-functional requirements*;
- The solution takes a process-oriented approach, where non-functional requirements are treated as a special kind of goal, “softgoals”;
- This work adopts the ideas in work on decision support systems and the DAIDA environment.

The NFR Framework aims to provide a systematic methodology for generating large systems from requirements that consist of both functional and non-functional requirements. There are several lines of work which address aspects of these aims. Related work is discussed in three parts.

- Since we treat non-functional requirements as softgoals, we discuss goal-oriented approaches in several different areas.
- Since we are concerned with developing systems, we discuss existing tools, methodologies, frameworks, and environments used for developing software systems.
- Since quality control issues are relevant to non-functional requirements and have been dealt with in those areas of management, manufacturing, software, and information system development, we discuss projects and approaches in such areas.

Goal-Oriented Methodologies

Systematic goal-oriented approaches have been used in a variety of areas. However, the concern of many of such approaches is not system development, in particular.

Of particular relevance to our work, is work in design rationale. In the context of qualitative decision support systems, [J. Lee91] presents a goal-oriented approach to facilitating decision-making process, which extends an earlier model for representing design rationale [Potts88] by making explicit the goals presupposed by arguments. More specifically, each design issue is treated as a goal to fulfill which is supported, denied or presupposed by various claims. Each claim or its relationship with the associated goal can again be treated as a goal. This leads to a recursive argumentation process, which is captured in term of a dependency representation graph (DRG). In order to support this approach, Lee builds a system, called SYBIL, which consists of a dependency representation language (DRL), four types of services, and a user interface. DRL is used to describe various relationships among claims, and the four types of services are intended for: (i) managing such relationships, (ii) making use of previous decisions, (iii) evaluating the degree of satisfying softgoals according to the scheme which each particular decision-maker provides, and (iv) allowing

alternative decisions with different priorities on the terminal goals. Our work specializes Lee's work to the context of system development with NFRs. When compared to Lee's work, the NFR Framework distinguishes goals that are associated with functional requirements from ones that are with NFRs. We further categorize the latter kinds of softgoals into three kinds of softgoals: NFR softgoals, operationalizing softgoals, and claim softgoals; all sharing a common structure of type and topic. By separating concerns, this categorization helps identify the right concerns at the right places during system development. The provision of operationalizing softgoals within this categorization makes it possible to use NFRs as criteria for selecting among design alternatives. Our focus on system development facilitates the capture and reuse of NFR-related knowledge, in terms of three kinds of methods and three kinds of softgoals. The NFR Framework also offers finer-grained contribution types with semi-formal semantics, and a built-in, evaluation procedure which is semi-automatic, allowing for developer input. Goal synergy and conflict is catalogued in terms of correlations, for later reuse. Another benefit of our approach is that our methods, correlations, and an evaluation procedure make it possible to automate parts of the system development process.

The TI (Transformational Implementation) project [Balzer85] [Fickas85] led by Balzer at USC ISI is one of the foremost automatic programming projects in the world, attempting to develop software through the interaction of system builders and automated assistants. A main theme of the project is the efficient implementation of conceptual system designs constructed in terms of a wide-spectrum executable language, called Gist. Gist is based on state-transitions coupled with the entity-relationship model. Specification refinements are achieved by constraining the allowable state transitions. While treating each design component as a goal to fulfill, the project has generated a set of design alternatives (or methods) and a set of criteria for design decisions that form the backbone for a software development methodology.

KATE is one of several projects which address programming-in-the-large. It is led by Fickas at the University of Oregon [Fickas87] [Fickas88] and its scope extends that of the TI project by considering requirements specifications. A fuller specification gradually evolves from an incomplete and inconsistent one, and that is later transformed into a specification that can serve as a starting point for the TI framework. TI and KATE use a wide-spectrum executable language, Gist, while the NFR Framework looks to the DAIDA project with declarative non-executable knowledge representation language features. Importantly, Fickas acknowledges the need to consider non-functional requirements in generating conceptual system designs.

Glitter [Fickas85] takes a goal-oriented, formal approach, using problem solving to automatically find and apply portions of transformations. Glitter takes an interactive approach: it tries to automate many tasks, leaving small portions of tasks to developers. In this regard, it is more automated than the NFR Framework. Both Glitter and the NFR Framework represent such con-

cepts as goals, methods, design rationale, evaluation and correlation (indexing to inter-relate goals).

Work on critiquing [Fickas88] looks for inconsistencies in specifications, such as policies (goals and associated priorities) which are not met positively by a target. Fickas points out the need to deal with conflicting policies via compromise and tradeoffs, which are important in the NFR Framework. A qualitative approach to critiquing is also available [Downing92]. It is also goal-oriented, relating decisions to goal achievement. Interestingly, decisions are partitioned into groups which have equivalent impact. Downing's exhaustive-generation approach differs from the NFR Framework's developer-directed approach. A goal-oriented approach has been used also for requirements acquisition [Dardenne91].

In the domain of VLSI high-level synthesis, the FAD (Feedback Aided Design) project [Liew90] takes a quantitative approach to improving resource usage by intelligently selecting optimizations and applying them to a previous solution similar to a currently desired solution. This project considers firstly multiple resource goals, such as (quantitative) limitations on time (e.g., time under 2000 units), area (e.g., area under 1000 units), and power (e.g., under 1500 units), and secondly an objective function whose parameters are the resources. In an attempt to improve resource usage of a previous solution, resource goals are, one at a time, decomposed in the manner of a tree; optimization strategies (which will modify a subset of resource components) applicable to the leaves are listed; and the value of the objective function with the optimization strategies is computed. Choosing the better one between the current and the previous value, the developer repeats the improvement process until satisfactory. Although specific contents would be different, the decomposition, optimization strategies, and objective function computation may respectively be comparable to our decomposition, satisficing, and evaluation methods (argumentation is not offered in FAD). Besides the quantitative approach taken here, however, the project does not provide the guidance for detecting possible conflicts among the goals and optimizations or the scheme to capture design rationale in the sense of argumentation.

In the area of machine translation, [DiMarco93] and [DiMarco90] present a scheme for automatically evaluating the translation of one natural language (e.g., English) into another (e.g., French). The evaluation criteria are stylistics such as clarity and dynamicity, which are treated as softgoals. According to the (stylistic) grammar associated with each goal, each target sentence is categorized into one of three types, each reflecting the bias (or orientation) of the sentence (e.g., a sentence is dynamic, neutral, or static). The work is interesting to us in that it recognizes three types of categories into which the evaluation scheme differentiates each target sentence. As the work's primary concern lies in evaluation rather than translation, it does not address those generation phase issues such as conflict detection and resolution, design rationale maintenance or selection among alternative target constructs.

In the area of computer architecture design, [Hooton88] provides a theory of plausible design (TPD). In this theory, each requirement consists of a combination of functional and non-functional requirements, such as “maximal concurrency,” and is treated as a goal to satisfy. The plausibility or satisfiability of the decomposed subgoals (or goal if a terminal node) categorizes each goal into one of four types: validated, assumed, unknown, and refuted. A goal is *validated* if there is a significant evidence for and no evidence against its satisfiability, *assumed* if there is no evidence against its satisfiability, *refuted* if there is some evidence against its satisfiability, and *unknown* when initially introduced. Then, the decomposition of a softgoal into subgoals, devised by human developers when needed, is guided since satisficing a softgoal of one type can be achieved in terms of its subgoals of the same type. The presence or absence of evidence or counter-evidence is determined by human developers through simulation, experimentation, literature search, etc. An arbitrary decomposition is also allowed when certain justification is present. The work is of interest as it recognizes the need for distinguishing different types of relationships that can hold between a goal and its subgoals, and subsequently offers an evaluation scheme which is a modification of an ATMS [de Kleer86]. The NFR Framework’s evaluation procedure adapts some concepts from ATMSs.

Software Development

In the field of software development, much attention has been given to the generation of efficient implementations from formal specifications, either manually or automatically. For example, [Spivey87] and [Wordsworth87] describe a software development methodology based on a set-theoretic formalism named Z, while [Abrial88] describes a tool (named the B tool) that can serve as proof assistant in establishing that an implementation generated through a sequence of transformations is indeed consistent with a given Z specification.¹ Another project in the same league is the Cornell Program Synthesizer [Teitelman81] which inspired the Nuprl proof development system [Constable86]. Generally, such work focuses on programming-in-the-small applications and is most relevant to the mapping problem from designs to implementations. A fine survey of AI-related work on this problem, which has traditionally been called “automatic programming” can be found in [Barstow87], while [Hayes87] gives an excellent comparative account of several formal specification methodologies.

The Gandalf project at CMU [Haberman86] includes a component called SMILE, which consists of a multi-user software engineering environment. SMILE is intended to offer intelligent assistance to the software developer. However, unlike DAIDA [Jarke92a, 93b] whose user support comes from a knowledge base (managed by the GKBMS), the knowledge used by SMILE is hard-coded and cannot be changed or extended easily. [Kaiser87] describes

¹Both Z and the B tool play an important role for the design-to-implementation mapping assistant of DAIDA [Borgida89].

an effort to generalize SMILE by including an *objectbase* and a *model of the software development activity* which represent explicitly some of the hard-coded knowledge of SMILE. SMILE is similar in spirit to DAIDA. However, DAIDA differs by focussing on information systems and taking full advantage of its narrow focus in selecting languages, developing tools, and adopting methodologies.

The CHI project [D. Smith85] [Green86] at Kestrel Institute, a descendant of PSI project carried out earlier at Stanford [Green76], is based on a *wide-spectrum language*², called V, that is executable. To its credit, the project has a commercial product called REFINETM. The starting point here for the software development process is a formal specification consisting of logical assertions which is mapped, via several transformations, to an efficient LISP implementation [Westfold84]. However, the logical assertions that constitute the initial formal specification tend to focus on the behaviour of the software-to-be-built rather than its intended environment and information content and CHI relies on computational notions such as sequencing, compute-versus-store, etc., to make them executable. Thus, CHI, like other projects focussing on programming-in-the-small applications, bypasses the requirements modelling and analysis phase and only addresses the problem of going from a less deterministic description of the software-to-be-built to a completely deterministic (and efficient) one. Also, CHI does not offer much guidance on how conceptual system designs are produced.

As one of the foremost advocates for the need of world modelling in software development, Jackson [Jackson83] presents a system development method (known as JSD) which is in commercial use. JSD starts with a world model and, through a functional system design, produces an implementation. However, JSD, as other diagrammatic approaches, adopts an *ad hoc* set of modelling features and is often quite informal in the specifications it starts with and produces. The NFR Framework uses concepts from knowledge representation and artificial intelligence.

The work of Bubenko [Bubenko81] is close in spirit to the NFR Framework, using layers of formal software specifications as a basis for the definition of a software development process. The first layer of Bubenko's work comprises an abstract world model, called "understanding level," and the second layer comprises a conceptual system design. In order to define model behaviour in non-procedural terms, time is considered as the most essential concept and the world model is described in an extended time perspective rather than in points of procedure invocation.

Recently, there have been many proposals for goal-oriented approaches to requirements engineering, including [Kaindl97] which is semi-formal and pragmatic, and [Dardenne91] which is more long-term and "heavy-weight." In KAOS, functional requirements are successively decomposed into more concrete ones and *operationalized* in terms of agents or constraints. Thanks to

²A language is wide-spectrum if it offers constructs for specifications ranging from highly abstract and non-procedural to ones that are implementation-oriented [Bauer76].

its solid formal semantics, KAOS is amenable to the construction of sophisticated analysis tools. However, these goal-oriented approaches mainly deal with functional requirements.

The notion of goals has also been used extensively to deal with various kinds of key issues in requirements engineering. For example, goals are treated as the central concept in developing requirements specification [Anton96], an important ingredient during scenario analysis [Potts94], and verification of requirements [Dubois89], and support for reuse [vanLamsweerde97]. Goals are also used to deal with conflicting requirements, with multiple views [Nuseibeh93] and multiple stakeholders [Boehm96], which may require negotiations to resolve [Robinson90].

Some software development methodologies have a strong emphasis on functional aspects of the system, hence they focus on functional requirements, while offering product- and process-oriented approaches. The NFR Framework provides a basis for dealing with both non-functional and functional requirements (Section 3.4) in a process-oriented way. In comparison, work on software metrics addresses NFRs in a product-oriented way.

In the area of information system development, the NATURE project recognizes the crucial role that non-functional requirements play in requirements engineering, and has in its scope the representation of NFRs [Jarke93a]. This project also attempts to address teamwork support and the use of repositories in dealing with NFRs.

A research group at the University of Trondheim, Norway, has contributed to the area of information system engineering. For example, a framework for performance engineering for information systems is presented by Opdahl [Opdahl92]. Focussing on the prediction and improvement of performance of information systems during development, it incorporates models of software, hardware, the organization and the applications. It offers sensitivity analysis and a number of tools, and is integrated into an environment for a larger performance evaluation project [Brataas92]. Opdahl [Opdahl94] argues for quantifying performance demands during requirements specification.

Quality Assurance and Control

The field of quality control is rich in general guidelines and statistical measurement techniques that are used on a daily basis. Although related to each other and sometimes used interchangeably, quality assurance is regarded as being broader in scope than quality control. According to the ISO 8402 standard, quality assurance refers to the overall management function that determines and implements the quality policy, which is established by senior management as the overall quality intentions and objectives of an organization. According to ISO 9001, quality control is concerned mostly with the latter part of quality assurance, ensuring firstly the use of appropriate tools in producing the final product and secondly that the final product meets pre-determined standards.

One of the most advanced methodologies for quality control, described by Hauser et al. [Hauser88], aims to reflect customer attributes in different

phases of automobile design, by helping establish and visualize relations between manufacturing functions and customer satisfaction. In this kind of so-called Quality Function Deployment methodology, (QFD) customer attributes are (recursively) treated as bundles of more finer-grained attributes and customers' evaluations of their relative importance are registered. The relationship between these customer attributes and engineering characteristics is then expressed in terms of different degrees of scale, along with the relationship between different engineering characteristics. Finally information associated with engineering characteristics, such as technical difficulties, relative weights, and cost, are recorded and used in making design decisions.

For strategic quality analysis and management, Garvin [Garvin87] presents eight critical dimensions or categories of quality: performance, features, reliability, conformance, durability, serviceability, aesthetics, and perceived quality. He states that some dimensions have more objective standards than some other ones, stating that performance has more objective standard than perceived quality. He also states that some dimensions are always mutually reinforcing, while some other ones are not. As an improvement in one may be achieved only at the expense of another, a product or service can rank high on one dimension of quality and low on another. This interplay is noted as the key that makes strategic quality management possible and suggests competing on selected dimensions.

Among the eight dimensions, robustness, which Garvin treats as a component of conformity, has drawn a significant interest in the area of manufacturing design. Taguchi et al. [Taguchi90] presents a statistical approach to dealing with robustness, where one major emphasis lies in taking into account of critical interactions among design components. This approach emphasizes that quality loss starts from the time a product is shipped and should include warranty costs and nonrepeating customers. Garvin considers that quality loss increases by the square of deviation of design components from the target value.

A more focussed area, of interest to the software engineering community is *software quality assurance (SQA)*, which is defined in ANSI/IEEE Standard 729-1983 [ANSI] [IEEE]:

"In software system engineering, a planned and systematic pattern of all actions necessary to provide adequate confidence that the software and the delivered documentation conforms to established technical requirements."

Previous work in this area has focused primarily on managerial issues, quantitative measurements or product testing and inspection (See, for instance, [IEEESoftware87]).

The complementary nature of process-oriented and product-oriented approaches to dealing with non-functional requirements has been noted. Hailstone [Hailstone91], for instance, warns against the danger in taking a unilateral approach. Taking a solely product-oriented view, on the one hand, may ignore or poorly implement any software attribute that is difficult to quantify, such as usability, flexibility, and maintainability. On the other hand, taking a solely process-oriented view would, in the extreme, lead to abandoning any form of software testing.

Taking a quantitative approach, earlier work by Boehm et al. [Boehm78] considered quality characteristics of software, noting that merely increasing developers' awareness would improve the quality of the final product. Also supporting a quantitative approach to software quality, Basili and Musa [Basili91] advocate models and metrics of the software engineering process from a management perspective.

Also from a management perspective, Zultner [Zultner91] and Oakland [Oakland89] apply the *Total Quality Management* approach (TQM) of Deming [Deming86] to software development and maintenance. The application results in guidelines for software managers to know and do (called *fourteen points*), for management to avoid (*seven deadly diseases*), and poor practices that hinder quality and productivity (*fifteen obstacles*). Similarly, albeit less specific to software development, a general management approach to improving productivity has been widely publicized, in the name of *quality (control) circle (QC)* which aims to increase efficiency through worker motivation, increased communication among and involvement of employees at all levels within the organization in the decision-making process [Crocker84]. In relation to software quality assurance, some of these and other related work, such as work by Ishikawa, Juran [Juran64] [Juran79], Sandholm, and Crosby, are surveyed and assessed in [Schulmeyer87].

III Types of Non-Functional Requirements

5 TYPES OF NFRs

Part II shows how the NFR Framework can accommodate different types of non-functional requirements. Part II focusses on three: accuracy, security and performance requirements.

Other types of non-functional requirements (e.g., usability, portability, etc.) could be treated in essentially the same manner; however, they are not specifically addressed in this book.

In each case, non-functional requirements are represented as a set of softgoals. Concepts for each type of NFR are collected and organized, along with associated development techniques. Methods for decomposing and analyzing softgoals are presented and organized in catalogues, along with rules for correlating softgoals. These methods and rules are used to move towards a target system. The selected system is evaluated to determine how well the system meets the non-functional requirements.

Chapter 5 provides a “roadmap” of different types of NFRs. This draws on definitions and classifications of NFRs from various categorizations and standards.

Chapters 6 through 9 apply the NFR Framework to deal with particular NFRs: accuracy, security and performance requirements. NFRs are represented as softgoals, which can be conflicting or synergistic. Each chapter presents refinement methods for the particular type of NFR, and organizes methods into catalogues. Tradeoffs among methods are organized in correlation catalogues. These catalogues are available for reuse in refining NFR softgoals by generating

new softgoals and interdependencies from existing ones. To move towards a target system, softgoals are “operationalized” while taking a “satisficing” approach (finding “good enough” solutions). Justifications are stated for development decisions, which are made in the context of the particular domain, and in the presence of resource limitations and softgoal conflicts. The evaluation procedure is used to show developers the impact of development decisions upon NFRs.

In Chapter 6, the NFR Framework is applied to accuracy requirements. Like other types of NFRs, accuracy requirements are treated as softgoals. The chapter shows how accuracy-related requirements can be treated. This includes a treatment of information flow and its impact on accuracy. An illustration details the use of the components of the Framework to deal with accuracy requirements. It shows how the Framework helps a developer detect softgoal conflicts, consider development alternatives, and address the needs of the domain (e.g., priorities) in resolving the conflict.

In Chapter 7, the NFR Framework is applied to security requirements. This chapter shows how different notions of security can be arranged in the security type. Security-related development techniques are captured as refinement methods. Tradeoffs among methods are represented as correlation rules. Security requirements are then treated as softgoals, which are repeatedly disambiguated, refined and co-related to one another, using catalogues of methods and correlation rules. The impact of decisions is determined using the evaluation procedure. The process of dealing with security requirements is illustrated. It shows how to take general security concerns and policies into consideration, and how to focus on a particular aspect of security in addressing the needs of a particular application domain.

In Chapter 8, the Framework is applied to performance requirements. This chapter introduce performance concepts and factors that should be considered, along with principles for building performance into systems. It illustrates how the Framework and its catalogue of methods can be used to deal with trade-offs, priority softgoals, and how domain information can be reflected in meeting performance requirements. Catalogues of methods are further organized by a language-based layering, to make the development process more manageable.

To be more specific about performance, Chapter 9 focusses on performance requirements for a particular kind of system, namely information systems. Performance requirements are addressed while considering priorities, workload and other characteristics of the organization for which a system is being developed. The method catalogues draw on results from databases, performance engineering, and implementation of object-based specification languages for information systems. There are many implementation alternatives with varying performance characteristics, and these are reflected in the method and correlation catalogues.

There is a large variety of non-functional requirements, which are important to address. In essence, the objective of Part II of this book is to show that

such types of NFRs can be suitably addressed by using the NFR Framework of Part I. We will demonstrate this for the cases of accuracy, security and performance requirements (Chapters 6 to 9). For each type of NFR addressed in Part II, our approach is to collect and represent concepts and terminology relevant for the particular NFR (e.g., performance, or accuracy). Then we gather and catalogue refinements and operationalizations which are appropriate for the particular NFR.

The problem of addressing NFRs has existed for a long time. This chapter starts Part II with an overview of the types of NFRs. It outlines some categorizations of NFRs, and some standards relating to NFRs, which are taken from the literature. Then, from our perspective, we present a list of a fairly large number of NFRs.

We feel that such types of NFRs can be accommodated using the NFR Framework; however most of them have not yet be addressed in detail using the Framework.

It's important to note that particular requirements have been addressed with completely different classifications, models and means of analysis. For performance requirements, approaches include [C. Smith90] and [Jain91]. For integrity we can turn to [NIST90], for example.

5.1 CATEGORIZATIONS OF NFRs

We now consider some categorizations of NFRs. First, however, it should be noted that there is not a formal definition or a complete list of non-functional requirements. Neither is there a single universal classification scheme, accommodating all the needs of different application domains under different situations. Furthermore, different people use different terminologies; this can make it harder to use given categorizations without customizing them.

Early work on software quality [Boehm76] presents a tree of software quality characteristics (See Figure 5.1). Meeting a parent quality in the tree implies meeting the offspring qualities. This includes a number of “-ilities” for a variety of software quality attributes.

A great diversity of types of non-functional requirements is also discussed in [Roman85]. Roman's presentation includes several classes of NFRs:

- *interface constraints*,
- *performance constraints*, including response time, throughput, storage space, reliability, security, survivability, productivity, etc.,
- *operating constraints*, including physical constraints, personnel availability, skill-level considerations, etc.,
- *life-cycle constraints*, including maintainability, enhanceability, portability, flexibility, reusability, compatibility, resource availability, time limitations, methodological standards, etc.,
- *economic constraints*, including development cost,

Factor/Acronym	Performance					Design			Adaptation				
	EF	IG	RL	SV	US	CR	MA	VE	EX	FX	IP	PO	RU
ACQUISITION CONCERN													
Criterion													
P	Accuracy			X									
A	Anomaly management		X	X									
E	Autonomy			X									
R	Distributedness			X									
F	Effectiveness	X											
O	communication												
R	Effectiveness	X											
M	processing	X											
A	Effectiveness storage	X											
N	Operability			X									
C	Inconfigurability				X								
E	System accessibility		X		X								
	Training				X								
D	Completeness					X							
E	Consistency					X	X						
S	Traceability					X							
I	Visibility						X	X					
N													
A	Application independence								X				X
D	Augmentability									X			
D	Commonality										X		
A	Document accessibility											X	
T	Functional overlap									X			
A	Functional scope										X		
T	Generality								X	X			X
I	Independence									X	X		X
O	System clarity										X		X
N	System compatibility									X			
	Virtuality								X				
G	Modularity				X				X	X	X	X	X
E	Self-descriptiveness			X				X	X	X	X	X	X
N								X	X	X	X	X	X
R	Simplicity					X	X	X	X	X	X	X	X
A													
L													

Legend

EF	Efficiency	CR	Correctness	EX	Expandability
IG	Integrity	MA	Maintainability	FX	Flexibility
RL	Reliability	VE	Verifiability	IP	Interoperability
SV	Survivability			PO	Portability
US	Usability			RU	Reusability

Table 5.1. Software quality factors and criteria (From [Keller90]).

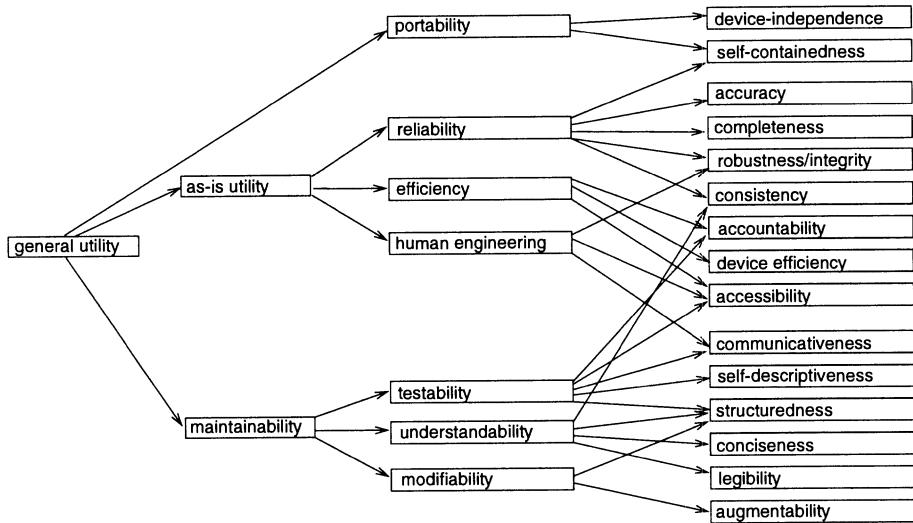


Figure 5.1. Software Quality Characteristics Tree (From [Boehm76]).

- *political constraints*, including policy and legal issues.

In Chapter 1, we referred to classifications of NFRs done at the Rome Air Development Center (RADC). *Consumer-oriented attributes (software quality factors, Table 1.1)* are NFRs (such as efficiency, correctness, and interoperability) which are of interest to software consumers. *Technically-oriented attributes (software quality criteria),* on the other hand, are NFRs (such as anomaly management, completeness, and functional scope) which are more meaningful to software producers. Table 5.1 [Keller90] relates the consumer-oriented attributes to the technically-oriented ones. This table was developed as part of a large software quality assurance project, which incorporated a metrics-oriented approach.

In the area of usability inspection, a number of heuristics are offered for reviewing a design for usability [Nielsen93]:

- Use simple and natural language,
- Speak the users' language,
- Minimize the users' memory load,
- Have consistency,
- Have feedback,

- Have clearly marked exits,
- Have shortcuts,
- Have precise and constructive error messages,
- Prevent errors.

Another classification scheme [Sommerville92] categorizes a number of non-functional requirements into three general groups: process considerations, product considerations, and external considerations (See Figure 5.2).

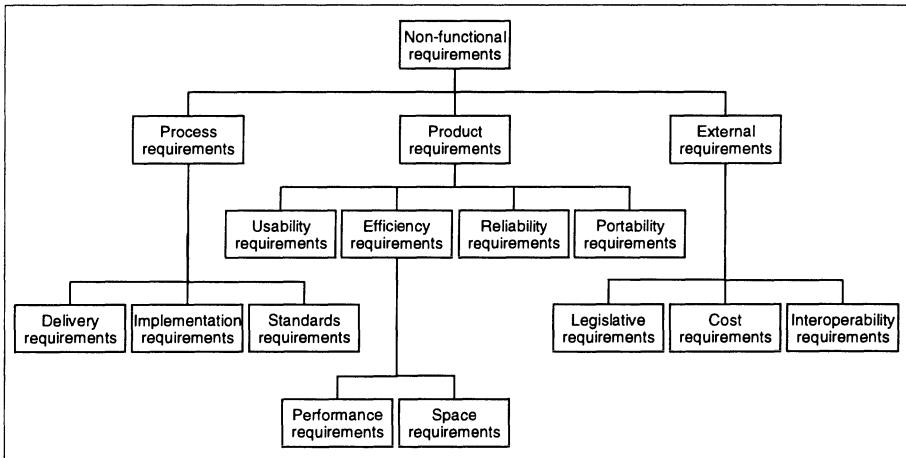


Figure 5.2. Types of non-functional requirements (From [Sommerville92]).

We have outlined some of the many categorizations of NFRs. Other classifications of NFRs are available. There is a chapter on specifying “nonbehavioral requirements” in [Davis93]. A variety of classifications of non-functional requirements is presented in [Loucopoulos95].

5.2 STANDARDS

As the problem of NFRs has existed for a long time, we can draw upon a body of existing work. To understand the breadth of possible NFRs, we can look to standards.

Standards are developed and published by a number of agencies, including [ISO] [IEEE] [ANSI]. Standards for NFRs are described in [Thayer90] [Pohl96]. In looking at [Thayer90], for example, we see many standards, including Canadian standards, United States military standards, and IEEE no. 830 (1984), among others; further definitions are given in that book’s glossary.

Standards provide high-level guidelines. For example, they can provide a structure for documents and processes. The IEEE standard has English definitions of several NFRs. This tells us, for example, how reliability differs from security. Standards do not always provide detailed guidelines for methodology. For example, the Trusted Computer Systems Evaluation Criteria [TCSEC85] imposes no constraint on the methodologies used.

It should be noted that there can be many standards for a particular NFR. For security, for example, standards include [ITSEC91] [CTCPEC92] [TCSEC85].

5.3 A LIST OF NFRs

NFRs cover a wide variety of issues for software quality. They are sometimes referred to as “-ilities and -ties.” To give the reader a feel for the breadth of possible areas which can be addressed by NFRs, we present a list of NFRs in Table 5.2.

Note that we have not made any effort to organize the list by categories. This is because our purpose in presenting this list is to illustrate the large possible scope of NFRs, but not to present a particular arrangement of the NFRs. As we have seen earlier in this chapter, there are a variety of ways to categorize NFRs.

Subsequent chapters will present catalogues of NFRs, for accuracy, security and performance, using the NFR Framework. These three NFRs are addressed in detail. However, we have not addressed in detail most of the other NFRs in Table 5.2.

5.4 OUR APPROACH: THE NFR FRAMEWORK

The NFR Framework aims to provide a more systematic and global view of NFRs. It offers a rational and systematic approach to dealing with NFRs. It provides a way to categorize NFRs. It provides a qualitative approach to dealing with NFRs. Using the Framework, analysis of NFRs involves looking at NFRs, refining them (using methods), correlating and operationalizing them. This helps to decompose NFRs, and deal with ambiguities and other defects.

The NFR Framework does not require any assumptions about the completeness of non-functional requirements, the use of any single classification scheme, unanimous agreement on terminology, or homogeneity of different application domains. Quite the contrary, the NFR Framework helps the developer accommodate different needs of different applications while considering their particular characteristics.

Next we use the NFR Framework to deal with some specific NFRs. We show how the framework is specialized to deal with accuracy, security and performance requirements. Then we further specialize one framework, for performance, to deal with a particular area, information systems. Other NFRs mentioned in this chapter are not treated in detail in this book.

accessibility,	accountability,	accuracy,
adaptability,	additivity,	adjustability,
affordability,	agility,	auditability,
availability,	buffer space performance,	capability,
capacity,	clarity,	code-space performance,
cohesiveness,	commonality,	communication cost,
communication time,	compatibility,	completeness,
component integration cost,	component integration time,	composability,
comprehensibility,	conceptuality,	conciseness,
confidentiality,	configurability,	consistency,
controllability,	coordination cost,	coordination time,
correctness,	cost,	coupling,
customer evaluation time,	customer loyalty,	customizability,
data-space performance,	decomposability,	degradation of service,
dependability,	development cost,	development time,
distributivity,	diversity,	domain analysis cost,
domain analysis time,	efficiency,	elasticity,
enhanceability,	evolvability,	execution cost,
extensibility,	external consistency,	fault-tolerance,
feasibility,	flexibility,	formality,
generality,	guidance,	hardware cost,
impact analyzability,	independence,	informativeness,
inspection cost,	inspection time,	integrity,
inter-operability,	internal consistency,	intuitiveness,
learnability,	main-memory performance,	maintainability,
maintenance cost,	maintenance time,	maturity,
mean performance,	measurability,	mobility,
modifiability,	modularity,	naturalness,
nomadicity,	observability,	off-peak-period performance,
operability,	operating cost,	peak-period performance,
performability,	performance,	planning cost,
planning time,	plasticity,	portability,
precision,	predictability,	process management time,
productivity,	project stability,	project tracking cost,
promptness,	prototyping cost,	prototyping time,
reconfigurability,	recoverability,	recovery,
reengineering cost,	reliability,	repeatability,
replaceability,	replicability,	response time,
responsiveness,	retirement cost,	reusability,
risk analysis cost,	risk analysis time,	robustness,
safety,	scalability,	secondary-storage performance,
security,	sensitivity,	similarity,
simplicity,	software cost,	software production time,
space boundedness,	space performance,	specificity,
stability,	standardizability,	subjectivity,
supportability,	surety,	survivability,
susceptibility,	sustainability,	testability,
testing time,	throughput,	time performance,
timeliness,	tolerance,	traceability,
trainability,	transferability,	transparency,
understandability,	uniform performance,	uniformity,
usability,	user-friendliness,	validity,
variability,	verifiability,	versatility,
visibility,	wrappability.	

Table 5.2. A list of non-functional requirements.

5.5 LITERATURE NOTES

This chapter is based on [Mylopoulos92a] and [Chung93a].

6 ACCURACY REQUIREMENTS

The accuracy of information is often regarded as an inherent property of any automated information system. As a familiar example, some people inquire about a payment request, such as monthly credit card or telephone bill, and get a reply from a staff member saying, “As the transactions were handled by the computer, there *can’t* be any error!”

Accuracy is a necessary virtue of almost all information that an information system maintains. The Usenet newsgroup **comp.risks** shows various cases of inaccurate data which is useless and often invites user complaints, confusion, and distrust. The newsgroup also shows how inaccurate financial information may lead to monetary loss and damage to reputation of a financial institution, and how inaccurate medical information may lead to wrong diagnosis, medication or surgery. Laudon [Laudon86] described how, in a criminal-record system, inaccurate information may lead to wrongly issued warrants, false arrest or rejection of employment.

However, accuracy does not come automatically. Rather, it needs to be “designed into” the system. Consider a travel expense management system. Employees of the particular organization travel to various cities in different countries and participate in various meetings. The system should generate monthly expense reports for each employee, meeting, and project, as well as cheques to employees with the correct amount. The reports are used to monitor the adequacy of spending, as well as to estimate the budget of all the projects in the company for the following fiscal year.

Now, how do we design accuracy into the system so that the information in monthly expense reports and cheques is accurate?

For one thing, employees' expense vouchers can be *validated* — perhaps by some experts — before the system accepts any expense information. For another thing, *auditors* can periodically examine the vouchers and the information in the system. The more operationalizations for accuracy that are adopted, the more accurate such information should become.

However, such an operationalization may have negative side-effects. For example, validation may *HURT* operating cost as it involves the time of some experts. It may also *HURT* user-friendliness, as it introduces a time delay for employees to enter information into the system. What if the system allows users to directly update information in their own files? By not using the time of experts, this would *HELP* operating cost, and by not delaying input, this would *HELP* user-friendliness. But it may *HURT* accuracy, as we will not have a high level of confidence in the information the system maintains.

Thus, to design accuracy into the system, we need to be aware of various decomposition and operationalization methods and use them appropriately. They should not excessively hurt other NFRs such as user-friendliness, performance, cost and production time. Better yet, we need a systematic way of using the methods, considering their tradeoffs, recording design rationale, and evaluating design decisions.

In this chapter, we show how the NFR Framework helps to systematically “design accuracy into” a system. In effect, we produce an *Accuracy Requirements Framework* which customizes the NFR Framework to deal with accuracy requirements.

This first involves clarifying the notion of accuracy, which is used to treat accuracy requirements as softgoals. For this, Section 6.1 identifies different types of accuracy requirements, which are used as types of accuracy softgoals; the things that should be accurate are used as topics of softgoals.

In addition, this involves cataloguing the knowledge of methods for refining accuracy softgoals. Sections 6.2 through 6.4 show the three types of refinement methods described in Chapter 4, for decomposing and clarifying accuracy softgoals, operationalizing accuracy softgoals, and providing rationale for design decisions.

Finally, this involves cataloguing knowledge of implicit contributions between accuracy softgoals and other NFRs. Section 6.5 presents an accuracy correlation catalogue.

Section 6.6 illustrates the use of the Accuracy Requirements Framework to develop a sample system. Accuracy softgoals guide the overall process. The evaluation procedure is applied to a softgoal interdependency graph to determine the impact of design decisions. Evaluation draws on contributions, which help describing methods and correlations, and on decisions recorded as softgoal labels.

6.1 ACCURACY CONCEPTS

What does it mean for information to be accurate? Firstly, we assume that accuracy is a fundamental semantic attribute of any *information item*. By comparison, weight and volume are fundamental physical attributes of a material item. By *information item*, we mean a piece of information, such as “the colour of the White House,” “Sue’s address,” “Peri is a requirement engineer,” or “Matthias is a software architect,” that we ordinarily use in our communication.

If *Info* is an information item, then Accuracy of *Info*, written “Accuracy[*Info*]” is a requirement that *Info* accurately describe the corresponding information in the application domain.

How can we have a high degree of confidence in the accuracy of an information item? In order to answer this question, we need to better understand how information is created, manipulated and managed within and outside the system. For this, we introduce the notion of *information flow*. This notion will help clarify various types of accuracy, or lack thereof, and provide intuition for a variety of methods and correlations.

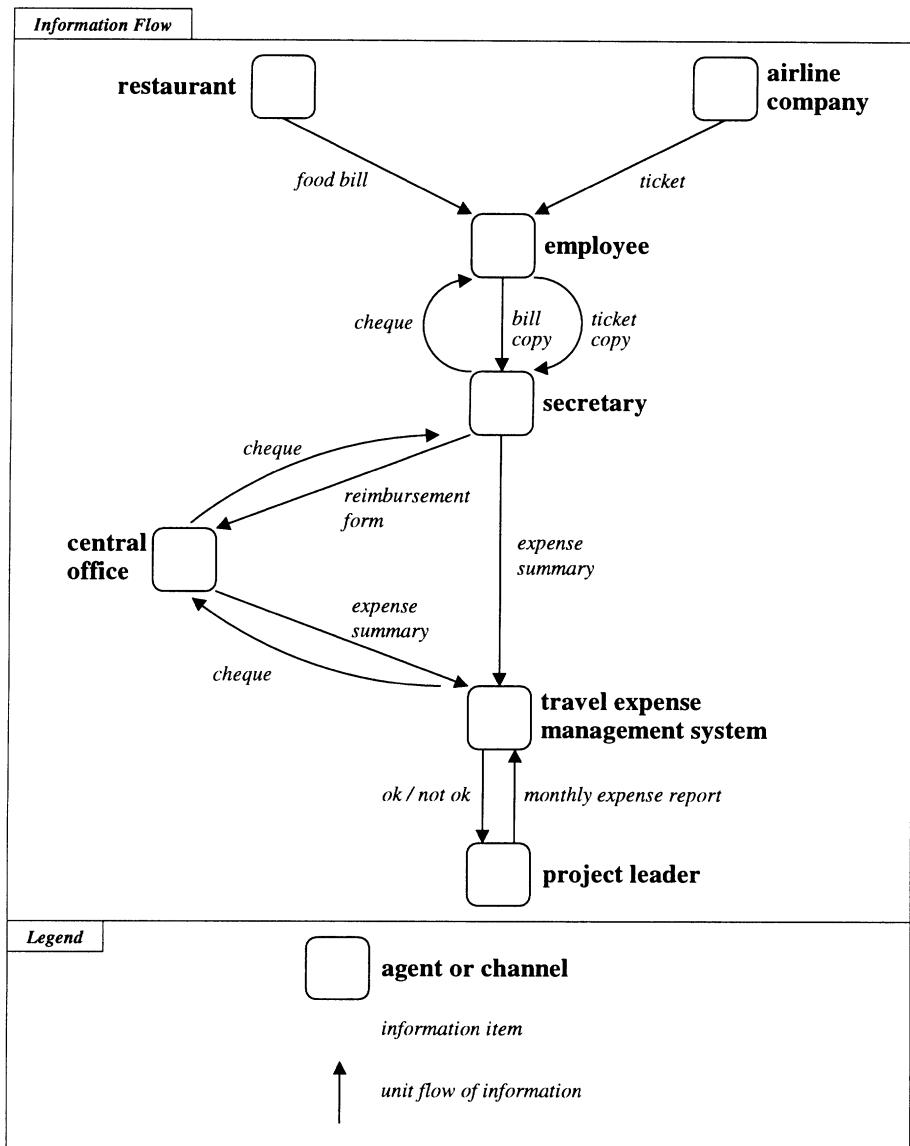
A Model of Information Flow

Accuracy requirements for information systems require the information maintained by the system to faithfully describe the corresponding aspects of the application domain. A fundamental premise in designing accuracy into such a system is that the accuracy (or inaccuracy) of a piece of information heavily depends on the way that information is *manipulated within the system as well as in its environment*.

For example, information may flow along sequential or parallel paths. Intuitively, we expect that long sequential paths will allow intermediaries to introduce inaccuracy, while parallel paths enhance accuracy since the same message is sent to its destination through more than one route, and the received messages can be compared with each other.

Figure 6.1 shows a sample information flow network for the travel expense management system. It models the ways that employees and other agents pass around information items.

In this information flow network, an airline sends the ticket to a project employee (project member) who gives a copy of the ticket to the secretary. Similarly, a restaurant gives a receipt to a project employee, who gives it to the secretary. The secretary determines the total expense from the individual expense items and sends a reimbursement request form with the vouchers as supporting documents to the central accounting office. The secretary may also insert an expense summary into the information system via electronic mail. In turn, the central accounting office enters the reimbursement request form into the information system and a reimbursement is issued after a suitably irritating delay. At the end of each month, the information system produces a monthly expense report and sends it to the project leader.

**Figure 6.1.** An example of information flow.

This simple, but common, scenario shows that an information item may be created, derived from other information items, maintained, and transmitted by a number of *agents*, which may be persons or systems. In addition to agents, the scenario includes *channels* used to transmit information. Individual links in the information flow network shown in Figure 6.1 represent *unit flows*.

We will present a qualitative model with decomposition rules for accuracy softgoals as well as operationalization methods that *HELP* or *MAKE* accuracy softgoals. Other approaches may treat accuracy as a probabilistic, fuzzy, or quantitative measure, in particular domains.

We treat accuracy as a non-functional requirement, to be satisfied. Our confidence in the accuracy of information items can range from total confidence to a complete lack of confidence.

Accuracy Types

Figure 6.2 shows a type catalogue which organizes the various specializations of accuracy.

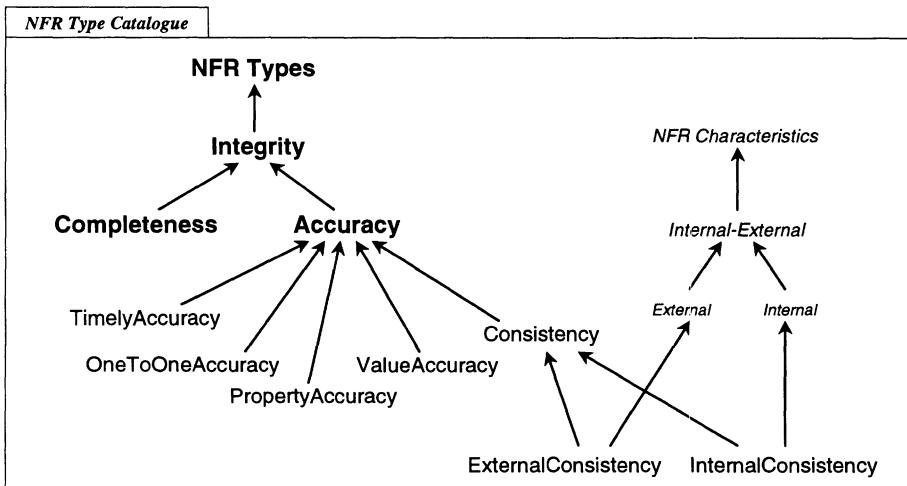


Figure 6.2. An accuracy type catalogue.

Why do we have an accuracy type catalogue? A type catalogue helps the developer accommodate different needs of different application domains. Here, for example, accuracy is presented as a specialization of integrity. Instead of forcing the developer to adopt any particular definition of accuracy, the type catalogue helps the developer pick out the concepts of accuracy that are most appropriate to the particular application domain. As a consequence, accuracy

requirements can be refined into more specialized requirements involving subtypes of accuracy.

Characteristics of NFRs are shown in small italics at the right of Figure 6.2. They are not NFR types in their own right. However, they modify and specialize the meaning of types. Specialized NFR types can be formed by combining NFR types and characteristics. For example, **InternalConsistency** is a specialization of the **Consistency** subtype of accuracy, and takes on the *Internal* characteristic.

There are several specializations of the **Accuracy** type. Some relate entities in the domain to information in the system.

Suppose the system records information that the credit account of Chris is a gold account. If Chris' account is indeed a gold account, the information in the system would have **PropertyAccuracy**. By a property accuracy requirement, we mean that an object in the system is recorded as an instance of the correct class. This requirement is written **PropertyAccuracy[Info]**, where *Info* is an object in the system. **PropertyAccuracy** is also called **ClassAccuracy**.

ValueAccuracy[Info, attribute] is a requirement that the value of an *attribute* of *Info* be accurate. For example, the information that the balance on the credit account of Chris is \$700 would have **ValueAccuracy** if \$700 is indeed the value of the balance.

OneToOneAccuracy[Info] is a requirement that an object in the system has one and only one corresponding *entity* in the application domain [Borgida85a]. For example, if Chris is represented in the system as if he were two different clients, thus stored in two different records, the information will not be accurately maintained by the system. Also, if one system record is used to refer to more than one client, say Chris and Sue, that particular information is again inaccurate.

Another kind of accuracy requirement has to do with the timeliness of information items. This addresses concerns such as "the age of a person, as recorded in the system, is within one year of the real age of the person involved" [Greenspan84] (p. 87). Thus, an information item has **TimelyAccuracy**, written **TimelyAccuracy[Info]**, if the time interval of the information in the system is faithful to the corresponding time interval in the domain. **TimelyAccuracy** is different from the **TimePerformance** type discussed in Chapters 8 and 9.

Another kind of accuracy is **Consistency**. **ExternalConsistency** is about the correspondence between values in the system and corresponding values in the application domain. In contrast, **InternalConsistency** is about the validity of relationships among various values in the system.

Each accuracy softgoal then has one of the accuracy subtypes as the softgoal type, and one or more information items as topics of the softgoal. An example is:

Accuracy[Account]

This is requirement that Account information be accurate.

As accuracy softgoals have information items as topics, satisfying such softgoals means providing a degree of confidence in the accuracy of information

items maintained by the system. Accuracy softgoals can be written in a longer form, introduced in Chapter 3, and may optionally have other attributes, such as priority and author.

Note that accuracy is treated extensionally rather than intensionally here. In other words, an accuracy requirement for, say, most-highly-rated client accounts is interpreted extensionally as a requirement for the accuracy of particular accounts maintained by the information system, rather than intensionally as a requirement for the accuracy of the description of the concept of `MostHighlyRatedClientAccount` included in the system specification. An intensional treatment of accuracy would attempt to measure the accuracy of information items such as generic descriptions; the description of the `HaveMeeting` activity, for example, may be accurate or inaccurate depending on its declared sub-activities, constraints, parameters, etc. This notion of accuracy seems appropriate when one attempts to measure the faithfulness of the *world model* — that is a part of functional requirements — to the application domain.

REFINEMENT METHODS

Refinement methods (Chapter 4) are generic procedures for refining a softgoal into one or more offspring. When collected and organized into catalogues, refinement methods can offer a body of knowledge for dealing with NFRs in general, and accuracy requirements in particular. Methods can be shared, extended, tailored and reused.

Sections 6.2 through 6.4 present refinement methods for decomposition, operationalization and argumentation of accuracy softgoals.

6.2 DECOMPOSITION METHODS

Decomposition methods provide decomposition of accuracy requirements. They also allow for different interpretations of accuracy requirements to coexist, hence accommodating different needs of different application domains.

Decomposing a parent accuracy softgoal results in a set of offspring accuracy softgoals. Type decomposition methods generate offspring whose types differ from that of the parent. On the other hand, topic decomposition methods generate offspring whose topics differ from that of the parent.

Figure 6.3 shows a catalogue of decomposition methods for accuracy softgoals. This catalogue, of course, can be tailored and extended by the developer. Sometimes method names are abbreviated. For example, method names may omit the term “Accuracy.”

Type Decomposition Methods

A type decomposition method produces offspring softgoals with different types than the parent softgoal. Some examples of type decomposition methods are:

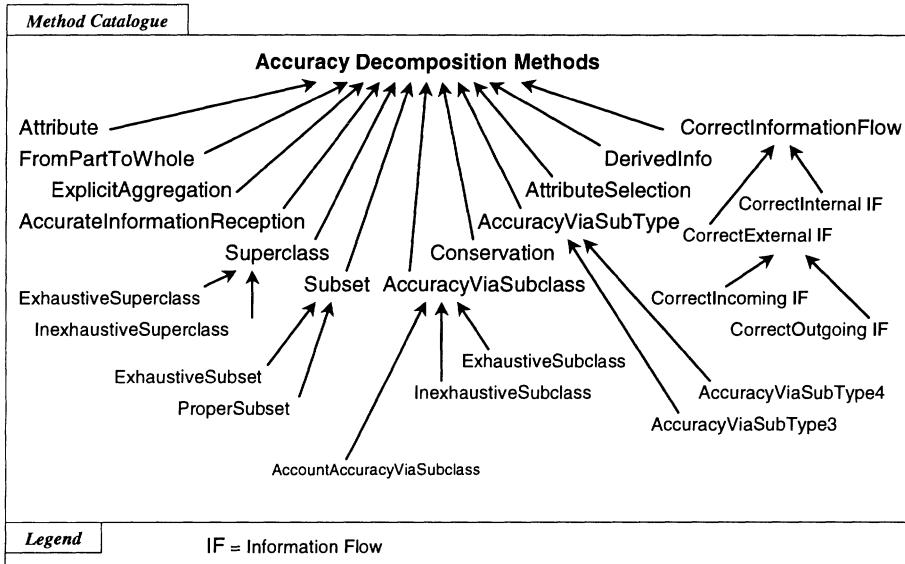


Figure 6.3. A catalogue of accuracy decomposition methods.

■ **AccuracyViaSubType** method:

To establish the accuracy of an information item, establish subtypes of accuracy, for the information item. This is a specialization of the **SubType** method, described in Chapter 4.

One specialization of this method is the **AccuracyViaSubType4** method which addresses the properties, attribute values, one-to-one correspondence, and time interval, for information items:

$$\begin{aligned}
 & \text{PropertyAccuracy[Info]} \text{ AND } \text{ValueAccuracy[Info]} \text{ AND} \\
 & \text{OneToOneAccuracy[Info]} \text{ AND } \text{TimelyAccuracy[Info]} \\
 & \text{SATISFICE Accuracy[Info]}
 \end{aligned}$$

The developer could use only some of the subtypes. For instance, the **AccuracyViaSubType3** could decompose an accuracy requirement into property, value, and timely accuracy. The particular selection of subtypes can be based on academic and industrial experience, and the needs of the domain.

Methods Addressing Information Flow.

There are type decomposition methods which deal with information flow. These decomposition methods enable developers to divide their concerns ac-

cording to the way information items are manipulated, within and outside of the system. Figure 6.4 shows kinds of information flow; it is more abstract than Figure 6.1.

In Figure 6.4, there are different types of *agents*: *source agents* from whom information originates; *intermediate agents* who receive information and further transmit it; and *destination agents* to whom the original information was intended. In Figure 6.4 “Computer System,” such as a travel expense management system, is an intermediate agent in the flow of information, *Information1*, from a source agent to a destination agent.

Of course, an agent can take on different roles, becoming a source agent for one information item at one time, an intermediate agent for another information item at another time, and a destination agent at yet another time, etc. For example, a secretary may travel to participate in a meeting and request for expense reimbursement. The secretary would be the source agent for information describing the nature of the travel, as well as an intermediate agent for retransmitting information about airfare from an airline to the central accounting office.

To generate an information item, several information items may be needed by an agent. For example, the travel expense management system may use *InformationB* (e.g., *per diem* food expense) to service another information item, *Information4* (e.g., total expense). In this case, the accuracy of a derived information item would depend on the accuracy of other information items.

Methods for Correct Information Flow.

The following methods address the correct manipulation of information items, whether the information is accurate or not.

- **CorrectInformationFlow (CIF) method:**

To establish correct manipulation of information items, establish that they are correctly manipulated by the system (*CorrectInternalInformationFlow*) and that they are correctly manipulated while they are outside the system (*CorrectExternalInformationFlow*).

$$\begin{aligned} &\text{CorrectInternalInformationFlow[Info] AND} \\ &\text{CorrectExternalInformationFlow[Info]} \\ &\text{SATISFICE } \text{CorrectInformationFlow[Info]} \end{aligned}$$

- **CorrectExternalInformationFlow method:**

To establish correct manipulation of information items while they are outside the system, establish that they are correctly manipulated from the source agent until received by the system (*CorrectIncomingInformationFlow*), and that they are correctly manipulated from the system until received by the destination agent (*CorrectOutgoingInformationFlow*).

$$\text{CorrectIncomingInformationFlow[Info] AND}$$

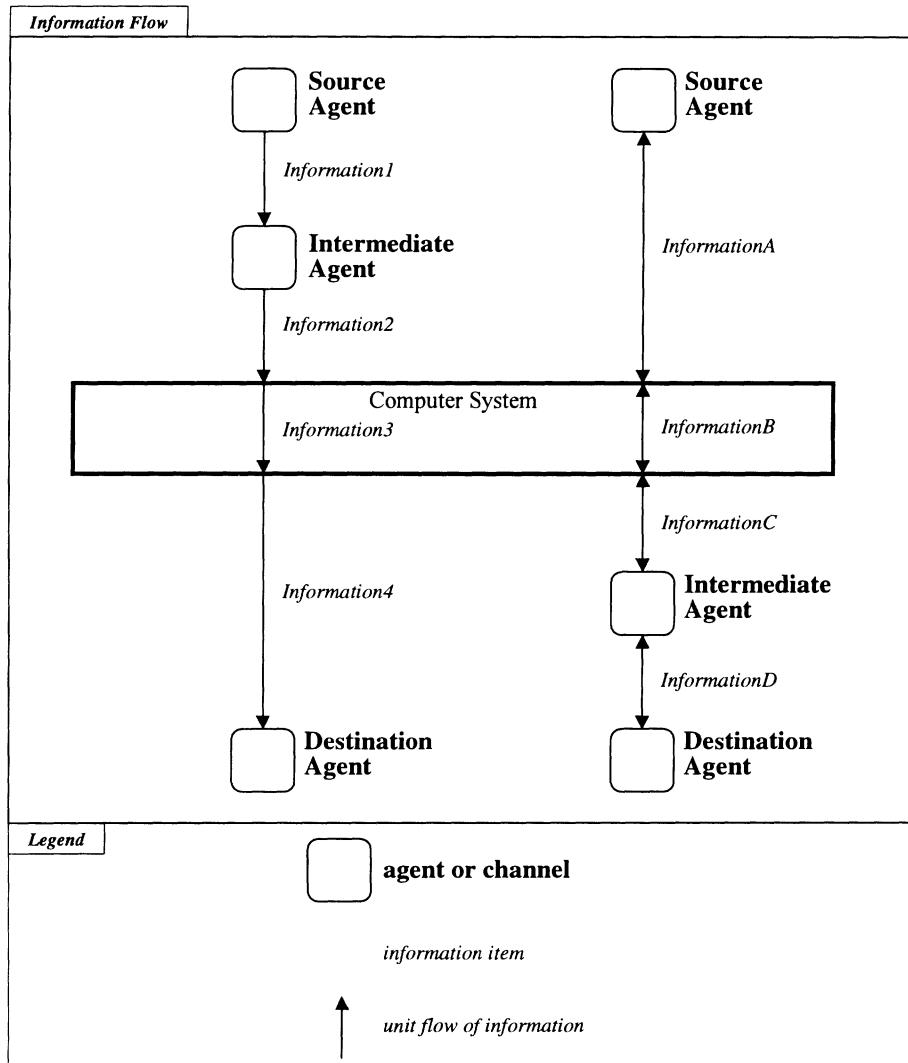


Figure 6.4. A description of information flow.

CorrectOutgoingInformationFlow[*Info*]

SATISFICE CorrectExternalInformationFlow[*Info*]

Shortly below, we will discuss the relationship of correct *internal* information flow to accuracy.

- **CorrectIncomingInformationFlow** method:

To establish correct manipulation of information items obtained from source agent until received by the system, establish that they are correctly manipulated from the transmission by the original sender until received by the *interface agent* who transmits them to the system (**CIFBetweenSourceAndInterfaceAgent**), and that they are correctly transmitted by the interface agent to the system (**CIFFFromInterfaceAgentToSystem**).

$$\begin{aligned} & \text{CIFBetweenSourceAndInterfaceAgent[Info]} \text{ AND} \\ & \text{CIFFFromInterfaceAgentToSystem[Info]} \\ & \text{SATISFICE } \text{CorrectIncomingInformationFlow[Info]} \end{aligned}$$

Similarly, for the **CorrectOutgoingInformationFlow** method, we have:

$$\begin{aligned} & \text{CIFFFromSystemToIntermediateAgent[Info]} \text{ AND} \\ & \text{CIFFFromIntermediateAgentToDestinationAgent[Info]} \\ & \text{SATISFICE } \text{CorrectOutgoingInformationFlow[Info]} \end{aligned}$$

Relating Accuracy and Information Flow.

Using the notion of information flow, it is possible to address areas of concern for accuracy by way of some general accuracy type decomposition methods. In order to relate information accuracy to correct information flow, the developer can use methods dealing with the accuracy of information at the source agent and in the system:

- **AccurateInformationReception** method:

To establish accuracy of information items received by the system, establish that they are accurate when first transmitted by the original sender (**AccurateSource**), and that they are subsequently correctly manipulated until received by the system (**CorrectIncomingInformationFlow**):

$$\begin{aligned} & \text{AccurateSource[Info]} \text{ AND } \text{CorrectIncomingInformationFlow[Info]} \\ & \text{SATISFICE Accuracy[received(Info)]} \end{aligned}$$

- **Conservation** method:

To establish the accuracy of a collection of information items currently in the system, establish (i) their accuracy, when received by the system from some external agent, and (ii) their correct internal manipulation by the system. In other words, information in the system is accurate if the information was

accurate when received, and the system's internal processing has internally *conserved* the accuracy.

Accuracy[received(*Info*)] AND CorrectInternalInformationFlow[*Info*]
SATISFICE Accuracy[*Info*]

This method draws on a decomposition [ITSEC91] of Accuracy into ExternalConsistency and InternalConsistency.

- **DerivedInfo** method:

To establish the accuracy of information items, establish that the function *f* which derives them is correctly designed and implemented, and that the function's source parameters are accurate:

DecompositionMethod DerivedInfo
 parent: Accuracy[*Info*]
 offspring: {CorrectDerivFn[*f*, *Info*], AccurateParameters[*f*]}
 contribution: AND
 applicabilityCondition: *Info* = *f*(*Info*₁, ..., *Info*_{*n*})

In turn, **AccurateParameters[*f*]** means:

Accuracy[*Info*₁] AND ... AND Accuracy[*Info*_{*n*}].

This method can be applied to those accuracy requirements whose topics are derived from other information items. An example is the **computeAmount** function shown later in Figure 6.8.

Topic Decomposition Methods

As described in Chapter 4, topic decomposition methods relate topics of a parent softgoal to topics of its offspring. Topics of NFRs are often drawn from the functional requirements. For large information systems, functional requirements are often organized using a number of “structural axes” of conceptual modelling, such as specialization and aggregation.

When NFR softgoals are revised via topic decomposition methods, different aspects of these structural axes (or organizational primitives) can be explored by the developer. Different kinds of softgoals (e.g., accuracy, performance or security) can equally be refined this way. Let us consider the case of accuracy softgoals.

Decompositions along the Generalization—Specialization Axis.

Chapter 4 introduced some generic topic decomposition methods, including the **Subclass** method. To deal with accuracy softgoals, this method is specialized to the **AccuracyViaSubclass** method: to establish the accuracy of a

class of information items, establish the accuracy of each immediate specialization of the class.

DecompositionMethod AccuracyViaSubclass

```

parent: Accuracy[to]
offspring: {Accuracy[to1], ..., Accuracy[ton]}
contribution: AND
applicabilityCondition: to: InformationClass
constraint: forAll i: toi isA to
    and /* set up one offspring for every subclass
        of the parent topic */

```

While the **Subclass** method is parameterized on the softgoal topic, **AccuracyViaSubclass** fixes the topic to be **Accuracy** (or one of its subtypes).

If the union of the extensions of the subclasses contains all instances of the general class, we say that the use of subclasses is exhaustive, and the **AccuracyViaSubclass** method is specialized into the **AccuracyViaExhaustiveSubclass** method with a contribution type of **AND**.

Otherwise, the method is specialized into the **AccuracyViaInexhaustiveSubclass** method and we will need a different contribution type. Suppose that the three subclasses of **TravelExpense** (**MeetingExpense**, **ProjectExpense** and **MemberExpense**) do not contain all the instances of **TravelExpense**. Then establishing the accuracy of the three subclasses is necessary but not sufficient to establish the accuracy of **TravelExpense**. This is represented by the three subclasses making an **AND** contribution to their (intermediate) parent, which in turn **HELPS** **Accuracy[TravelExpense]**. This combined contribution is called **AND_HELPs**, and was described in Figure 4.33. **AND_HELPs** is also used for other NFRs, such as performance.

While **AccuracyViaSubclass** examines specializations, the **AccuracyViaSuperclass** method allows for navigating the structure of information items along the generalization hierarchy.

Decompositions along the Aggregation—Decomposition Axis.

Several generic topic decomposition methods are described in Chapter 4. They can be specialized to deal with accuracy softgoals. For example, the **Attributes** method refines an NFR softgoal for an item into several NFR softgoals, each dealing with one attribute of the item. Here the method is specialized to the **AccuracyViaAttributes** method. Using this method, to establish the accuracy of a topic, establish the accuracy of each attribute of the topic. This method enables the developer to navigate the structure of information items along the aggregation dimension.

Other topic decomposition methods include:

- **AttributeSelection** method:

To establish the accuracy of an information item obtained by a sequence of

attribute selections, establish the accuracy of each information item obtained in the sequence. For example, to establish the accuracy of `Chris.account.balance`, first establish the accuracy of `Chris.account` and then of `(Chris.account).balance`. This attribute selection assumes that there are classes of customers (`Customer`) and accounts (`Account`), as well as a class of legitimate account balance values.

- **ExplicitAggregation** method:

To establish the accuracy of an information item whose value is derived from a “source” information item by a simple assignment, make the source an attribute of the derived item and use the `attributeSelection` method. Suppose that the date an expense is incurred (`Expense.date`) is derived from the date specified in a reimbursement request form (`reimbursementRequest.date`). Now to establish `Accuracy[Expense.date]`, establish `Accuracy[Expense.reimbursementRequest.date]` after making `reimbursementRequest` an attribute of expense information. This way, any change to the date in a reimbursement request gets automatically propagated to the relevant expense date.

- **FromPartToWhole** method:

To establish the accuracy of an attribute of an information item, establish the accuracy of the information item in its entirety. This method is in contrast to the attribute method which decomposes the whole into parts. This method is intended to shift focus from individual attributes to the set of all the attributes of a class of information items.

Note that the generic versions of these methods also apply to other NFRs, including performance and security.

Other Decomposition Methods.

Topic decomposition methods along the classification or context hierarchy can also be introduced when needed.

Other types of topic decomposition methods are also possible. One example is the `Subset` method:

- **Subset** method:

To establish the accuracy of a set of information items, establish the accuracy of each subset of information items.

If the union of the subsets includes all members of the set, this method is specialized to `AccuracyViaExhaustiveSubset`, with contribution type `AND`. Otherwise, the `AccuracyViaProperSubset` method is used, with contribution type `AND_HELPs`.

- **Superset** method:

The `AccuracyViaSuperset` method is also available.

6.3 OPERATIONALIZATION METHODS

As described in Section 4.3, the know-how for satisfying non-functional requirements can be captured and encoded as operationalization methods, and compiled into catalogues. Accuracy operationalization methods refine accuracy softgoals into *accuracy operationalizing softgoals* which are intended to enhance the level of our confidence in information accuracy.

For instance, the accuracy of information about high transportation expenses, `Accuracy[TransportationExpense.highSpending]` could be aided by periodic auditing of expense databases. This knowledge can be encoded as an operationalization method, shown earlier in Figure 4.21:

OperationalizationMethod Auditing

```
parent: Accuracy[Info]
offspring: Audit[Info]
contribution: SOME+
applicabilityCondition: Info in Database
```

For another example, the `Validation` method can refine a more specific accuracy softgoal, such as `CorrectInfoFlow[Info]`, into an accuracy operationalizing softgoal, `Validation[Info]`. For example,

```
Validation[TransportationExpense.highSpending] HELPS
CorrectInfoFlow[TransportationExpense.highSpending]
```

Following procedural guidelines for validation and performing comprehensive checking of `Info` will help the accuracy of `Info` at the source and correct manipulation during the transmission from this source to the system.

To avoid introducing numerous distinct names, we use the convention that the names of an operationalization method and the type of the operationalizing softgoals it introduces can be identical (here, `Validation`).

Accuracy operationalization methods can refine accuracy softgoals into accuracy operationalizing softgoals. These methods can also refine accuracy operationalizing softgoals into other accuracy operationalizing softgoals.

Methods for Changing the Processing of Information

We take the premise that the accuracy of information items depends entirely on the process in which they are manipulated within the system and its environment. As a consequence, our accuracy operationalizing methods alter the information manipulation process. James Martin [Martin73], for instance, offers a glossary of techniques for improving accuracy. Figure 6.5 shows a catalogue of accuracy operationalization methods.

Some examples of accuracy operationalization methods are:

- **Confirmation**

The informant, either a machine or a person, double-checks the previously-submitted information item. This method can be specialized to `ConfirmationViaIdenticalChannel`, if the transmissions for both the first and the second

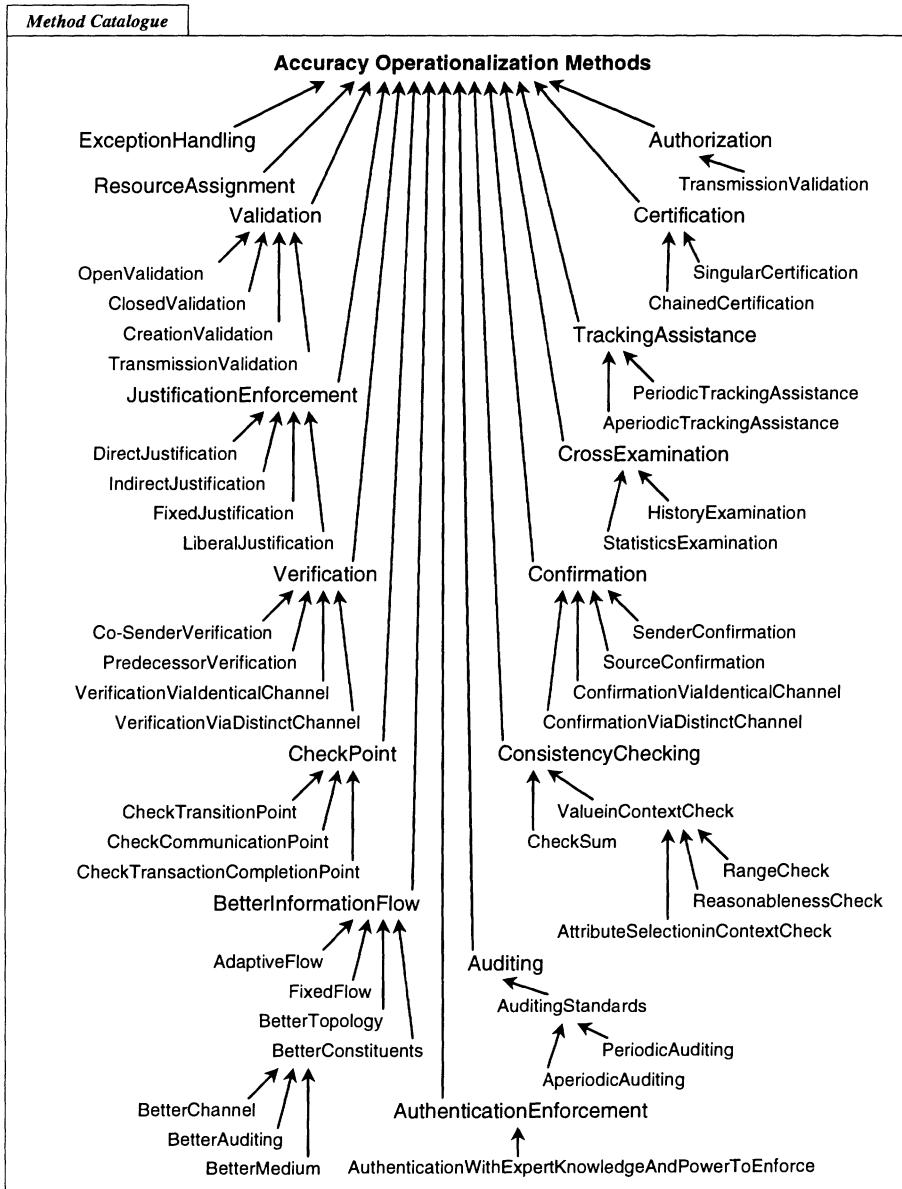


Figure 6.5. A catalogue of accuracy operationalization methods.

checks take place using the same channel, and **ConfirmationViaDistinctChannel** (e.g., via a daisy-channel), otherwise.

- **Verification:**

A *verifier*, who is a co-worker of the sender of certain information item makes a duplicate entry of this item, possibly in a separate recording device, into the system (e.g., via duplicate key-entry operations). As with **Confirmation**, this method can be specialized to **VerificationVialdenticalChannel** and **VerificationViaDistinctChannel**.

- **Validation:**

A *validator* performs checking of certain information item, using certain records or procedural guidelines to ensure that this item meets predetermined standards. The kind and thoroughness of the checking is specified in specialized methods. The **CreationValidation** method requires an agent to directly contact the information source and ensure that the information item at the system is identical with that at the source. Drawing on [Wino-grad86], other specializations of this method include **Experimentation**, which requires conducting a test to determine the validity of the information item, **TransmissionValidation**, which requires inspecting transmissions to unveil the presence of some faulty substance, and **LogicalValidation**, which requires deducing formal proofs of claims.

- **Certification:**

The **Certification** or **ExpertConsultation** method seems to be a frequently used method in a variety of organizations. A *certifier*, with a high ranking being regarded as highly reliable (usually with expertise in a certain area), assumes certain responsibilities for the future. For example, a bank manager may issue a letter of credit, thus making a commitment to make payments to others in the future.

- **Authorization:**

An *authorizer* grants the receiver of certain information items authority to transmit them to the system.

- **Audit:**

An *accuracy auditor* uses procedures to periodically go through suspicious sampled information items.

A specialization of this method is **InternalAuditing**, in which the system checks the validity of information items against a set of inter-related constraints and reports suspicious information items. [Svanks81] reports that internal auditing enabled the discovery of several welfare “claimants” who were in fact deceased.

- **ConsistencyChecking:**

To prevent frequently-occurring errors, the system enforces certain integrity constraints. For example, check-sums incorporated into ISBNs protect against transposition of digits.

Various specializations of this method are used to satisfy internal consistency, a subtype of accuracy [ITSEC91].

These methods can be specialized in accordance with the kind of agent which performs the needed task, the mode of checking (e.g., interactive vs. batch), the presence and kind of evidence attached, the time of checking (e.g., input or output time), etc.

A Categorization of Accuracy Operationalization Methods

Accuracy operationalization methods can be used for *precautionary*, *preventive* or *curative* purposes:

Precautionary Accuracy Operationalizing Methods:

These methods can be used to reduce the chances of occurrence or recurrence of inaccurate information items, which are caused by faulty transmission. For example, the **BetterInfoFlow** method and its variants are intended to achieve more reliable information flow by upgrading what is involved in information transmission, such as senders, receivers, and communication channels.

Preventive Accuracy Operationalizing Methods:

To prevent inaccuracies, these methods usually require direct interaction between the system and agents in the application domain. These methods are carried out at the time information items are received by the system. For example, the **Confirmation** and **Verification** methods are intended to detect inaccuracies of information items and prevent them from permeating the system.

Curative Accuracy Operationalizing Methods:

These methods may either disallow and avoid possibly inaccurate information items, or identify them as such and allow them to persist, as in [Borgida85b]. These methods can be carried out whenever inaccuracies are suspected, including information processing and output time. For example, the **ExceptionHandling** method can be used to take certain measures against any further propagation of inaccuracies. In order to detect the source of errors and to recover from inaccuracies, the **CheckPoint** and **Validation** methods can be used to trace inaccuracies backward to their transmission paths and sources.

Resource-Related Operationalization Methods

In order to use some of the above methods, the developer needs to allocate certain resources in the application domain and assign them to the tasks associated with the methods. For instance, there is the **ValidationResourceAvailability** method (Figure 6.6):

AvailableInfo[*Info*] AND

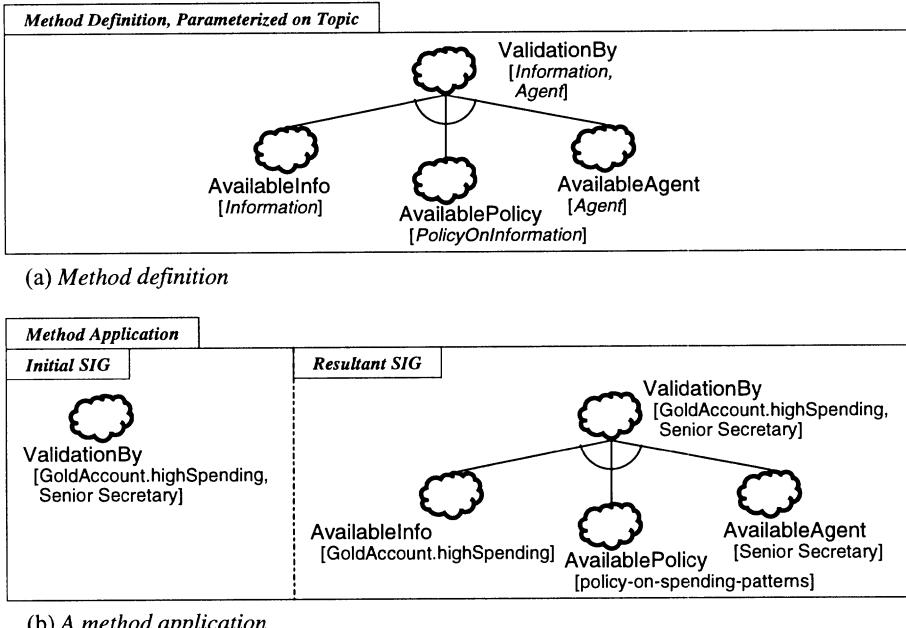


Figure 6.6. The ValidationResourceAvailability method.

$$\begin{aligned}
 & \text{AvailablePolicy[PolicyOnInformation]} \text{ AND} \\
 & \text{AvailableAgent[Agent]} \\
 & \text{SATISFICE Validation[Info, Agent]}
 \end{aligned}$$

To validate information, three resources should be available: knowledge about the information, a policy which states predetermined standards on permissible values of the information, and an agent to perform the task.

For instance, validation can be carried out by a senior secretary (Figure 6.6), or by a junior secretary who compares expense vouchers:

$$\begin{aligned}
 & \text{AvailableInfo[SummaryAndVouchers]} \text{ AND} \\
 & \text{AvailablePolicy[policy - on - voucher - comparison]} \text{ AND} \\
 & \text{AvailableAgent[JuniorSecretary]} \\
 & \text{SATISFICE ValidationBy[SummaryAndVouchers, JuniorSecretary]}
 \end{aligned}$$

This method application takes an operationalizing softgoal as the parent and produces other ones as offspring.

Agents can be made available by assigning new or existing resources to them. Granting new resources can be operationalized by `NewAcquisition`. Existing resources, possibly under-utilized can be assigned via `WorkloadIncrease`.

`ValidationResourceAvailability` is a specialization of the `ResourceAvailability` method:

$$\begin{aligned} & \text{AvailableInfo[Info]} \text{ AND} \\ & \text{AvailablePolicy[Policy]} \text{ AND} \\ & \text{AvailableAgent[Agent]} \\ & \text{SATISFICE } \text{NFRType[Info, Agent]} \end{aligned}$$

6.4 ARGUMENTATION METHODS

Argumentation methods are used to support or deny the use of accuracy refinement methods, as well as prioritization. For instance, treating the accuracy of information about foreign travel expenses as critical can be sufficiently justified via a specialization of the `VitalFewTrivialMany` prioritization template, introduced in Chapter 4:

$$\begin{aligned} \text{Claim} \quad & [“\text{Foreign travel costs more than domestic;} \\ & \quad \text{treat foreign travel expense as critical}”] \\ \text{MAKES} \quad & (\text{!Accuracy[ForeignTravelExpense}\}\{\text{critical}\}) \\ & \quad \text{MAKES Accuracy[ForeignTravelExpense])} \end{aligned}$$

Here the argument is used to justify the prioritization of parent `Accuracy[ForeignTravelExpense]` into offspring `!Accuracy[ForeignTravelExpense]\{\text{critical}\}`. The first `MAKES` indicates that the claim satisfies the interdependency link which is enclosed in parentheses. The second `MAKES` indicates that satisfying the critical accuracy softgoal will satisfy `Accuracy[ForeignTravelExpense]`. Recall Figure 4.30 which provides a template for prioritization.

Figure 6.7 shows a catalogue of argumentation methods. Note that several, such as `PolicyManualConsultation` and `Prioritization`, are generic and can be applied to a variety of NFRs (e.g., accuracy, performance and security). Let's consider some of the methods.

- **PreferentialSelection:**

Select a method which helps meet preferred softgoals. For instance, if an operationalizing softgoal helps or makes one accuracy softgoal but hurts or breaks a second accuracy softgoal, a pre-existing preference for the first softgoal over the second would be a positive argument for selecting this operationalization method.

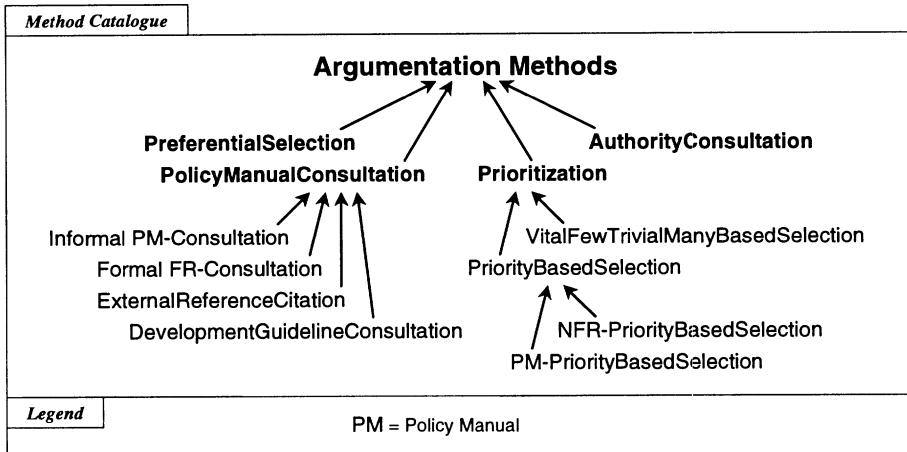


Figure 6.7. A catalogue of accuracy argumentation methods.

■ **PolicyManualConsultation:**

When a question arises about the applicability of various types of methods, consult policy manuals in the application domain.

■ **PriorityBasedSelection:**

Select a method in accordance with their relative priority. Suppose an operationalizing softgoal is good for one accuracy softgoal but bad for a second. Now if the first has greater priority than the second, this would be an argument for selecting the first operationalizing softgoal. The above PreferentialSelection method is similar to this method, but does not necessarily depend on priority, but can be related to users' or developers' taste.

■ **SynergisticSelection:**

(This method is not shown in the figure.) Select a method if two or more softgoals can collectively justify its application. Each one alone may be insufficient to justify the application. As we will see in Chapter 8, this principle is also applicable to performance requirements, where a “family plan” retrieval obtains several pieces of information in one operation.

6.5 CORRELATIONS

As described in Section 4.5, correlation rules catalogue knowledge about interactions between and among NFR softgoals, operationalizing softgoals and claim softgoals. Such knowledge is compiled into a catalogue of correlation

rules for reuse, tailoring, and extension. Some correlation rules were described in Section 4.5, such as `FlexibleUserInterfaceHURTSAccuracy`.

Let's consider some other accuracy correlation rules, used to describe situations in application domains. Consider the `Validation` operationalization method which can be applied to information about expense date, amount, location, etc., in order to enhance our level of confidence in the accuracy of such information. This method can cause delays, as it takes time to carry out the validation process, and consequently make information items less timely. This negative interaction with timely accuracy requirements is written:

```
Validation[Info] HURTS TimelyAccuracy[Info]
WHEN excessive(duration(validationActivity(Info)))
```

Correlation rules can lead to a large number of interdependencies in a softgoal interdependency graph. A condition can be used to control the invocation of rules. In the above example, the condition follows the `WHEN` keyword. This correlation can also be written in a longer format:

```
CorrelationRule ValidationHURTSTimelyAccuracy
parent: TimelyAccuracy[Info]
offspring: Validation[Info]
contribution: HURTS
condition: excessive(duration(validationActivity(Info)))
```

Here, the developer can estimate the duration of the `validationActivity`, after considering the person and documents involved. If the duration seems prolonged enough to cause significant delays for the users of the system, the developer can set `excessive(duration(...))` to true. This will activate the correlation rule.

Situation Descriptors

Note that `excessive(...)`, `duration(...)`, and `validationActivity(...)` are not NFRs. Rather they are *descriptors of situations*. Here they are used to form *conditions* for a correlation rule. They can also be used within methods and claims. To syntactically distinguish them from NFRs, situation descriptor names start with a lower-case letter, and their parameters are placed within parentheses. Note that situation descriptors can be used when dealing with a *topic* of an NFR, as in `validationActivity(Info)`.

These are examples of correlation rules which allow for the capture of interactions between operationalizing softgoals and accuracy softgoals.

Conditions can also specify relationships between topics of various softgoals, using situation descriptors. Relationships such as `isA`, `instanceOf`, `subsetOf` and `supersetOf` can be used in situation descriptors. For example:

```
Validation[Info'] HURTS TimelyAccuracy[Info]
WHEN   excessive(duration(validationActivity(Info)))
       and   Info' isA Info
```

The WHEN condition requires the topic of the Validation softgoal to be a subclass of the topic of the TimelyAccuracy softgoal. This generates a variety of topics which meet the condition.

Correlation Catalogues

Correlation Catalogue		
Contribution of offspring <i>Operationalizing Softgoal</i>	to parent NFR Softgoal	
	Accuracy [Info]	Security [Info]
ValidationBy [Agent, Info]	HELPS WHEN not cond1 HURTS WHEN cond1	HELPS WHEN not cond2 BREAKS WHEN cond2
MutualID [Agent, Info Procedure, Time]	HELPS	HELPS

cond1: excessive(duration(validationActivity(Info)))

cond2: accessDisallowed(Agent, Info)

Table 6.1. An accuracy correlation catalogue.

Correlation rules are collected in *correlation catalogues* (Table 6.1). Catalogue entries can be of the form:

contribution

This indicates the *contribution* of the offspring to the parent. This particular catalogue shows the contributions of operationalizing softgoals to accuracy softgoals and other NFR softgoals. Entries can also be of a more general form:

contribution WHEN condition

When the *condition* holds, these entries provide the *contribution* of the offspring to the parent.

In addition to operationalizations for accuracy, other operationalizations, say for security, can be positive or negative for an accuracy softgoal. These can also be recorded in a correlation catalogue. Consider the security operationalizing softgoal MutualID[Agent, Info, Procedure, Time], used when an Agent attempts to access information item *Info*. To mutually ensure their identities, the Agent and the system process use a test *Procedure* during a *Time* interval. The procedure requires alternating queries and answers by the two; this is similar to the *challenge response* process [Pfleeger89]. This would be positive

for accuracy softgoals if mutual identification prevents a malicious user from penetrating the system and falsifying information.

6.6 ILLUSTRATION

In this section, we illustrate how accuracy requirements are addressed for the example of the travel expense management system.

Initial Accuracy Requirements and Functional Requirements

Suppose that “All travel expenses must be accurate” is an initial accuracy requirement. The developer can represent this requirement as an NFR softgoal: **Accuracy[TravelExpense]**.

In addition to accuracy requirements, other NFRs may also be stated. For example, one security requirement is that junior staff not have access to reimbursement information.

In the functional requirements, **TravelExpense** has three subclasses: **MeetingExpense**, **ProjectExpense** and **MemberExpense**. Attributes of these classes are also specified. One of them, **ProjectExpense.amount**, is derived by a derivation function **computeAmount(Expense.project, Expense.date, ProjectExpense.month)**. More specifically, the amount of each project expense is the sum of all those expense items which belong to the particular project and whose date falls within the month of the project expense item.

In addition, company policy limits the time that managers spend on reimbursements.

Method Applications

At this point, the developer might feel that the topic of the expression is too coarse-grained to apply any specific accuracy operationalizing softgoals. Thus, the developer applies the **Subclass** method and decomposes the accuracy softgoal **Accuracy[TravelExpense]** into three offspring softgoals, which correspond to the three subclasses of **TravelExpense**.

$$\begin{aligned} & \text{Accuracy[MeetingExpense]} \text{ AND} \\ & \text{Accuracy[ProjectExpense]} \text{ AND} \\ & \text{Accuracy[MemberExpense]} \\ & \text{SATISFICE Accuracy[TravelExpense]} \end{aligned}$$

This now allows the developer to deal with accuracy concerns for the subclasses of travel expenses. The above decomposition is illustrated at the top of Figure 6.8. The legend for figures is given at the front of this book.

Now each of these offspring needs to be satisfied. Focussing on the offspring **Accuracy[ProjectExpense]**, the developer applies the **AccuracyViaIndividualAttributes** method and decomposes the softgoal into the accuracy of the

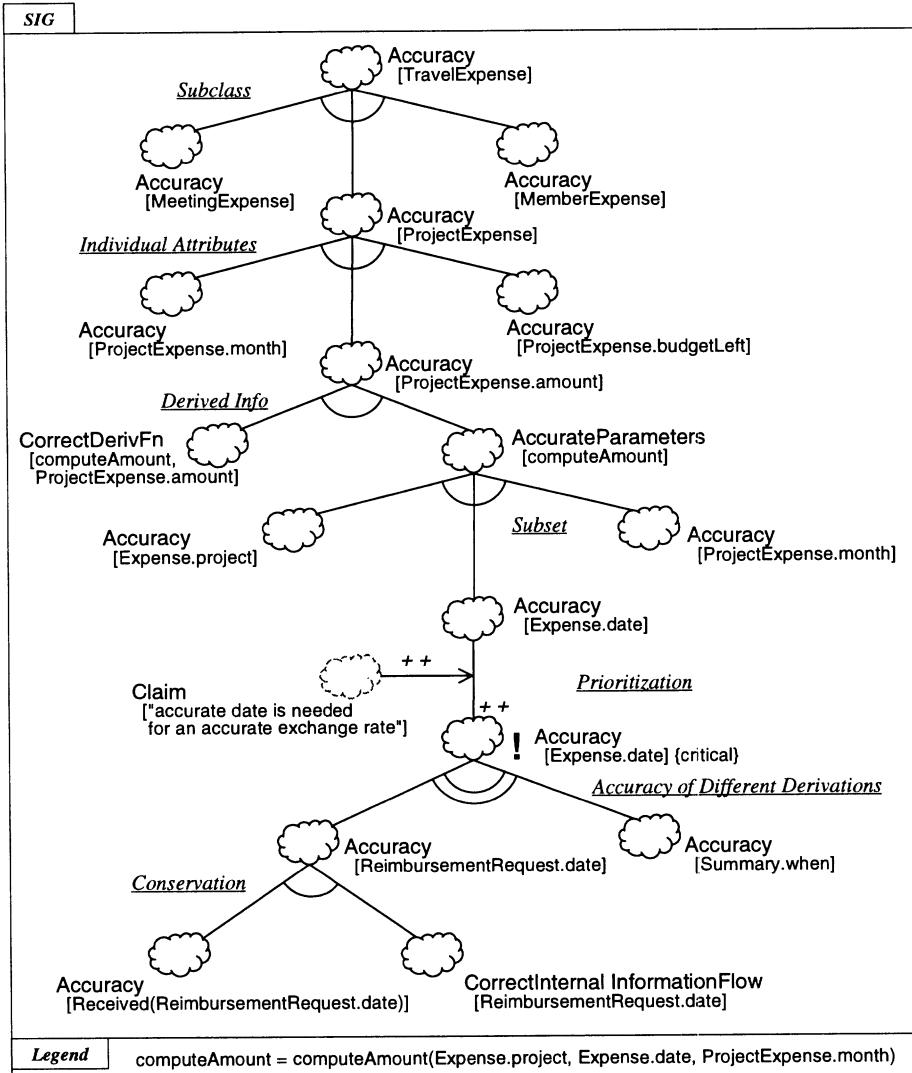


Figure 6.8. Decompositions for accurate travel expenses.

individual attributes:

Accuracy[ProjectExpense.month] AND
 Accuracy[ProjectExpense.amount] AND

$\text{Accuracy}[\text{ProjectExpense.budgetLeft}]$
 SATISFICE $\text{Accuracy}[\text{ProjectExpense}]$

The developer first considers expense information, and focusses on the accuracy of total monthly project expenses, $\text{Accuracy}[\text{ProjectExpense.amount}]$. Since the functional requirements specify that $\text{ProjectExpense.amount}$ is *derived* by a derivation function $\text{computeAmount}(\text{Expense.project}, \text{Expense.date}, \text{ProjectExpense.month})$, the developer instantiates the `DerivedInfo` decomposition method. This method decomposes $\text{Accuracy}[\text{ProjectExpense.amount}]$ into two offspring, one for correctly designing the function, and the other for the accuracy of the parameters of this function (Figure 6.8):

$\text{CorrectDerivFn}[\text{computeAmount}, \text{ProjectExpense.amount}] \text{ AND}$
 $\text{AccurateParameters}[\text{computeAmount}($
 $\text{Expense.project}, \text{Expense.date}, \text{ProjectExpense.month})]$
 SATISFICE $\text{Accuracy}[\text{ProjectExpense.amount}]$

Now the developer takes the second step of the `DerivedInfo` method. To examine the parameter list of `computeAmount`, the developer applies the `Subset` decomposition method to $\text{AccurateParameters}[\text{computeAmount}]$:

$\text{Accuracy}[\text{Expense.project}] \text{ AND}$
 $\text{Accuracy}[\text{Expense.date}] \text{ AND}$
 $\text{Accuracy}[\text{ProjectExpense.month}]$
 SATISFICE $\text{AccurateParameters}[\text{computeAmount}]$

One of the parameters of `computeAmount`, the date when the expense was incurred, requires careful treatment, since this information is used in currency conversion, which is required for most of the expenses. In order to reflect this, the developer treats the accuracy of date information as being critical, written $!\text{Accuracy}[\text{Expense.date}]\{\text{critical}\}$. This prioritization, indicated by “!,” can help reduce the search space in expanding the softgoal interdependency graph, and facilitate conflict resolution.

Two alternatives are foreseen by the developer in obtaining this critical information. The developer uses the `AccuracyOfDifferentDerivations` method to indicate that the information may come from:

- (a) the expense reimbursement requests (by requiring the members to send their reimbursement request forms to the central management office), or
- (b) the expense summary (by requiring the secretary to submit it directly).

This disjunction,

$\text{Accuracy}[\text{ReimbursementRequest.date}] \text{ OR } \text{Accuracy}[\text{Summary.when}]$
 SATISFICES $!\text{Accuracy}[\text{Expense.date}]\{\text{critical}\}$

is shown near the bottom of Figure 6.8.

The functional requirements indicate that **ReimbursementRequest** should be received from an external agent. To ensure its accuracy, the developer uses the **Conservation** method. The accuracy of **ReimbursementRequest.date** depends on its accuracy at the time it is received, and on correct internal processing by the system afterwards:

Accuracy[received(**ReimbursementRequest.date**)] AND
 CorrectInternalInformationFlow[**ReimbursementRequest.date**]
 SATISFICE Accuracy[**ReimbursementRequest.date**]

Instead of focussing on accuracy of just the date, the developer considers all the information in reimbursement request forms received. This is done by using the **FromPartToWhole** method (Top of Figure 6.9):

Accuracy[received(**ReimbursementRequest**)]
 MAKES Accuracy[received(**ReimbursementRequest.date**)]

The developer uses the **AccurateInformationReception** method to refine the softgoal:

CorrectIncomingInformationFlow[**ReimbursementRequest**] AND
 AccurateSource[**ReimbursementRequest**]
 SATISFICE Accuracy[received(**ReimbursementRequest**)]

This decomposition is at the top of Figure 6.9.

Considering, Choosing and Justifying an Operationalization

Unfortunately, ensuring correct creation of information items at the source or subsequent transmissions from the source to the system is in many cases costly and impractical. Being unsure of what operationalization methods are appropriate for the application domain, the developer consults the director in charge of policies. The director prefers a policy of careful examination for those materials that are directly related to issuing a cheque. With the view that the **Validation** method conforms to this policy, the developer first decides to apply this method (middle of Figure 6.9).

Validation[**ReimbursementRequest**] HELPS
 CorrectIncomingInformationFlow[**ReimbursementRequest**].

Then the developer supports the decision by referring to the policy director's statement:

Claim [“Director of policy : Careful examination when issuing a cheque”]

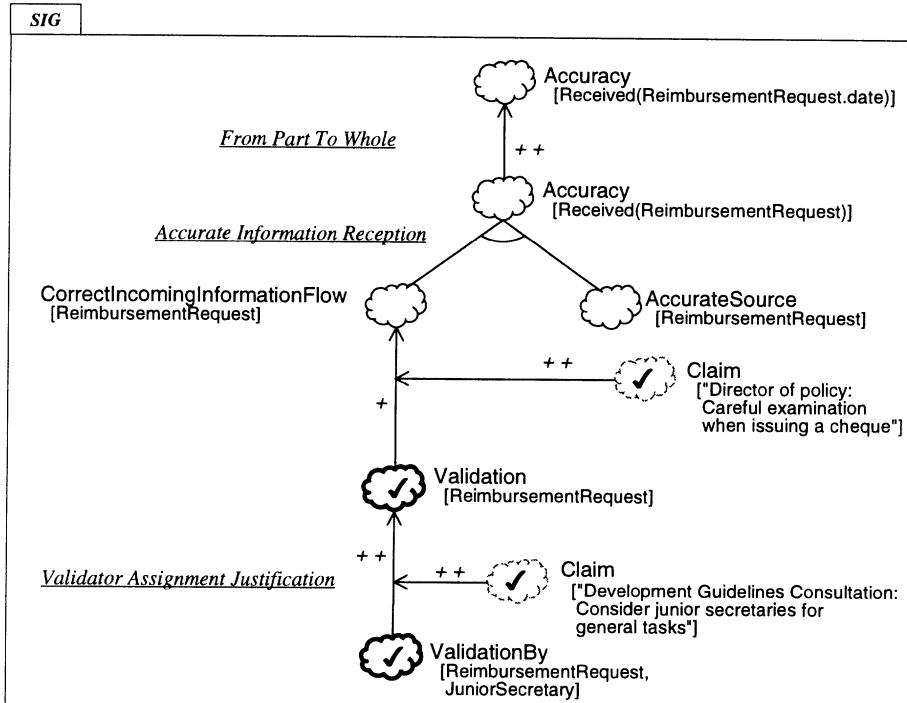


Figure 6.9. Operationalizing the accuracy of reimbursement requests.

MAKES

(Validation[ReimbursementRequest] *HELPS*
CorrectIncomingInformationFlow[ReimbursementRequest])

Here the claim **MAKES** the interdependency link, shown within parentheses.

To successfully validate the reimbursement request, resources need to be allocated and assigned to the validation task. The developer consults the development guidelines and discovers that junior secretaries are one good class of candidate for carrying out the validation, although not the only one. Thus, in allocating resources, a junior secretary could be assigned the role of validator, via the **ValidatorAssignment** method, resulting in the refined operationalizing softgoal **ValidationBy[ReimbursementRequest, JuniorSecretary]** (bottom of Figure 6.9). The assignment of the junior secretary as validator is supported by a development guideline consultation, using the **ValidatorAssignmentJustification**

template (Section 4.4):

Claim [“Development Guidelines Consultation :
Consider junior secretaries for general tasks”]

MAKES

(ValidationBy[ReimbursementRequest, JuniorSecretary]
MAKES Validation[ReimbursementRequest])

The developer selects ValidationBy[ReimbursementRequest, JuniorSecretary]. The selection of this operationalizing softgoal is shown by “ \checkmark ” (satisficed) label in Figure 6.9. Using the evaluation procedure of Chapter 3, this choice satisfies Validation[ReimbursementRequest].

While no conflict has yet been encountered for the decomposed offspring, note that later refinements of the graph may make selected operationalizations unsuitable.

We have shown the use of methods to refine softgoals. Although omitted, all the interdependencies in the SIGs of this chapter have \checkmark as their default labels, since they result from applications of catalogued methods.

Softgoals can also be directly refined by developers, using their own expertise. Our course, such refinements may then be considered for inclusion in a method catalogue.

Detecting a Conflict

In particular, the non-functional requirements disallow junior staff from accessing reimbursement information. This security requirement, shown at the left of Figure 6.10, is in direct conflict with having junior secretaries validate reimbursements:

ValidationBy[ReimbursementRequest, JuniorSecretary] *BREAKS*
Security[ReimbursementRequest, accessDisallowedBy, JuniorSecretary]

This conflict is detected by a correlation rule, which generates a new *BREAKS* contribution shown with a dashed line in the figure. (A correlation catalogue for accuracy was shown in Table 6.1.)

As a result of further evaluation in Figure 6.10, the security softgoal is labelled “ \times ” (denied), as validation of information about reimbursement requests by a junior secretary directly violates the requirement (see left-hand side of figure). This leads to more general security softgoals (top left of figure) being denied as well.

Considering an Alternative

There are many options available to the developer, when confronted with a conflicting situation like the above. Senior staff could do the validation. Other

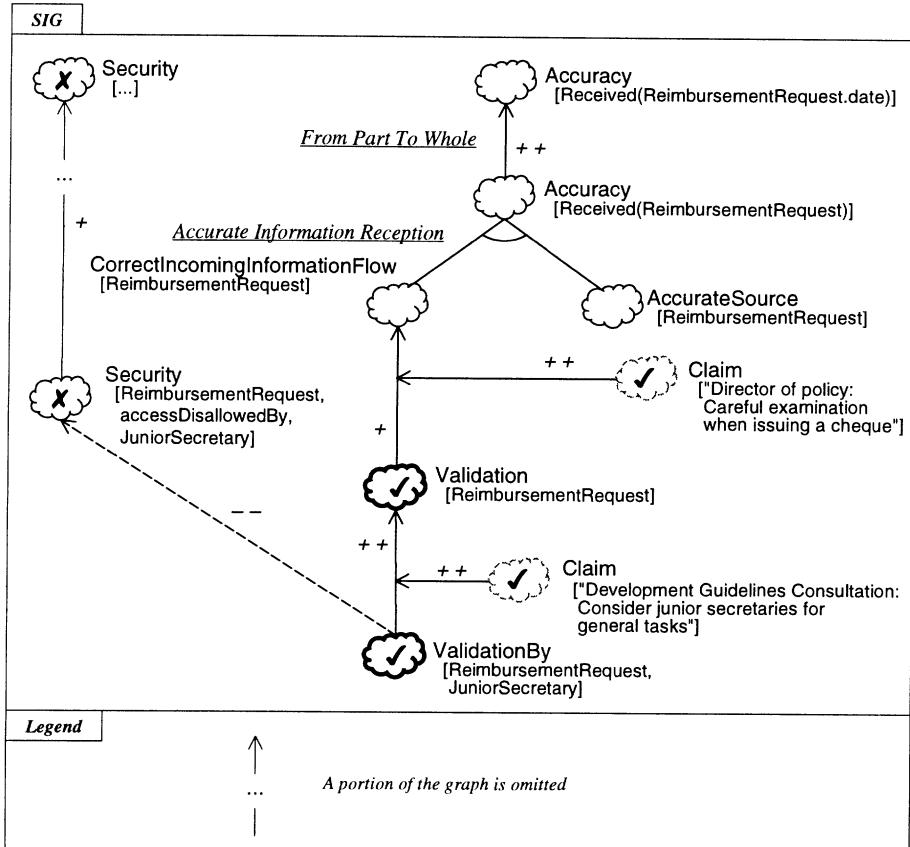


Figure 6.10. Detecting a negative impact on a security requirement.

operationalization methods can be considered: reimbursement requests could be received without validation or without any operationalization method; and the security softgoal could be relaxed.

Two alternatives for $\neg \text{Accuracy}[\text{Expense.date}]\{\text{critical}\}$ were shown near the bottom of Figure 6.8. The developer already considered $\text{Accuracy}[\text{ReimbursementRequest.date}]$ in Figures 6.9 and 6.10.

Now the developer considers $\text{Accuracy}[\text{Summary.when}]$. The developer applies the Conservation and FromPartToWhole methods. Then the developer applies the Confirmation method (resulting in $\text{Confirmation}[\text{Summary}]$ in the middle of Figure 6.11), which does not result in a conflict.

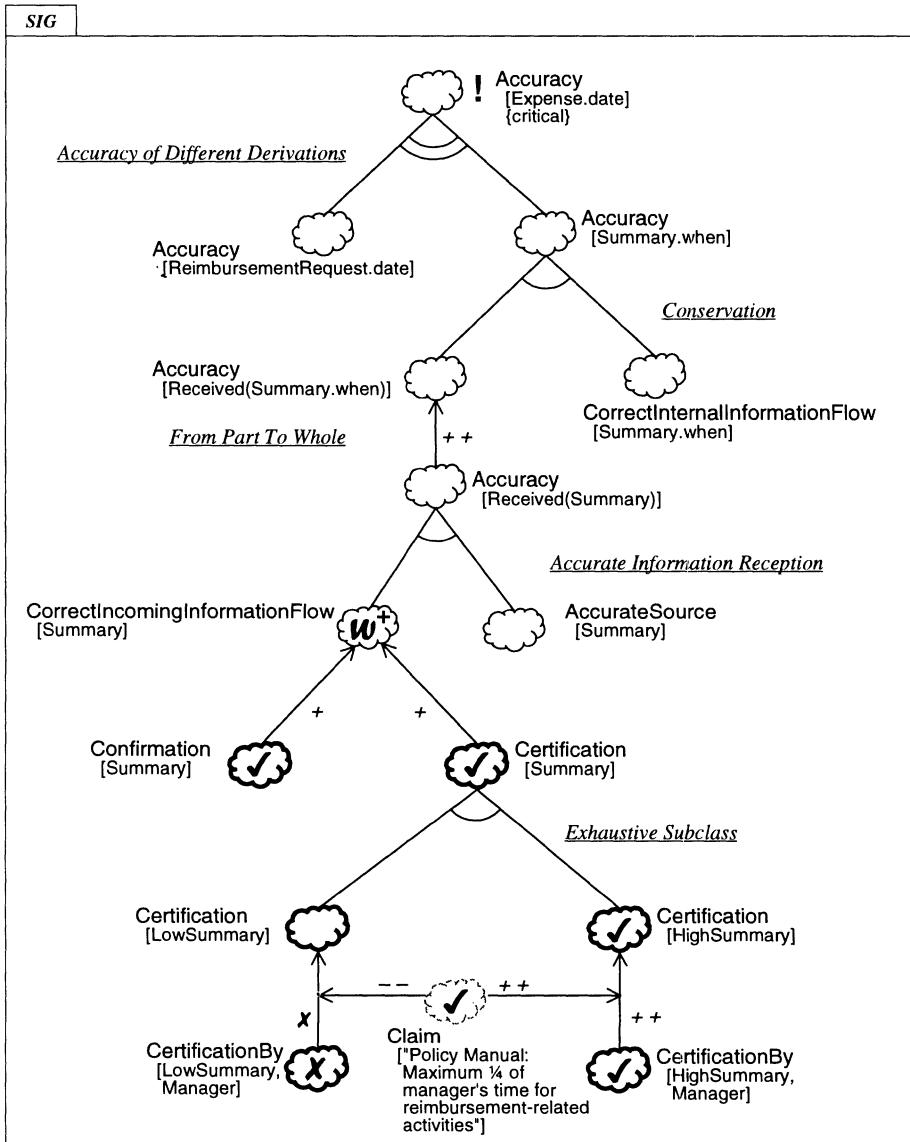


Figure 6.11. Evaluation of selective certification of expense summaries.

The developer feels that summary information would be indeed accurate, if more operationalization methods can be used. Thus, the developer applies another method, the **Certification** method, where the manager of the accounting department in the central management office is assigned the role of the certifier. Thus, two operationalization methods are used for the accuracy of the summary.

We imagine that company policy would limit the time that managers spend on reimbursement-related activities, to, say, one-quarter of their time. However, certifying all expense summaries would well exceed this limit. In resolving this situation, the developer again consults the policies and the managers to find out that managers could keep within the time limit by certifying only summaries with a high monetary value. We estimate that these constitute only 5% of all summaries. Figure 6.11 shows the design rationale recorded as a result of eliciting and acquiring knowledge about the application domain. As well, the figure shown the progress in the overall development.

We have specified the nature of the subclass refinement. Here it is exhaustive; it is also disjoint. Note that we can use this information in later steps to restrict correlations and control graph development.

Selection of an Alternative

The developer decides to have members send reimbursement requests to the central management office. To ensure accuracy, confirmation is used, along with certification of high summaries by a manager. The selected alternatives are shown with a “√” (satisficed) at the bottom of Figure 6.11. Low summaries are not certified. Rejected (denied) alternatives are shown with “×” in the figure.

To avoid security problems, the developer does not choose validation and submission by junior secretaries, which had been previously chosen in Figures 6.9 and 6.10. This is an example of *retracting* a selection.

Relating the Target System to Functional Requirements

The right hand side and bottom of Figure 6.12 relate the functional requirements for expense management to the alternatives for the target system. The target alternatives are related to the operationalizing softgoals of the SIG.

Here, the functional requirements include the maintenance of expense information. This is related to the selected target, which has summaries confirmed and high summaries certified. The selected target is related to the selected operationalizing softgoals.

Procedure Manual

The success or failure of operationalization methods relies on the cooperation between the system and agents in the environment, which is described in a *users' procedure manual*, which is initially drafted by the developer during the design process. The manual indicates policies that the agents in the environ-

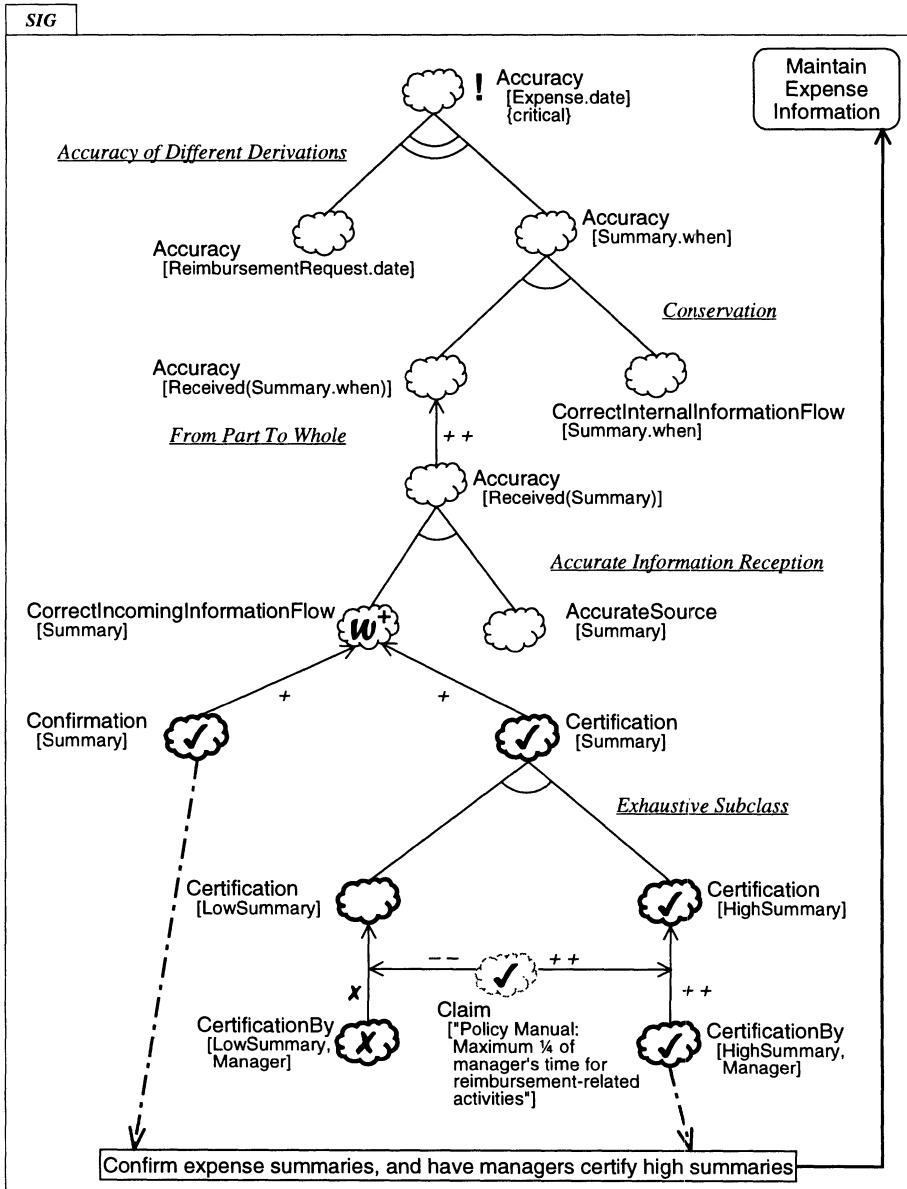


Figure 6.12. Relating functional requirements to the target system.

ment should obey when interacting with the system in order to satisfy the methods selected. For instance, if a **Validation** method is selected, the manual indicates that members must transfer their expense information to the system and to the project office which will enter the same information into the system. Reubenstein [Reubenstein90] recognizes the need for generating such manual documents, in acquiring formal requirements from possibly inconsistent or ambiguous informal descriptions.

6.7 DISCUSSION

Key Points

In this chapter, we have shown how to use the features of the NFR Framework to represent and use accuracy requirements during information system development.

This “Accuracy Requirements Framework” captures development knowledge for accuracy requirements, by cataloguing refinement methods and correlation rules from industrial and academic sources.

To deal with accuracy, a model of information flow has been provided. This helps a developer consider the accuracy of information while it moves around a system and its environment.

The travel expense management system has illustrated the use of accuracy requirements for choosing among alternatives and justifying decisions, throughout the development process. These are systematically recorded in *soft-goal interdependency graphs, which capture the complex and dynamic development process*. The graphs provide a development history which is available for later review. The development process could be quite complex, involving many alternatives and tradeoffs, and quite dynamic, as a selected helpful method can later become unsatisfactory when a conflict is encountered or a better solution is found.

This complexity and dynamicity in turn support the need for the process-oriented framework. With a final product alone, be it a design or an implementation, it would be even harder for the developer to understand what went on during the process, and consequently to maintain and upgrade the system.

Literature Notes

This chapter is based on [Chung93a] [Mylopoulos92a].

There are several sources of the methods and correlations catalogued in this chapter. Accuracy depends on the correctness of information in the system, with respect to corresponding items in the domain. This is discussed in [Borgida85a].

In [Parker91], accuracy is described as having two components: *authenticity* which reflects the faithfulness or true representation of data, and *integrity* which means wholeness or completeness of the data being maintained by the system or the data being void of missing elements. In [Motro89], database in-

tegrity is characterized by two components: *validity* which is like authenticity, and *completeness* (or *onto* aspect), the database's maintaining every relevant fact.

The notion of Value Accuracy is similar to Heninger's [Heninger80] description of accuracy constraints in hardware and software engineering as the allowable deviation between actual and ideal values. This may be contrasted with the notion of *external consistency*, the correspondence between values in the system and what they are supposed to represent in the application domain. As will be discussed in detail in Chapter 7, external consistency is one of the two notions which characterize accuracy in the area of security [ITSEC91]; the other notion is *internal consistency*, valid relationships among various values in the system. There are other similar notions of accuracy in the areas of security and database.

Categorization of information items, along with a treatment of accuracy as a partial function, is given in more detail in [Chung91a] and [Chung91b].

To resolve conflicts, a negotiation-based approach may be taken (e.g., [Robinson90], [Johnson91]). We use argumentation methods to record how conflicts are resolved, e.g., by attachment of priorities.

7

SECURITY REQUIREMENTS

One important concern in building an information system is *information security*, the protection of information as an asset of an enterprise, just like money or any other forms of property. But how do we “design in” information security?

To design information security into a system, we need to make the design process less *ad hoc* and more systematic. For such a systematic methodology, we need to address the following issues:

1. *How can a wide variety of security methods, and their tradeoffs, be made available to the developer?*
2. *How can security requirements be systematically integrated into the design?*

Without a systematic methodology, security requirements are often retrofitted late in the design process (e.g., how to securely update an account), or pursued in parallel but separately from functional requirements (e.g., how to update an account). These practices tend to result in systems which cannot be accredited, are more costly and less trustworthy [Benzel89].

This chapter shows how to systematically “design in” information security using the NFR Framework, while dealing with the above issues. The result is a “Security Requirements Framework.”

In the “information age,” information security is becoming more and more important. This is especially true for the increasingly-used Internet, which is not secure as it stands today. A pragmatic option is to understand how attacks occur and how best to protect against them [Bishop97].

We use the notion of “satisficing” in the NFR Framework, which closely corresponds to the notion of security *assurance* [E. Lee92]. The notion of “satisficing” reflects the discussion in [Moffett88] that the risk of security breaches can only be limited in magnitude, reduced in likelihood and made detectable, but *not* removed.

Section 7.1 discusses several different notions of security and how they are treated as softgoals, which then drive the overall design process. Sections 7.2 through 7.5 describe how to catalogue of methods and correlations for security. Section 7.6 illustrates the use of these components to address security requirements for an account management system. Finally, Section 7.7 summarizes key points in this chapter and presents references for further reading.

7.1 SECURITY CONCEPTS

Information security means protecting information. However, when it comes to the issue of the scope of protection, a multitude of definitions is encountered, and these diverse definitions can be a source of confusion.

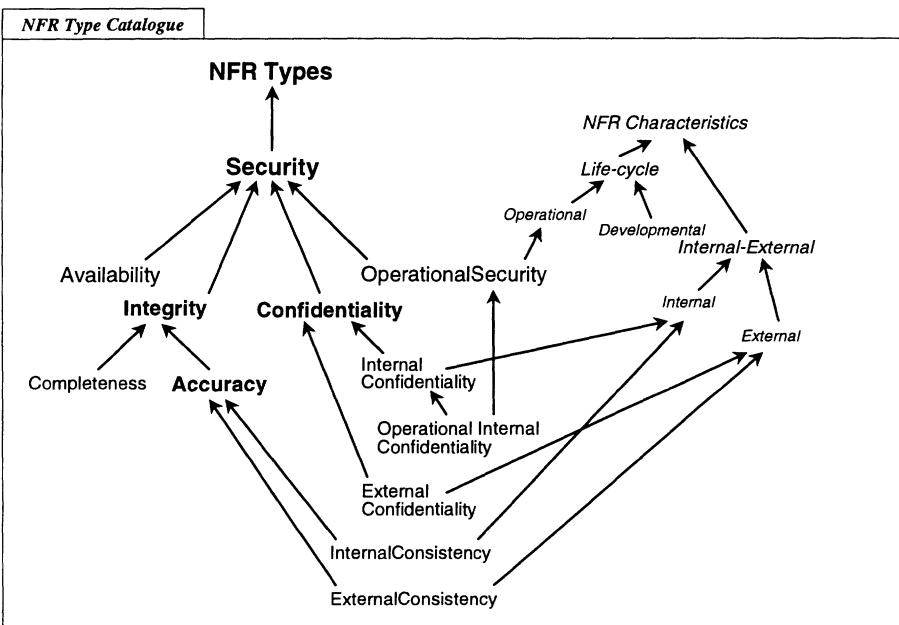


Figure 7.1. A catalogue of security types.

The NFR Framework allows for such diverse notions to be captured in *types* of security softgoals and organized in a *security type catalogue* (Figure 7.1). This type catalogue then serves as a rich set of alternatives to choose from as well as check-points to guard against omitting any important security concerns. There are some important aspects of security to consider:

- There are different emphases in definitions of security.
 - Confidentiality, guarding against unauthorized disclosure, is the primary emphasis in evaluation criteria [TCSEC85], [ITSEC89], and [ITSEC91].
 - In some commercial applications, the focus is on Integrity [Clark87], guarding against unauthorized update or tampering.
 - Availability or Assured Service, guarding against interruption of service; this, along with confidentiality and integrity, are general concerns in evaluation criteria.
 - A broader definition of security encompasses Availability as well as Authenticity, genuineness or faithfulness of true representation (comparable to “external consistency” in [ITSEC91]), Integrity, wholeness or completeness, Utility, fitting the use (e.g., a money amount in dollars, not in yen), and Confidentiality or Secrecy [Parker91].
- In addition, there are different *characteristics* of security requirements, shown on the right-hand side of the figure:
 - The primary scope of protection could be confined to information resident *internally* in the computer system, *externally*, or both.
 - The scope can be *operational*, for run-time operation, or *developmental*, for the development stage [Amoroso91].

Figure 7.1 shows subtypes of Security which have these characteristics. OperationalSecurity is one subtype, which refers to information security during system operation, while InternalConfidentiality refers to the confidentiality of information items residing in the system.

The type catalogue shows different aspects of security. Note that focussing on one subtype of security may not suffice other subtypes. For instance, focussing on the confidentiality aspect may not be sufficient to meet the accuracy needs of a specific application domain. Authorized access, meeting access-rule security requirements, does *not* necessarily preserve accuracy, due to either unintentional mistakes or intentional fraud.

Once a type is chosen, appropriate topics are needed in order to express a security softgoal. One important class of topic of a security softgoal is *information*, or information item, which seems to be appropriate as our concern lies in the development of information systems. For instance, Security[Account] is a security softgoal for our account management system, where Security is the softgoal type and Account is the softgoal topic. Account stands for all of the

accounts. This softgoal expresses the requirement that all accounts should be secure.

In addition to information items, such as *Account*, security softgoals can have other topics, such as the authorizer, access condition, delegation function ([Wood80] [Hartson76]), and task [Steinke90]. A general security softgoal would be of the form:

```
Security [Info, Authorizer, Agent,
          Task, AccessCondition,
          DelegateTo, DelegationFunction,
          FailureCondition, FailureAction]
```

where *Authorizer* specifies what *Agent* is allowed to access *Info* under *AccessCondition* during the course of *Task*. Agents may *DelegateTo* others their rights when *DelegationFunction* evaluates to true. To deal with access failures, *FailureCondition* may be specified together with some appropriate *FailureAction* to be taken.

A sample security softgoal would be:

```
Security [Account, SecurityAdministrator, Manager,
          AllTasks, Always,
          AssistantManager, LeaveOfAbsence,
          UpdateOwnInfo, NotifySecurityAdministrator]
```

Here, security administrators authorize managers to always access account information for any task, with the permission to delegate their rights to assistant managers during their leaves of absence. Employees should not update their own information (e.g., salary), and any such attempt should be reported to the security administrator.

In the rest of this chapter, security softgoals have information items as the main topic. Hence, satisfying such softgoals means assuring the security of information items maintained by the system. The security softgoals that follow often include only the information topic.

In the example used in this chapter, we will describe the application of the NFR Framework to security requirements to achieve operational security — security of information system at run-time.

REFINEMENT METHODS

There are three types of refinements: decomposition, satisfying and argumentation. Each refinement would be the result of either the instantiation of a generic method or input given explicitly by the developer. Generic refinement methods conveniently allow the capture, organization, and reuse of research results and industry experiences. The next three sections describe refinement methods for decomposition, operationalization, and argumentation of security softgoals.

7.2 DECOMPOSITION METHODS

Security requirements are often ambiguous and invite many interpretations from different groups of people. Decomposition methods help analyze security softgoals and clarify their meaning. They also link security requirements with other NFRs.

Decomposition methods also facilitate separating more sensitive information from less sensitive, thereby avoiding a single but costly strategy that *uniformly* protects *all* types of information. They help to *selectively* add security enforcement features on top of a particular type of operating system which might be adopted to meet a target security level. Examples of *security levels* (security classes) include *mandatory security* (the system is responsible for enforcing access control based on the assignment of *sensitivity* labels to information and clearance levels to users) and *discretionary security* (the user/owner is responsible for access control).

Reflecting the traditional wisdom of structural “divide and conquer,” decomposition methods also alleviate the extreme difficulty with the design for very large requirements descriptions and their certification. For instance, Di Vito et al. [Di Vito90] note (p. 307) “If over 380 000 lines of text were printed at 50 lines per page, we would have over 7600 pages of proof documentation,” and advocate the need for decomposition: “It is essential that the proof effort be decomposed and modularized to avoid confronting too many details at once.”

Decomposition of security requirements can result in changes in types and topics, as well as other attributes, such as their levels of sensitivity.

Some methods are fairly generic, applying to a variety of NFRs (security, performance, accuracy, etc.). Other methods are specific to the security type or security techniques.

Security Sub-Type Methods

The **SecurityViaSubType3** method is a type decomposition method, based on [TCSEC85], that introduces three subtypes: Confidentiality, Integrity, and Availability. This method decomposes a security softgoal into three offspring softgoals:

$$\begin{aligned} & \text{Confidentiality[Info]} \text{ AND } \text{Integrity[Info]} \text{ AND } \text{Availability[Info]} \\ & \text{SATISFICE Security[Info]} \end{aligned}$$

This is a specialization of the *SubType* method, which can be used to deal with the various definitions of security as described in the previous section. Let’s consider some other type decomposition methods:

- **SecurityViaSubType5** method:

To establish the security of information items, establish each of the component subtypes of security: Availability, Authenticity (or genuineness), Integrity (wholeness), Utility (fitting the use), and Confidentiality (or secrecy), as in [Parker91].

- **InternalExternal** method:

To establish information security of the organization, establish **InternalSecurity** — security of information residing inside the system — and **ExternalSecurity** — security of information external to the system. This is an adaptation of the *automated security policy* and *organizational security policy* in [Sterne91].

Security Topic Decomposition Method

As a topic decomposition method, the **Subclass** method, which is applicable to a variety of softgoals (including accuracy and performance), is also applicable to security softgoals. For instance, since there are two specializations of **Account** in the account management system, the confidentiality softgoal **Confidentiality[Account]** can be refined using the **Subclass** method:

$$\begin{aligned} \text{Confidentiality[RegularAccount]} & \text{ AND Confidentiality[GoldAccount]} \\ \text{SATISFICE Confidentiality[Account]} \end{aligned}$$

This method is an adaptation of the inheritance policy in [Fernandez89].

Security Specializations of Generic Methods

There are several other topic decomposition methods that are applicable to a variety of NFRs (including security, performance and accuracy):

- **Subset** method:

To establish the security of a set of information items, establish the security of each subset of information items.

- **AttributeSelection** method:

To establish the security of an information item to be obtained by a sequence of attribute selections (e.g., `Joe.project.budget`), establish the security of each information item obtained in the sequence (e.g., `Joe.project`, then `(Joe.project).budget`).

- **Class** method:

To establish the security of a class of information items, C , establish the security of its defining aspects and its extension:

$$\begin{aligned} \text{Security[containingClasses}(C)\text{]} & \text{ AND} \\ \text{Security[superclasses}(C)\text{]} & \text{ AND} \\ \text{Security[attributes}(C)\text{]} & \text{ AND} \\ \text{Security[extension}(C)\text{]} \\ \text{SATISFICE Security[}(C)\text{]} \end{aligned}$$

- **DerivedInfo** method:

This was introduced in the previous chapter. The security of a set of information items is established in terms of security of both the function, which

can be used to derive these items, and each of this function's source parameters. For instance, the security of account balance information, which can be derived from debit and credit information, depends on the security of the derivation function and each of this function's parameters.

- **IndividualAttribute** method:

To establish the security of the attributes of a class, establish the security of each attribute of the class. This method draws on *vertical splitting* in [J. Smith83], which is also used for addressing performance requirements.

Now we consider some methods that draw on security techniques or adapt generic methods to security:

- **IndividualAttributeSecurityLevel** method:

To establish the security of an attribute of a class, establish the security of each security level associated with that attribute. This method is similar, in spirit, to the *horizontal splitting* in [J. Smith83].

- **Inheritance** method:

To establish the security of the attributes of a class, establish the security of the attributes of its immediate specializations (this is an adaptation of the first policy in [Fernandez89]; this is contrasted with the **Subclass** method which may lead to other aspects of security, for the specializations, such as definitions, object existence, etc.).

- **ClassAttribute** method:

To establish the security of a class, establish the security of its inherited and specialized attributes (this is an adaptation of a policy in [Fernandez89]).

- **AccessGroup** method:

To establish information security, establish information security in accordance with the access groups.

- **Compartmentalization** method:

For information security for an organization which has complex structural divisions, introduce compartmentalization, departmentalization or segmentation, by introducing new topics.

For example, if the accounting department is divided into foreign and domestic work units, this method can be used to separate foreign security concerns from the domestic ones:

Security[Account, DomesticAccounting] AND

Security[Account, ForeignAccounting]

SATISFICE Security[Account, AccountingDept]

Security Prioritization Methods

As described in Chapter 3, the *priority* of a softgoal can be changed by modifying an attribute of the softgoal, such as its *criticality*. Suppose that a high level of internal confidentiality is demanded for gold accounts, as these are large in number and usually have a high spending limit. This is recorded by setting the priority attribute to **critical**:

InternalConfidentiality[GoldAccount]{critical}.

A variety of priority levels can be specified. This can be mapped to a hierarchical design, which facilitates simultaneously attaining several important requirements [Neumann86],

- **Categorization (or SecurityClass) method:**

To meet the target level of quality, categorize a given requirement as *mandatory*, *discretionary*, or *minimal*. Mandatory security requires a very high-level of care. Discretionary security requires access to information be controlled by users. Minimal security categorizes a security softgoal as non-sensitive.

At the beginning of a softgoal interdependency graph expansion process, the notion of security tends to be quite coarse-grained and yet it is often unclear what notion of security to adopt. The developer can cope with this type of situation by repeatedly applying decomposition methods. At times, clarification of the notion of security can come later during the design process. Other NFRs can also be considered. Developers can choose to wait until they have gained more understanding by applying some operationalization or argumentation methods. Developers can select and apply appropriate decomposition methods. The developer can control the manner in which security softgoals are decomposed, focussing on particular types and topics, and give different treatment to different kinds of information items.

7.3 OPERATIONALIZATION METHODS

Operationalization methods refine a security softgoal into a set of security operationalizing softgoals, thereby committing the design that is being generated to particular ways of satisfying security softgoals. For instance, satisfying the internal confidentiality softgoal for accounts demands, among other things, access authorizations. Such a demand can be met by applying the **Authorization** method, with **Account** as the topic:

AccessAuthorization[Account] HELPS InternalConfidentiality[Account]

Access authorization not only has many variations but is only one of many available security operationalization methods. Figure 7.2 is a catalogue of security operationalization methods that includes methods from the literature that have been used in practice to enforce various types of security policies.

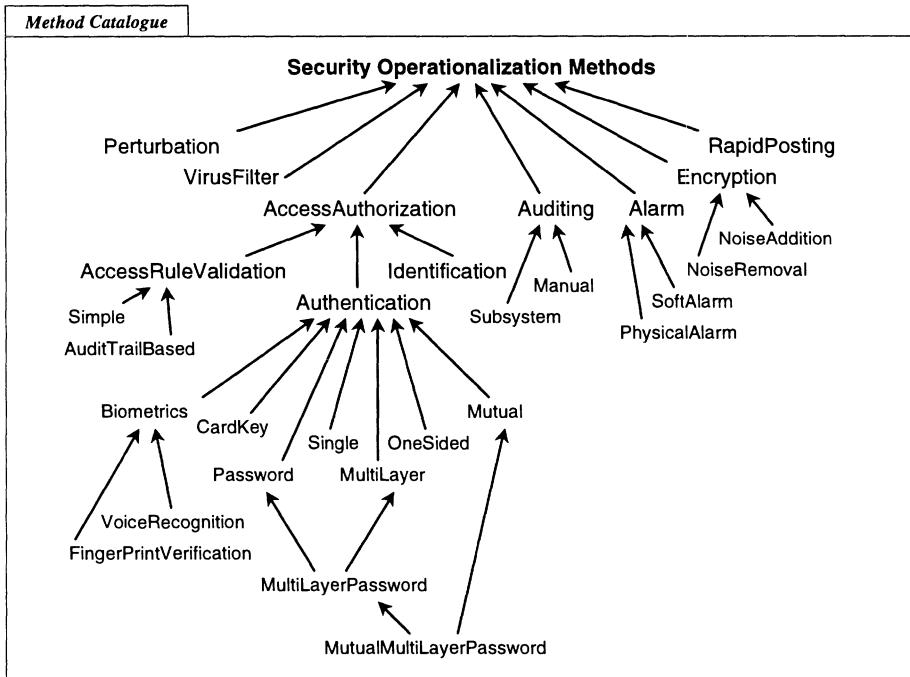


Figure 7.2. A catalogue of security operationalization methods.

Some of these methods are explained below:

- **Identification:**
To tell the system who uses it, identify the user's title.
- **Authentication:**
To ensure that users are in fact whom they claim to be, test their identity. This method can be classified and specialized along many dimensions, including:
 - *the type of protocol used.* This includes personal knowledge (such as a Password method), a physical device (such as CardKey), a physical trait (such as a biometrics device for FingerPrintVerification and VoiceRecognition).
 - *the number of authentications.* This includes SingleAuthentication, which requires performing single login-time authentication, and MultiLayerAuthentication, which requires multiple passwords or procedures. An example is multiple keys for bank safes (where each key is kept by a different employee) to protect highly sensitive information.

- *the parties involved.* This includes **OneSidedAuthentication**, where the (trusted) system authenticates the user, and **MutualAuthentication**, where the agent and the system mutually ensure the identity of each other (Recall the discussion of **MutualID** in the previous chapter).

An authentication method can be a combination of several methods. For example, the **MultiLayerPassword** (or **MultiplePassword**) is a specialization of the **Password** and **MultiLayerAuthentication**, hence exhibiting characteristics of both. Similarly, the **MutualMultiLayerPassword** is a specialization of **Password**, **MultiLayerAuthentication** and **MutualAuthentication**, hence exhibiting characteristics of all three.

■ **AccessRuleValidation:**

To allow access to information items, validate that the user or program has the needed access right. Specializations of this method include:

- simple access rule validation based on rules which state who can access what information under what conditions; and
- access rule validation based on an audit trail, which is input to the access authorization decision [Karger88], which may be needed for the enforcement of *separation of duties* [Clark87] (For instance, different tasks may be required to be done by different employees).

■ **Perturbation:**

To protect against inference from a statistical database, perturb its data [Matloff86]. Specializations of this method include **NoiseAddition** — perturbing data by adding random noise — and **ValueRemoval** — perturbing data by removing extreme values (e.g., for a census database).

■ **Alarm:**

To prevent potentially malicious access to certain vital information, notify authorities of such accesses, either failures or successes. This method may be specialized into **PhysicalAlarm**, notification with an alarm device, and **SoftAlarm**, on-line notification of authorities by the system.

■ **SecurityAuditing:**

To enable authorized personnel to monitor security breaches, selectively maintain an audit trail on significant events. Specializations of this method include **AuditingSubsystem** [Picciotto87] which is used for a compartmentalized systems with workstations.

■ **LimitingAccessTime:**

To reduce the potential for theft, limit access time.

■ **Checksum:**

To protect against changes to the data or classification labels, a checksum may be applied using various granularities of data. A specialization of this method is **CryptographicChecksum** [Denning84], where a cipher checksum is used.

7.4 ARGUMENTATION TEMPLATES AND METHODS

Argumentation templates and methods are used to generate arguments which can be used to support or deny security softgoals.

Claims help readers understand reasons about how rules of access rights were arrived at. For instance, for a complex organization, the size of access rules could become an issue: the bigger the size, the harder to establish their consistency and allow change later on.

Claims also help readers understand how a particular level of security, or confidentiality, was selected. For instance, the decision to adopt mandatory security could involve the consideration of the perceived risks of the system as related to its data and processing environment.

Some security requirements can be met by any number of different design techniques [Woodie83]. Here, claims help readers understand how refinement methods were chosen, especially when cost and resource limitations could become an issue.

For instance,

Claim [“Gold accounts whose balances exceed \$5 000 are few,
but a high level of confidentiality is desired”]

is an argument which can support the treatment of protecting large gold account spending as a very critical confidentiality softgoal.

Claims can be introduced and used to justify particular ways of treating softgoals or interdependency links. For instance, highly sensitive information, such as large gold account spending, may be separated from less sensitive information and treated as such by the use of the VitalFewTrivialMany argumentation method, which was presented in Chapter 4:

Claim [“Gold accounts whose balances exceed \$5 000 are few,
but a high level of confidentiality is desired”]

MAKES

(!Confidentiality[LargeGoldAccounts]{critical}
MAKES Confidentiality[LargeGoldAccounts])

Here the claim justifies the prioritization of the confidentiality softgoal.

7.5 CORRELATIONS

Security requirements can interact with each other, and with other NFRs. Correlation rules catalogue knowledge of conflict or harmony among various NFRs. They are then used to detect and consider tradeoffs among security operationalization methods.

For an example of conflict, consider the method of perturbing the data in a statistical database, *NoiseAddition*. This method contributes to enhancing confidentiality softgoals. However, such perturbation could induce severe bias

— underestimate or overestimate of the true value — into responses to user queries [Matloff86], hence negatively influencing accuracy softgoals. To warn the developer of any undesirable consequences, the knowledge about the generic interaction between accuracy requirements and the perturbation method can be expressed by a correlation rule:

<code>NoiseAddition[<i>Info'</i>]</code>	<code>HURTS</code>	<code>Accuracy[<i>Info</i>]</code>
<code>WHEN <i>Info'</i> isA <i>Info</i></code>		

Here, *Info'* isA *Info* holds if the topic of the noise addition softgoal (an information item) is a specialization of the accuracy softgoal.

There are other examples of correlations arising from the security methods described earlier. `LimitingAccess Time` could help both (value and property) accuracy and confidentiality, but hurt timely accuracy. `Encryption` helps confidentiality but could again hurt (value and property) accuracy, as it reduces the chance of direct examination. On the other hand, `Verification`, which was described in the previous chapter, is a commonly-occurring accuracy operationalization method, which could hurt confidentiality if the verifier is not allowed to access the verification information.

Recall from earlier chapters that *situation descriptors* can be used to form *conditions* for a correlation rule. A *topic relation condition* is a particular kind of condition of a correlation rule. For instance, the following topic relation condition relates *i*, the topic of a parent softgoal, to *i'*, the topic of an offspring softgoal, when one is a subclass, superclass, subset, superset or instance of the other:

$$\begin{aligned} & i' \text{ isA } i \quad \text{or } i \text{ isA } i' \text{ or} \\ & i' \text{ subsetOf } i \quad \text{or } i \text{ subsetOf } i' \text{ or} \\ & i' \text{ instanceOf } i \quad \text{or } i \text{ instanceOf } i' \end{aligned}$$

As a specialization of `AccessRuleValidation` (ARV), the `AuditTrailBasedARV` method requires built-in procedures for authorizing an access with the use of an audit trail as input to the access authorization decision. Such procedures in general require frequent access to disk. The larger the size of an audit trail, the more severe penalty such a softgoal would impose on the performance of processes doing a great deal of file system manipulations [Karger88]. In fact, [Picciotto87] reports that in building an effective auditing subsystem on workstations, disk-intensive processes were slowed down by a factor of 2, while processor-intensive activities were not affected.

7.6 ILLUSTRATION

We now show how a developer could deal with security requirements for a sample system, here an account management system. The developer's decisions and design rationale are recorded in a softgoal interdependency graph.

Functional Requirements and Security Requirements

The system handles expenses which are initially paid by employees, typically by credit card. Reimbursement cheques are issued to employees.

The overall security requirement is to maintain accounts securely. This includes information inside and outside the system. In the domain, priority is given to maintaining security to gold accounts, especially those with large balances.

Initial Softgoal and Subtype Refinements

The developer starts with an initial softgoal, **Security[Account]** (top of Figure 7.3).

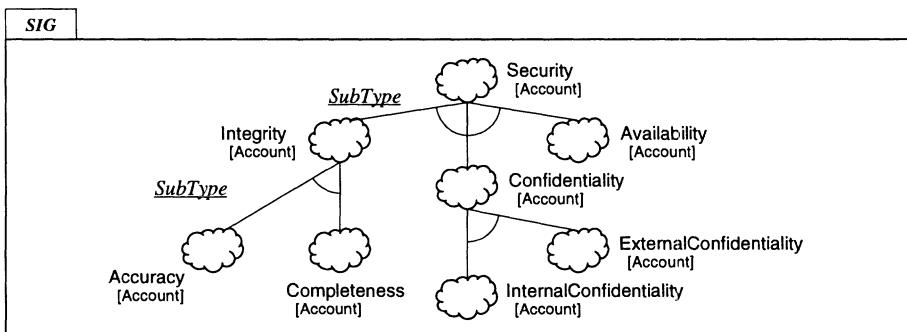


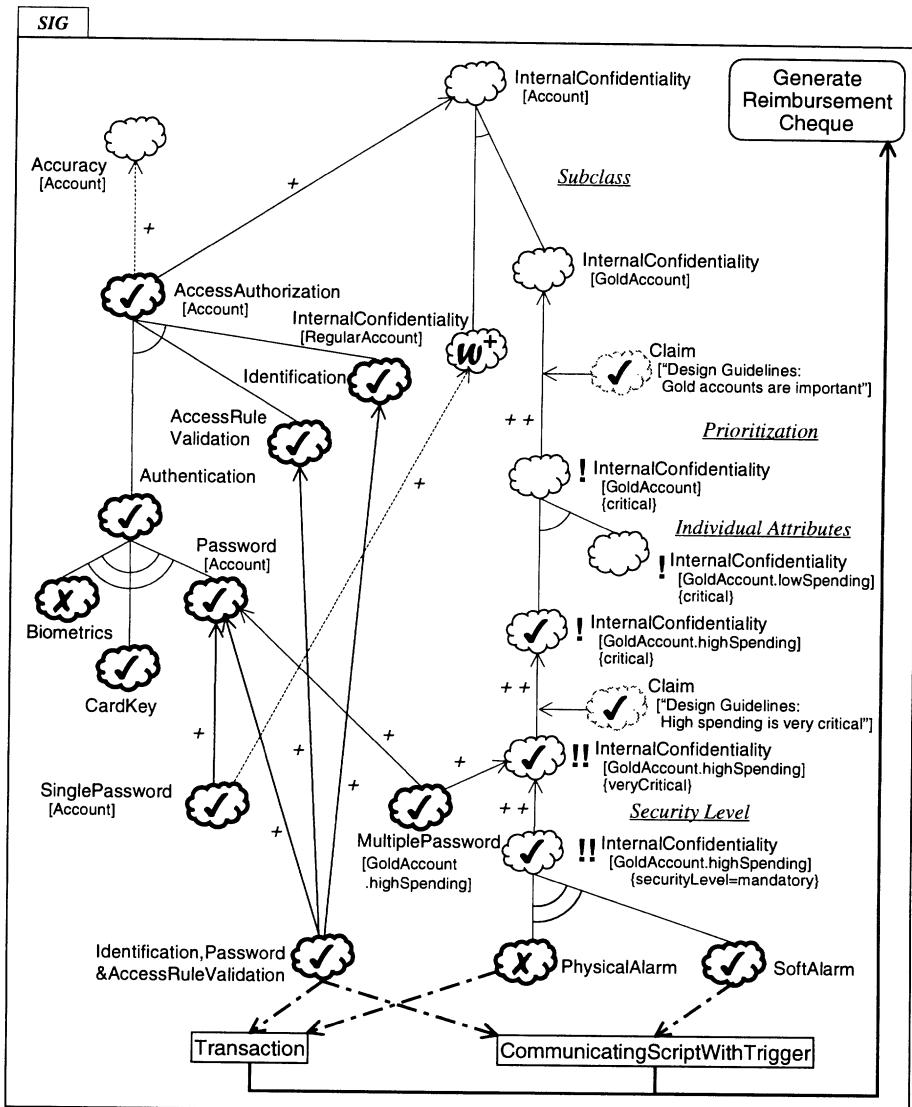
Figure 7.3. Refinement of a security softgoal by subtypes.

The developer repeatedly applies type decomposition methods (Figure 7.3). The developer first applies the **SecurityViaSubType3** method, generating three offspring, namely integrity, confidentiality, and availability softgoals. Then the developer refines the confidentiality and integrity softgoals, again using **SubType** methods.

Confidentiality needs to be enforced for both information that will reside inside the projected system, and information that will be in external media, such as paper or microfilm. In order to address these needs, the developer applies the **InternalExternal** method and generates offspring **InternalConfidentiality[Account]** and **ExternalConfidentiality[Account]**.

Dealing with Internal Confidentiality

Now the developer focusses on internal confidentiality of accounts (shown at the top of Figure 7.4, a continuation from the bottom of Figure 7.3).

**Figure 7.4.** SIG for confidential accounts.

Then the developer considers internal confidentiality of both gold accounts and regular accounts.

Prioritization and Security Levels

In the functional requirements, the security of gold accounts is very important, since they have high spending limits. Thus the developer focusses on `InternalConfidentiality[GoldAccount]` (right hand side of Figure 7.4). This softgoal is prioritized as critical, and this decision is justified by referring to the design guidelines.

There are two kinds of gold accounts: those with high spending usage, and those with low usage. Each account has attributes to identify these two groups, so the developer uses the `IndividualAttributes` method to produce two softgoals.

The developer then identifies the gold accounts with high spending usage as being *very critical* (shown with “!!” in the figure, and priority attribute `veryCritical`). This further prioritization is justified by referring to design guidelines.

In view of the importance of gold accounts with high spending usage, the developer applies the `SecurityLevel` method. This sets the security level (security class) of the softgoal to `mandatory`, one of the eight classes of enhanced protection in [TCSEC85]. To operationalize mandatory security, an alarm will be needed to notify authorities of accesses to information. It may be a `PhysicalAlarm` or a `SoftAlarm`.

Authorizing Access

We've considered operationalizations for internal confidentiality for an important, but small, group of gold accounts. Let's now consider some operationalizations for internal confidentiality of *all* accounts.

The developer chooses the `AccessAuthorization` operationalization (at the left of Figure 7.4, near the top). This *HELPS* `InternalConfidentiality[Account]` as well as `Accuracy[Account]`.

Using the catalogue of Figure 7.2, the developer can use the specializations of `AccessAuthorization` to refine the `AccessAuthorization[Account]` operationalizing softgoal. It is refined into `Identification[Account]`, `Authentication[Account]` and `AccessRuleValidation[Account]`. Note that softgoal topics are sometimes omitted from figures, typically when the parent and the offspring have the same topic.

To consider specific kinds of authentication the developer examines the specializations of `Authentication` in the catalogue. Accordingly, `Authentication[Account]` is refined into `Biometrics[Account]`, `CardKey[Account]` and `Password[Account]`. Any one of these operationalizations can be used, so the contribution type is *OR*.

Selecting a Target System

To address authentication, the developer chooses a card key system along with passwords. In fact, a combination of two password schemes is chosen. A sin-

gle password helps internal confidentiality for regular accounts. A multi-layer password discipline (`MultiplePassword`) helps internal confidentiality for gold accounts with high spending. Biometrics are not chosen, as some users may object to their usage.

A combination of identification, passwords and access rule validation is selected (shown near the bottom left of Figure 7.4).

This leaves the choice of alarms. `SoftAlarm` is chosen to avoid the noise generated by a `PhysicalAlarm`.

The use of `SoftAlarm` has some implications for the target system. Spending limits will have to be monitored over a long time period, and will be periodically adjusted. Certain situations will “trigger” alarms, adjustments, or communication between the system and the manager. This will be implemented by a long-term process (script) with communication and triggering mechanisms, shown as `CommunicatingScriptWithTrigger` at the bottom right of Figure 7.4.

The relationship of operationalizations to the target system is shown at the bottom of the figure. Along the right side of the figure is a link relating the target system to the source functional requirements, shown at the top right of the figure.

The success of security operationalization methods also relies on the co-operation between the system and agents in the environment. The *users' procedure manual* should indicate policies that the agents in the environment should obey when interacting with the system in order to satisfy the confidentiality-specific methods selected. For example, the nature of the communication between the manager and the system can be specified in the manual.

Evaluating the Impact of Decisions

What is the overall impact of the decisions? The evaluation procedure starts with the “leaves” of the graph, often found at the bottom of figures.

On the right hand side of Figure 7.4, the choice of an alarm leads to the critical NFR softgoal, `!InternalConfidentiality[GoldAccount.highSpending]{critical}` being satisfied.

On the left hand side of the figure, the use of card key and passwords satisfies authentication. The use of authentication, along with access rule validation and identification satisfies `AccessAuthorization[Account]`. This in turn *HELPS* satisfy the accuracy of accounts.

In addition, the use of a single password provides weak positive support `W+` for `InternalConfidentiality[RegularAccount]`.

Until more support is given to satisfy `!InternalConfidentiality[GoldAccount]{critical}` (for example, addressing internal confidentiality of low-spending gold accounts), the developer feels that the top softgoal of `InternalConfidentiality[Account]` has not yet been satisfied, and sets the label to *undetermined*.

Although omitted, all the *interdependency links* in Figure 7.4 are satisfied (i.e., have “ \checkmark ” have as their label), since the links are the results of applying catalogued methods.

7.7 DISCUSSION

Key Points

In this chapter, we have used the NFR Framework to produce a *Security Requirements Framework* to represent and address security requirements during information system development.

Various kinds of available development knowledge specific to dealing with security requirements needs to be captured and organized. The Security Requirements Framework helps meets this need.

Now let’s consider the questions we raised at the beginning of this chapter:

- *How can a wide variety of security methods, and their tradeoffs, be made available to the developer?*

Catalogues of security refinement methods and correlations facilitate the systematic capture and reuse of design knowledge. Decomposition methods can guide disambiguating and choosing appropriate notions of security from a rich, diversified set of security notions. For instance, the developer can focus on the confidentiality aspects of run-time operations, instead of addressing broader issues, such as availability and recovery.

Decomposition methods can also be used to gradually refine an abstract security requirement into one or more concrete ones with different levels of security. For example, classes and attributes can be associated with a sensitivity level (e.g., top secret or secret) [Fernandez89] [G. Smith89].

In moving towards a target design, operationalization methods can guide the selection of specific security techniques. For instance, authentication or access rule validation (e.g., [TCSEC85], [ITSEC89], [ITSEC91] or [CTCPEC92]) can be selected. Such operationalizations also have variations, such as a password authentication mechanism, using personal knowledge, biometrics for fingerprint-verification or voice-recognition, which test personal characteristics. Control or deterrent measures [Parker91] are also available, such as alarms, encryption, and physical-access locks.

Catalogues of correlation rules facilitate the systematic capture and reuse of knowledge of softgoal harmony and conflict. For example, a correlation rule can state that biometric authentication improves system security, but may not be user-friendly. Concerning tradeoffs for biometrics, [Lawton98] states that a biometric system is unlike many other security methods that can be lost, stolen, forged, or forgotten. On the other hand, such a system can be quite complex due to the need for sophisticated databases, search algorithms, etc.

- *How can security requirements be systematically integrated into the design?*

When security requirements are treated as softgoals, consideration of design alternatives, analysis of design tradeoffs, selection among design alternatives and justification of design decisions are all carried out in an attempt to meet the softgoals. Hence, security requirements become an integral part of the design process in which they serve as selection criteria for choosing among myriads of decisions, which then can act as basis for justifying the overall design.

Literature Notes

This chapter is based on [Chung93a,b].

Leakage, damage or loss of information, which is resident either in computer memory or in a medium such as paper, microfilm or communication line, could lead to violations of privacy, financial loss or even loss of human life. For instance, revealing credit ratings, medical history or criminal records could all have serious consequences for individuals, while destruction of computer equipment or networks could jeopardize the entire operation of an enterprise. Examples of information security breaches have been reported widely in the literature [Adam92] [Neumann91] [Perry84] [Parker84a].

Treating security requirements as softgoals to be satisfied can be seen as complementary to a product evaluation approach to security. In a product evaluation approach, evaluation criteria serve as benchmarks for selecting a level of security and meeting it for the target system. For instance, the “Trusted Computer Systems Evaluation Criteria” (also known as *TCSEC* or the more colloquially as the “Orange Book”) [TCSEC85] lays out criteria to categorize a system into one of eight hierarchical classes of enhanced protection (e.g., discretionary and mandatory). The “IT Security Criteria: Criteria for the Evaluation of Trustworthiness of Information Technology Systems” (or the “Green Book”) [ITSEC89] describes eight security functions, addressing a wide variety of commercial and governmental security concerns, in a less formal and less rigidly prescriptive manner. A more recent document, the “Harmonized Criteria” [ITSEC91], emphasizes the assurance aspect of security and methodological aspect for evaluating security functionality. For a further comparison, see [E. Lee92]. Due to their generality, these product evaluation criteria are *not* intended to prescribe or proscribe a specific development methodology for semi-formally managing or guiding the complex development process.

While not specific to information systems, there is a body of work which adapts existing software life cycle models to the development and change of secure systems. For instance, security activities can be integrated into the system life cycle, and reviewed and audited in the software quality assurance process [Tompkins86]. Additionally, guidelines can be offered for dealing with certain security concerns for battle management systems, such as visibility and configuration control [Crocker89], in accordance with military security standards for the software development process [DOD-STD-2167A]. Similarly, the

spiral model [Boehm86] can be adapted to meet military standards, by providing mappings for phases and points of iterations [Benzel89]. The spiral model can also be specialized to address both trust and performance, in the context of Ada development [Marmor-Squires89]. For a decision support system, which addresses concerns for cost-effective systems, a statistical approach can be taken to evaluate and choose the most preferred set of security control activities [Bui87].

[TCSEC85] describes integrity as the property that data meet an *a priori* expectation of quality. Schell [Schell87] relates the notion of integrity with accuracy and states: “The problem of integrity is . . . the problem of guarding the database against invalid updates.” (p. 202). Sterne [Sterne91] also seems to share this view, after pointing out that security policy objectives may be divided into organizational and automated ones. These two notions may be compared, respectively, to our notions of external and internal security. The notion of *integrity* or *accuracy* in the Evaluation Criteria seems to correspond to the combined notion of *authenticity* and *integrity* in [Parker91]. Parker separates out the notion of *authenticity* from the notion of *integrity*, after pointing out that some definitions of (data) *integrity* deal not so much with conformance to facts or reality, but with wholeness and completeness. He cites one definition from [TCSEC85], which describes integrity, among other things, as: “The state that exists when computerized data is the same as that in the source documents and has not been exposed to accidental or malicious alteration or destruction.” Another definition is also cited from [NIST90], which defines integrity as “. . . ensuring that data changes only in highly structured and controlled ways.”

This rich variety of works suggests the need for a comprehensive semi-formal development methodology — a methodology to capture the know-how for security enforcement techniques, and their interactions. The NFR Framework is intended to meet this need.

Chosen target systems should deal with selection of data structures (such as partial orders) and algorithms to better organize the accessors in relation to their permitted database operations and data (see, for instance, [Denning79] and [Thomson88]), enforcement techniques for object-oriented design constructs such as message-passing (see, for instance, [Rabitti88] and [Lunt89]), etc.

This chapter has dealt mainly with operational security. It would be interesting to see how the framework enhances user trust for developmental, or process, security [Amoroso91]. This would involve consideration of security requirements for the development process itself, techniques for enforcing process security, and correlations among the techniques. This in turn would require the examination of the various parties involved in the process, such as developers and domain experts, and how they interact with each other.

8 PERFORMANCE REQUIREMENTS

*Complex applications need performance, performance, performance.*¹

As performance is a vital quality factor for systems, an important challenge during system development is to deal with *performance requirements*. For a credit card system, for example, a developer might consider performance requirements to “achieve good response time for sales authorization” and “achieve good space usage for storing customer information.” Being global in their nature and impact, performance requirements are an important class of *non-functional requirements* (NFRs). To be effective, systems should attain NFRs as well as *functional requirements* (e.g., requiring a credit card system to authorize sales). However, it is generally difficult to deal with performance requirements and other NFRs, as they can conflict and interact with each other and with the many implementation alternatives which have varying features and tradeoffs.

This chapter presents a “Performance Requirements Framework” to help a developer deal with performance requirements for software systems. By giving the developer control of the development process, and the ability to interactively consider the particular needs and characteristics of the system under development, it helps a developer “build performance into” customized solutions. It starts with the NFR Framework to give a developer a structured, systematic,

¹From a presentation by Object Design, Inc., Toronto, 20 October 1990.

goal-oriented process to help meet NFRs. This is done by refining NFRs, considering and selecting among competing implementation alternatives, justifying and recording decisions, and evaluating their impact on meeting NFRs. We then apply the components of the NFR Framework to represent, organize and use performance requirements. This involves the representation of performance concepts in performance types, the use of principles for building performance into systems, and the representation and organization of implementation techniques and associated performance considerations as performance methods. In addition, performance issues are further organized by using a *layered* structuring approach. This reduces the number of issues considered at a time, while interrelating decisions about different issues.

In dealing with performance requirements, there are a number of factors to consider. The Performance Requirements Framework helps a developer organize and use this information, thus making the development process more manageable. The framework represents basic performance concepts, such as time and space, and provide a notation for describing performance requirements. It adapts existing principles for building good performance into systems being developed. It uses a variety system development techniques and addresses their impact upon performance requirements. Adopting a layered approach to address performance issues, the framework offers a layered structure to help organize the process of dealing with performance requirements, and reduce the number of issues considered at a time. For any particular system under development, developers can consider the particular domain, expected workload and other characteristics of the system, and its particular performance requirements. All these kinds of information are considered to produce customized systems which address performance requirements.

We illustrate how the framework can be used to deal with tradeoffs, critical softgoals, and how domain information and workload can be reflected in meeting performance requirements.

We start the presentation in this chapter with material on performance requirements which should be applicable to a wide variety of software systems. To go into more detail, we would then have to make assumptions about the kind of software. The next chapter considers performance requirements for information systems, and their implementation. It draws on experience in implementing specifications written in a particular class of object-based design languages, namely semantic data models.

8.1 PERFORMANCE CONCEPTS

Let's consider some of the performance concepts which are incorporated in the Performance Requirements Framework.

Performance and Performance Requirements

Our starting point for understanding *performance* is the standard set of *system-oriented* concepts (e.g., [Lazowska84]), such as the achievement of low response time, high throughput and low space usage.

We then consider principles for building performance into systems [C. Smith86, 90]. This approach considers the impact of software requirements and priorities upon performance *before* coding begins. For example, we can provide users of a banking system with good *responsiveness* — the elapsed time as observed by an end-user of a transaction — by designing a banking machine to dispense cash, without making the client wait for “gold card” travel bonus points to be calculated. These extra non-priority operations can be done later, after the client has departed.

In practice,² *performance requirements* often focus on response time, and are often stated very briefly. At least for information systems, performance requirements may be developed for particular application systems. Of course, there will be tradeoffs among requirements. For example, making a transaction as fast as possible may decrease flexibility for future changes in design. Although performance requirements may be brief, we will see that their ramifications can be complex.

Thus dealing with performance requirements has some difficulties. First, performance requirements can conflict with each other. When producing a target system, one needs to choose among the many development techniques available, which have tradeoffs. Furthermore, performance requirements are hard to handle because they have a global impact on the target system. To deal with performance, one does not just “add an extra module” to the system. Instead, performance will need to be considered throughout the system, and throughout the development process. Finally, one needs to consider the characteristics of the particular domain and system – such as its workload and priorities – in order to produce systems which meet the performance requirements, which vary from system to system.

The Performance Type

Performance requirements drive selection of implementation alternatives, and are stated in terms of performance concepts, such as response time.

Basic performance concepts are organized in the *Performance Type* (Figure 8.1). Types may be further subdivided into subtypes, representing special cases for each softgoal class. For instance, the **Performance** type has subtypes **TimePerformance** (or simply **Time**) and **SpacePerformance** (or simply **Space**), representing respective time and space performance requirements on a particular system.

²Many thanks to Michael Brodie and Ken Sevcik for their insight on the use of performance requirements in industry.

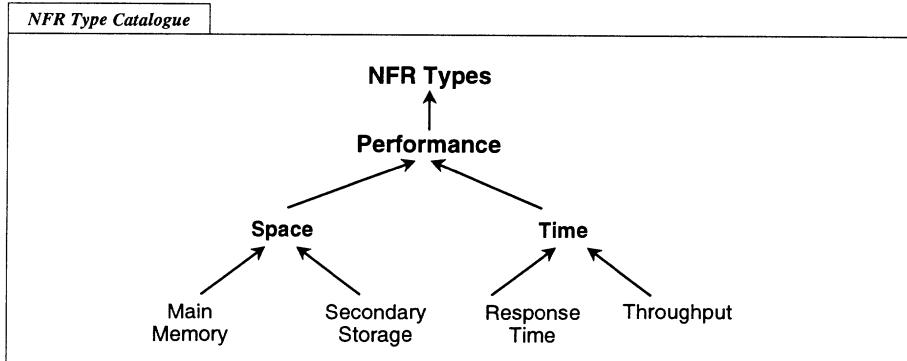


Figure 8.1. The Performance Type.

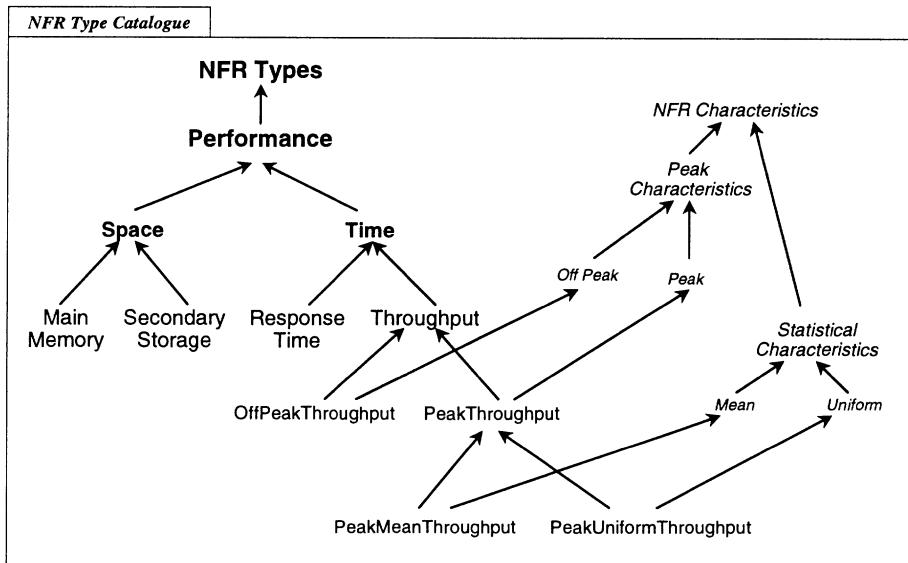


Figure 8.2. Characteristics of the Performance Type.

In turn, time can be refined into response time and throughput, while space can be refined into main memory and secondary storage.

Types are a part of each softgoal. We interpret softgoals for the subtypes of **Performance**, as follows. **Time**[...] is read as good time performance. For **Time** and **ResponseTime**, this means seeking decreased or low time. However, the units for throughput are the reciprocal of response time, so **Throughput**[...] means increased or high throughput. Good **Space** performance means decreased or low usage; this also applies to **SecondaryStorage** and **MainMemory**.

Performance types can be made more specific by considering *characteristics* of NFR types. C. Smith [C. Smith90] (section 3.1.1) and Jain [Jain91] call for performance requirements to be specific and measurable; for example does a requirement apply all day or at peak time? Recommending that performance requirements be specific, measurable, acceptable, reliable, and thorough, Jain provides the acronym “SMART” as an *aide-memoire*.

Concerning specificity of requirements, we distinguish requirements for peak time vs. those for off-peak times, and requirements for good mean value (e.g., low response time) vs. requirements for good uniformity of values (e.g., low standard deviation in response time). See Figure 8.2 for some examples of specifying *characteristics* of performance requirements, such as **PeakThroughput**. One could further refine types, e.g., to consider such measures as 90th and 95th percentiles, e.g., of response times for classes of transactions.

Principles for Building Performance into Systems

We draw on Software Performance Engineering principles [C. Smith86, 90] for designing good performance into software systems. These principles help a developer focus on priorities, and build systems which are responsive to users. The principles include:

1. *Centring Principle*: Efforts to improve performance should focus first on the *critical* and *dominant* parts of the workload. From the dominant workload (i.e., the few routines which are executed the most), trim what is unnecessary to do while the user waits. Assuming that the frequent operations can be sped up, overall average response time can be improved. This principle helps to focus optimization efforts on frequent cases.
2. *Processing vs. Frequency Tradeoff Principle*: Minimize the product of the processing time per execution of an operation, and the frequency of executing the operation. This is applicable to loops and frequently invoked operations. Sometimes there is a savings without a tradeoff, as execution time and/or frequency can be decreased. For example, if one can detect operations which have the effect of retrieving every instance of a class from secondary storage, several entities can be retrieved at the same cost as one entity. It is important to identify when a slight increase in time causes a great decrease in frequency (or vice versa), such as “family plan” situations where handling several requests does not cost much more than handling one request. This principle can be applied by identifying situations where subsequent requests can be handled by slightly extending the scope of an initial request.

3. *Fixing Principle:* For good response time to users, “fixing” (the process of mapping functions and information to instructions and data) is done as early as possible. There is a tradeoff between reducing resource usage (early fixing) and increasing flexibility (late fixing).

Structuring Issues using a Layered Approach

Performance issues can be further structured to help a developer focus on smaller sets of issues at a time, when desired. Layering approaches have often been used to structure performance work.

The framework offers such a layered approach, inspired by a framework for prediction of performance of relational databases [Hyslop91]. Each layer addresses a certain class of performance issues and produces outputs which can be used as inputs at lower layers.

In the framework, design decisions made at higher layers, corresponding to higher levels of abstraction, will be reflected in lower layers which describe the system in more detail. This results in softgoal interdependency graphs at each layer, illustrated later in this chapter.

A developer can choose whether to use a layered approach, and if so, how to structure the layers. Let’s consider a language-based layering.

Consider the decomposition of a source specification into a series of specifications in simpler languages (e.g., which use subsets of successively simpler data model features) until the target system is reached. The framework offers such a language-layering approach: starting with the source (top) language, successive lower layers deal with and eliminate data model features until we arrive at the target language.

We can provide a small sample layering which is applicable to a variety of systems, languages and data models.

1. First we start at the bottom layer with *entities* (or objects, variables, etc.).
2. Then at the second layer we can add *attributes* (or record fields, array elements, etc.).
3. Then at the third (top) layer, we can add *transactions* (or procedures, functions, etc.)

Note that subsets of layers can correspond to existing models. For example, the first two layers can represent the Entity-Relationship model [Chen76], and the three layers can represent a variety of programming languages, as well as the Entity-Relationship model extended with transactions.

That layering can be used to organize *selection* among implementation alternatives, *prediction* of performance of a selected implementation (an initial partial approach is given in [Nixon91]), and development tools. By sharing a common organization, results should be more easily shared among components.

The next chapter extends this simple layering to deal with additional features of languages used for designing information systems. We feel that this layered approach should be applicable to other data models.

Notation for Performance Softgoals

A performance softgoal is written in the form:

Performance Type[Topics, Layer] {Priority Attributes}

The type of a softgoal is written at the beginning of a softgoal, followed by the topics. The final topic is special; it is the layer topic, which is a layer number. Optional priority attributes (dealing with the criticality or dominance of a softgoal) are placed to the right in brace brackets, and exclamation marks are placed to the left of priority softgoals.

Topics include the subjects of a softgoal. They include *information items* such as **Employee** and **Meeting**. They also include operations on information items, whether primitive (predefined), such as `retrieve(Employee)`, or developer-defined, such as `Reimburse(Employee)`. Topics can include nested expressions, such as `!ResponseTime[retrieve(Employee.address), 2]{critical}`. To distinguish them, we capitalize the names of developer-defined operations, but not the names of the predefined ones.

8.2 FACTORS FOR DEALING WITH PERFORMANCE REQUIREMENTS

To give the flavour of the kinds of information used and generated in our approach to dealing with performance requirements, let us consider the factors that a developer might need to consider. During system development, there are a number of input and output factors to consider. We will illustrate them with the case of developing a research administration system.

Basic input factors include:

- *performance requirements* for the system, e.g., expense reimbursement should be fast.
- *other (non-performance) NFRs* for the system (e.g., requiring that expense account information be maintained accurately and securely), which may interact with performance requirements and development decisions.
- *functional requirements* for the system, e.g., the system should maintain records of projects and meetings, and issue reimbursements.
- *priorities* for the system and organization [C. Smith86], e.g., reimbursements should be issued quickly, since it is important to maintain employee morale.
- *workload expectations*. This can be expressed with varying degrees of detail, ranging from the organizational level (e.g., the number of employees, and the number of meetings held per year) to the systems level (e.g., the size of a database, and the number of transactions per minute).

The above basic input factors do not really depend on the particular development approach. Some will be available in early phases of development.

Additional input factors will also have to be considered. Their details may depend on the particular development approach taken, the particular specification languages used, and the development phase at which they are considered. They would include:

- *development techniques* to produce a target system from a source specification, and their associated performance features and tradeoffs. For example, in a database system, one technique would be indexing, which can help improve retrieval time, at the cost of space.
- *interactions and tradeoffs* among NFRs, priorities, development techniques, etc. For example, choosing a particular technique may result in time-space tradeoffs, time-accuracy tradeoffs, etc.
- the *source specification of the system*. For example, this could include definitions of projects, meetings and researchers, and associated operations, such as travel reimbursements.
- *features of the source specification language*. This might include the language's data model features, which would have associated performance features.

Even when different development methods are used (e.g., by using different source or target languages, or considering performance requirements during different phases of development), we would still expect the above kinds of additional inputs to be needed. However, their details might vary.

The results of the development process should include the following *outputs*:

- a *record of development decisions made*, and the *reasons* for decisions. This includes implementation decisions made by the developer, and design rationale given by the developer.
- an indication of *which performance requirements are met*, and to what extent.
- a record of the *interactions and tradeoffs* among NFRs, priorities, workload, decisions made, and alternatives not chosen.
- a *prediction of performance* of the selected system. This would provide a further indication of how well performance requirements will be met by an implementation of a system.

It can be seen that there are a lot of factors to consider during development, and many *interactions* are possible. It is therefore desirable to have a structured process to deal with performance requirements. In the Performance Requirements Framework, the inputs and outputs are recorded or reflected in softgoal interdependency graphs (SIGs).

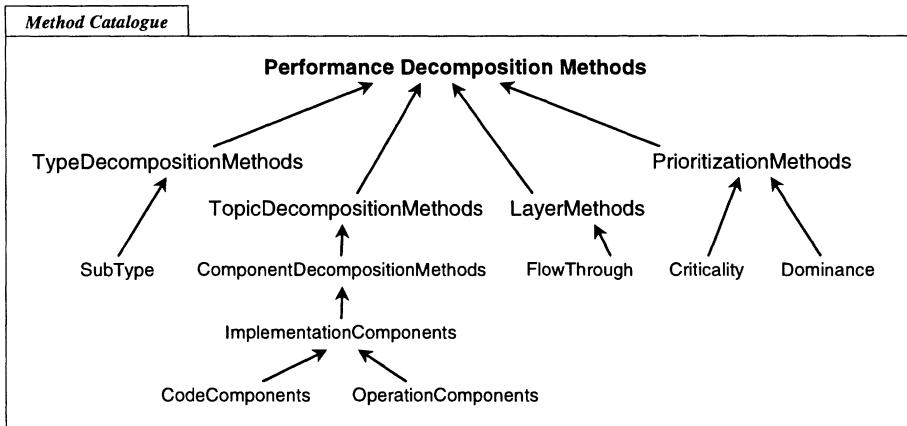


Figure 8.3. Catalogue of Decomposition Methods for Performance.

8.3 REFINEMENT METHODS

We can refine performance softgoals on their components: type, topic, layer topic, and priority attributes. Figure 8.3 shows a catalogue of some performance decomposition methods, arranged by these components.

Now let's present some performance decomposition methods.

SubType Decompositions

Type-based decomposition methods refine a softgoal on the basis of its type. For example, the **TimeSpace** decomposition method takes a softgoal of the **Performance** type, and produces two softgoals, one with the **Time** type, the other with the **Space** type. For example, we can decompose the softgoal of “good performance for the Researcher class at layer 3” into the softgoals of good time performance for Researcher at layer 3 and good space performance for Researcher at layer 3:

$\text{Time}[\text{Researcher}, 3] \text{ AND } \text{Space}[\text{Researcher}, 3]$
 $\text{SATISFICE } \text{Performance}[\text{Researcher}, 3]$

This means that if both offspring softgoals (shown here on the first line) are satisfied, then the parent softgoal (on the second line) will be satisfied.

The above method is a particular application of a general method, and has no parameters. We can define generic methods, using parameters which are shown in sans serif italics. *Info* is an information item, and *Layer* is a layer.

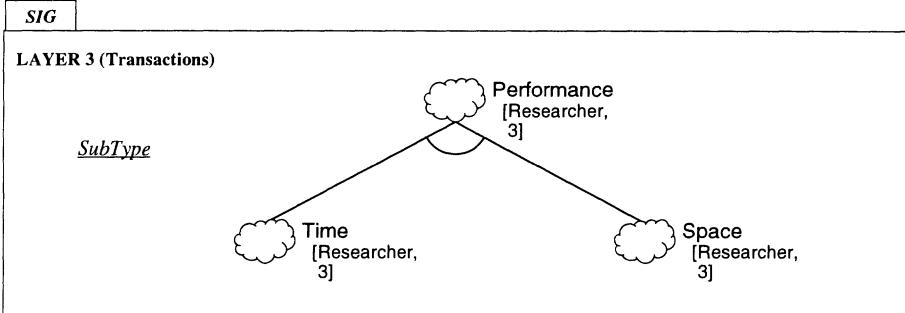


Figure 8.4. A SubType decomposition.

Here is the definition of the **Performance SubTypes** method:

$$\begin{aligned} & \text{Time[Info, Layer] AND Space[Info, Layer]} \\ & \text{SATISFICE Performance[Info, Layer]} \end{aligned}$$

To produce a particular application of a method, we replace the parameters with concrete items, layers and operations. For example, we can replace *Info* above by a variety of subjects: an information item (e.g., *Researcher*), operations on an information item (e.g., *Reimburse(Researcher)*; here the operation name is capitalized), attributes of an information item (e.g., *attributes(Researcher)* as is done in Figure 8.6), etc. In addition, *Operation(Info)* represents an operation on *Info*.

We can further decompose the subtypes of time and space.

- **Time SubTypes** method:

A time softgoal can be decomposed by the **Time SubTypes** method into softgoals for response time and throughput.

$$\begin{aligned} & \text{ResponseTime[Info, Layer] AND Throughput[Info, Layer]} \\ & \text{SATISFICE Time[Info, Layer]} \end{aligned}$$

- **Space SubTypes** method:

A space softgoal can be decomposed by the **Space SubTypes** method into softgoals on main memory and secondary storage.

$$\begin{aligned} & \text{MainMemory[Info, Layer] AND SecondaryStorage[Info, Layer]} \\ & \text{SATISFICE Space[Info, Layer]} \end{aligned}$$

Further type decompositions are possible. They correspond to subtypes shown in the performance type catalogue (Figure 8.2).

Decomposition Softgoal Topics into Components

Let's consider methods which decompose a softgoal based on its topics.

A softgoal for an information item (or an operation on an information item) can be decomposed into softgoals for components of the item, in a number of ways. These *component decomposition methods* refine a softgoal at a layer to one or more softgoals at the same or lower layers.

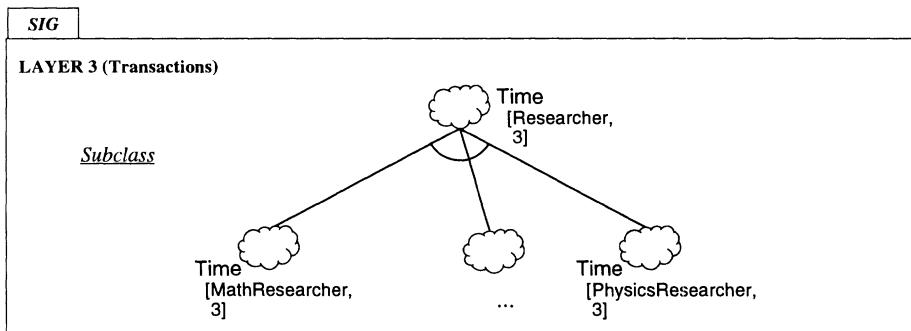


Figure 8.5. Decomposing a performance softgoal using the **Subclass** method.

- **Generic Methods:**

Generic methods discussed in Chapter 4 can be applied to performance softgoals. These include the **Subclass** method (Figure 8.5) and the **Individual Attributes** method (Figure 8.6).

- **ImplementationComponents** method:

By this method, a softgoal for an item is decomposed into softgoals for its implementation components, at a lower (or the same) layer of implementation.

A performance softgoal for an information item is refined into several performance softgoals, each dealing with one of the implementation components:

$$\begin{aligned}
 & \text{Performance}[\text{implementationComponent}_1(\text{Info}), \text{Layer}_j] \text{ AND} \\
 & \dots \text{ AND} \text{Performance}[\text{implementationComponent}_n(\text{Info}), \text{Layer}_j] \\
 & \text{SATISFICE Performance}[\text{Info}, \text{Layer}_k] \\
 & \text{WHERE } \text{Layer}_j \leq \text{Layer}_k
 \end{aligned}$$

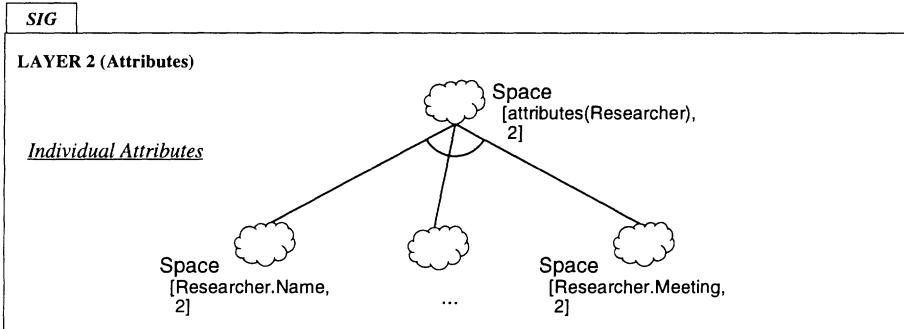


Figure 8.6. Using the IndividualAttributes method.

For example, a time softgoal for accessing an attribute, `Researcher.Meeting`, could be decomposed into time softgoals for two implementation components: finding the offset of `Meeting`, and retrieving the value from storage (Figure 8.7).

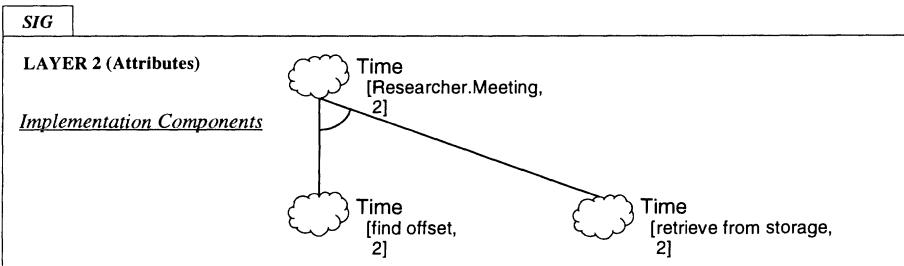


Figure 8.7. A decomposition based on implementation components.

- **OperationComponents** method:

We can also decompose a softgoal topic based on the components of an operation. A softgoal for an operation `Operation(Info)` on an information item `Info`, is decomposed into softgoals for the components of the operation, at the same or lower layers. This may assume that the implementor has some knowledge of the implementation.

`Performance[component1(Operation(Info)), Layerj] AND ... AND`

$\text{Performance}[\text{component}_n(\text{Operation}(\text{Info})), \text{Layer}_j]$
 SATISFICE $\text{Performance}[\text{Info}, \text{Layer}_k]$
 WHERE $\text{Layer}_j \leq \text{Layer}_k$

- **CodeComponents** method:

A softgoal for a body of code $\text{CodeBody}(\text{Info})$ dealing with an information item Info can be decomposed into softgoals for its components (e.g., sub-blocks) [C. Smith86, 90].

$\text{Performance}[\text{codeComponent}_1(\text{CodeBody}(\text{Info})), \text{Layer}_j] \text{ AND}$
 ... AND
 $\text{Performance}[\text{codeComponent}_n(\text{CodeBody}(\text{Info})), \text{Layer}_j]$
 SATISFICE $\text{Performance}[\text{Info}, \text{Layer}_k]$
 WHERE $\text{Layer}_j \leq \text{Layer}_k$

We adopt a convention applicable to all refinements. If the same type (e.g., **Performance**) appears throughout a method, the decomposition may be specialized by uniformly applying it to a subtype (e.g., **Time**). For example, we can uniformly replace **Performance** by **Time** throughout on both sides of the above method, to obtain:

$\text{Time}[\text{codeComponent}_1(\text{CodeBody}(\text{Info})), \text{Layer}_j] \text{ AND} \dots \text{ AND}$
 $\text{Time}[\text{codeComponent}_n(\text{CodeBody}(\text{Info})), \text{Layer}_j]$
 SATISFICE $\text{Time}[\text{Info}, \text{Layer}_k]$
 WHERE $\text{Layer}_j \leq \text{Layer}_k$

Refinements to Handle Multiple Layers of SIGs

Softgoals at a given layer are refined into softgoals at the same, or lower, layers. Thus softgoal interdependency graphs (SIGs) can be formed for each layer. In order to combine such SIGs into a global record, methods are needed to refine downwards through layers.

To relate softgoals which are at different layers, the **FlowThrough** method refines a softgoal at one layer to a softgoal with the same type and topic, but at a lower layer. Layers are connected by *inter-layer interdependency link*. Interestingly, they help developers deal with issues at different layers, and help connect different issues which are considered at different layers. Furthermore, they obey the normal rules for interdependency links: for example, the evaluation rules work in the same way for them.

$\text{Performance}[\text{Info}, \text{Layer}_m] \text{ MAKES } \text{Performance}[\text{Info}, \text{Layer}_n]$
 WHERE $\text{Layer}_m \leq \text{Layer}_n$

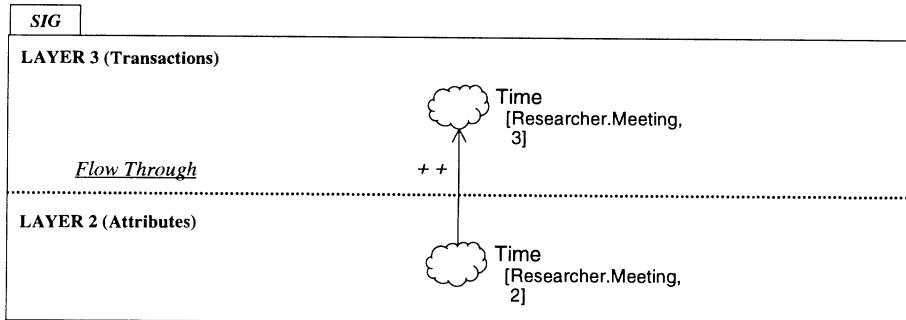


Figure 8.8. Using the FlowThrough method to link layers.

The contribution type for inter-layer interdependency links is *MAKES* when satisficing the lower-layer softgoal satisfies the higher-layer one. In some cases, where there is only a partial contribution, *HELPS* would be appropriate.

For example, Figure 8.8 shows how a time softgoal for *Researcher.Meeting* at Layer 3 (which deals with transactions) can be refined into a softgoal at Layer 2 (which deals with attributes).

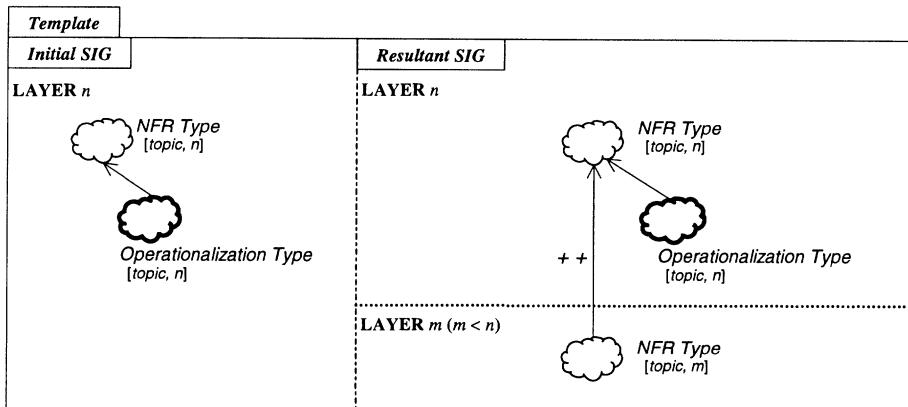


Figure 8.9. A template for inter-layer refinement.

To keep within the rules of the NFR Framework, however, one must be careful in making refinements. Operationalizing softgoals cannot be refined into NFR softgoals. This forbids the refinement of an operationalizing softgoal

(say at the bottom of a SIG at one layer) into an NFR softgoal (say at the top of a layer below). This can be resolved by refining an NFR softgoal into an operationalizing softgoal at the same layer, *and* an NFR softgoal at a lower layer (Figure 8.9). The inter-layer interdependency link between the two NFR softgoals will also allow propagation of evaluation results, upwards through layers. This approach is in keeping with the NFR Framework. For a different approach, allowing operationalizations to be refined into NFR softgoals, see Chapter 14.

Evaluation (labelling) begins at the bottom layer's SIG (See, for example, Figure 8.22). Using the normal labelling procedure, one starts at the bottom of that SIG, and obtains results at the top of the record for that layer; results are then linked to softgoals at a higher layer, starting at (or near) the bottom of that layer. This procedure is repeated until results are obtained at the top of the SIG at the top layer.

Refinements to Handle Prioritization

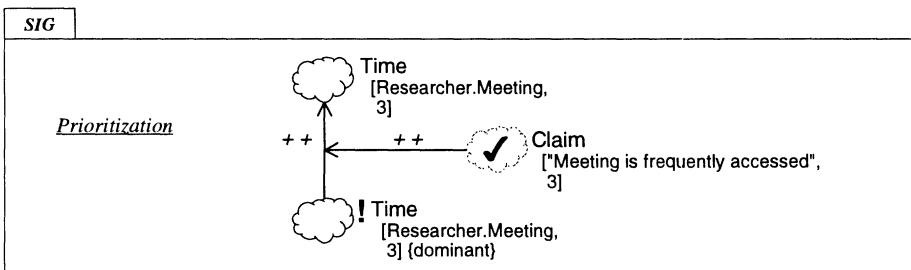


Figure 8.10. A prioritization argument.

C. Smith [C. Smith90] points out the need to address both dominant and critical parts of the workload. Priority softgoals are preceded by an exclamation mark, and are followed by an attribute indicating the kind of priority. For example, the Criticality method takes a softgoal of the form `Performance[Info, Layer]`, and gives the sub-softgoal a critical attribute, which indicates that the satisfying of the softgoal is critical for attaining good performance. This is written as `!Performance[Info, Layer]{critical}`. Note that attributes of softgoals reflect qualities of softgoals. They are not topics, and are shown in brace brackets after the softgoal topics.

The Dominance method identifies a softgoal as referring to a dominant part of the workload, by adding an attribute.

`!Performance[Info, Layer]{dominant} MAKES Performance[Info, Layer]`

For example, Figure 8.10 shows a time softgoal being identified as being dominant. The refinement is supported by a workload argument from the domain. Other methods identify softgoals as being non-dominant, non-critical, etc.

This treatment (based on [Chung93a]) of critical and dominant softgoals helps us realise the *centring principle* [C. Smith90]: we can focus on the *dominant* workload (i.e., the few routines executed the most), and then trim what is unnecessary to do while the user waits. Priority-based decompositions may be supported by arguments drawing on workload statistics.

Dealing with Priorities and Uncertainties

We have presented some refinement methods in one step, simply having a contribution type such as *AND*. In some cases this will not fully represent the situation, and it may be better to represent the situation in two or more steps.

One concern is when only *some* offspring of a decomposition are priorities (critical or dominant) for the satisficing of a parent. For example, a time softgoal for calculating reimbursements to employees may be refined (via an *AND* interdependency) into softgoals for researchers and managers, but only the researchers' reimbursements might be critical. Drawing on C. Smith's [C. Smith86] principles, satisficing the parent is much more influenced by satisficing the priority (critical or dominant) offspring than the non-priority ones. This is not fully captured by a simple *AND* decomposition in which denial of a non-priority offspring would deny the parent. Instead, the prioritization templates shown in Figures 4.30 and 4.31 are frequently used to deal with performance requirements.

Another concern arises when satisficing the offspring softgoals does not necessarily satisfice the parent softgoal. The developer may wish to indicate that the satisficing of the offspring and their parent may only *partially* contribute to the satisficing of the parent's parent. For example, the **ImplementationComponents** method, presented earlier, could be done in two steps. First a performance softgoal for an information item is refined into a performance softgoal for the (set of) implementation components of the information.

```

Performance[implementationComponents(Info), Layerj]
HELPS Performance[Info, Layerk]
WHERE Layerj ≤ Layerk

```

The contribution is *HELPS* because aspects other than implementation components may be needed to satisfice **Performance[Info, Layer_k]**. In the second step, a performance softgoal for the set of implementation components of the information is refined into several performance softgoals, each dealing with one of the implementation components:

```

Performance[implementationComponent1(Info), Layer] AND ... AND
Performance[implementationComponentn(Info), Layer]
SATISFICE Performance[implementationComponents(Info), Layer]

```

In this kind of situation, we can use a combined contribution type, here *AND_HELPs*, when referring to the contribution of the performance of the set of individual implementation components to *Performance[Info, Layer_k]*.

Another concern is when the offspring's topics (e.g., of an *AND* contribution) do not necessarily partition the parent's topic. In other cases, the developer might be uncertain whether a method is applicable to a given situation.

In these various cases, a developer may wish to "soften" a contribution. For example, an *AND* interdependency can be softened by placing an *HELPs* contribution above or below it. This was shown in Figures 4.33 (the *AND_HELPs* contribution) and 4.32 (the *HELPs_AND* contribution).

8.4 OPERATIONALIZATION METHODS FROM SOFTWARE PERFORMANCE ENGINEERING

Performance Operationalization Methods (OMs) make commitments to implementation decisions, while increasing our confidence of satisfying (fulfilling within reasonable bounds) a performance softgoal.

Some operationalization methods apply software performance engineering [C. Smith86, 90] principles for building performance into systems. For example, an *UncompressedFormat* for attributes uses more space than a *CompressedFormat*, but has the benefit of speeding access by avoiding repeated uncompression and compression of data. An *UncompressedFormat* is an example of *EarlyFixing*, the mapping, as early as possible, of functions and information to the instructions and structures which achieve them.

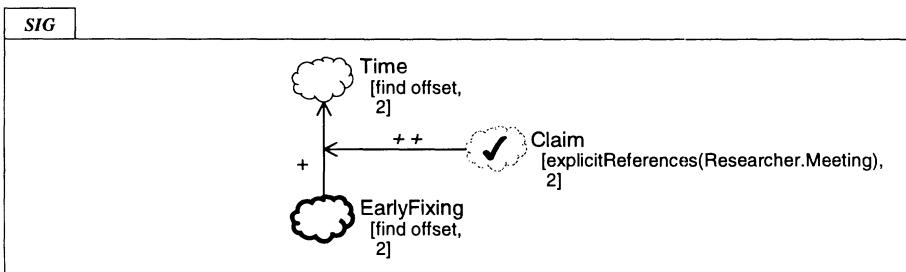


Figure 8.11. Using the *EarlyFixing* method.

Early Fixing Methods

By *EarlyFixing*, early connection (e.g., at compilation time) is made between an action and the instructions that achieve it [C. Smith90]. This method has a

positive impact on time performance. It will also have some impact on space performance, but we cannot in general say whether it will be positive or negative.

$\text{EarlyFixing[Info, Layer]} \text{ HELPS Time[Info, Layer]}$

For example, in Figure 8.11, EarlyFixing helps satisfice a time softgoal for finding the offset of an attribute. This choice is supported by an argument that the name of the attribute is given explicitly in the source code, allowing the offset to be determined statically.

There are several ways to specialize this method.

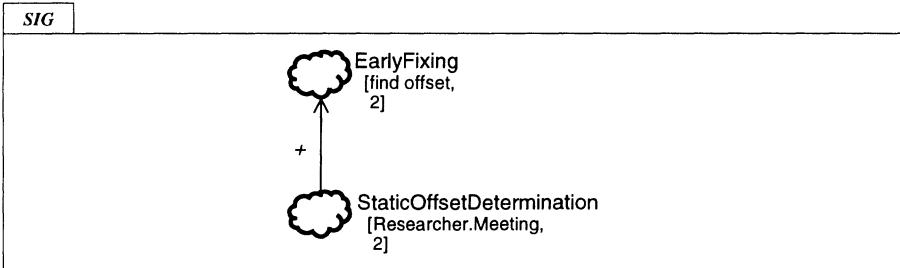


Figure 8.12. Refining an EarlyFixing operationalizing softgoal.

- One way is StaticOffsetDetermination which determines offsets statically, rather than at execution time. This is illustrated in Figure 8.12. Note that an operationalizing softgoal can be specialized to another operationalizing softgoal.

$\text{StaticOffsetDetermination[Info, Layer]} \text{ HELPS EarlyFixing[Info, Layer]}$

Since EarlyFixing *HELPS* Time, we conclude that:

$\text{StaticOffsetDetermination[Info, Layer]} \text{ HELPS Time[Info, Layer]}$

- Indexing helps EarlyFixing, hence it helps Time softgoals. However, it has a negative impact on space softgoals, due to the storage requirements for the index; this impact is represented by the *HURTS* contribution.

$\text{Indexing[Info, Layer]} \text{ HELPS Time[Info, Layer]}$

$\text{Indexing[Info, Layer]} \text{ HURTS Space[Info, Layer]}$

- UncompressedFormat. also helps EarlyFixing. Thus it also helps satisfice Time softgoals. Storing information in an uncompressed format uses extra space,

but avoids extra access time for encoding and decoding.

<code>UncompressedFormat[Info, Layer]</code>	<i>HELPS</i>	<code>Time[Info, Layer]</code>
<code>UncompressedFormat[Info, Layer]</code>	<i>HURTS</i>	<code>Space[Info, Layer]</code>

Late Fixing Methods

`LateFixing`, the opposite of early fixing, has a negative impact upon time performance, but also has undetermined impact on space performance.

<code>LateFixing[Info, Layer]</code>	<i>HURTS</i>	<code>Time[Info, Layer]</code>
--------------------------------------	--------------	--------------------------------

Late fixing hurts time performance. As we will see shortly, late fixing can have a positive impact on space softgoals, and some time softgoals. It is helpful to consider these kinds of tradeoffs. These impacts, positive and negative, are also recorded in correlations, which are catalogued in Section 8.6.

- One specialization of `LateFixing` determines offsets dynamically, rather than at execution time. Since `DynamicOffsetDetermination` *HELPS* `LateFixing`, and `LateFixing` *HURTS* `Time`, we conclude that:

<code>DynamicOffsetDetermination[Info, Layer]</code>	<i>HURTS</i>	<code>Time[Info, Layer]</code>
--	--------------	--------------------------------

- `CompressedFormat` also helps `LateFixing`. Storing information in a compressed format saves space, but the encoding and decoding (i.e., late fixing) necessitates extra access time.

<code>CompressedFormat[Info, Layer]</code>	<i>HELPS</i>	<code>Space[Info, Layer]</code>
<code>CompressedFormat[Info, Layer]</code>	<i>HURTS</i>	<code>Time[Info, Layer]</code>

- The `ReduceRunTimeReorganization` method can be viewed as a variant of late fixing. When dealing with frequent schema changes, `ReduceRunTimeReorganization` uses a structure which uses late fixing to reduce expensive run-time reorganization to offer less variation (i.e., greater uniformity) in response time, but at the cost of higher average response time.

<code>ReduceRunTimeReorganization[Info, Layer]</code>		
<i>HURTS</i>	<code>MeanTime[Info, Layer]</code>	

<code>ReduceRunTimeReorganization[Info, Layer]</code>		
<i>HELPS</i>	<code>UniformResponseTime[Info, Layer]</code>	

Interestingly, this method is negative for one type of time softgoal, but positive for another. This illustrates the need, pointed out in [C. Smith90] and [Jain91], to be careful when specifying requirements.

Figure 8.13 shows some of the impact (positive and negative) of this method on two time softgoals. With this method, during run-time we can use more look-ups of schema information, rather than compiling it in; this makes it easier to change the schema while allowing accesses.

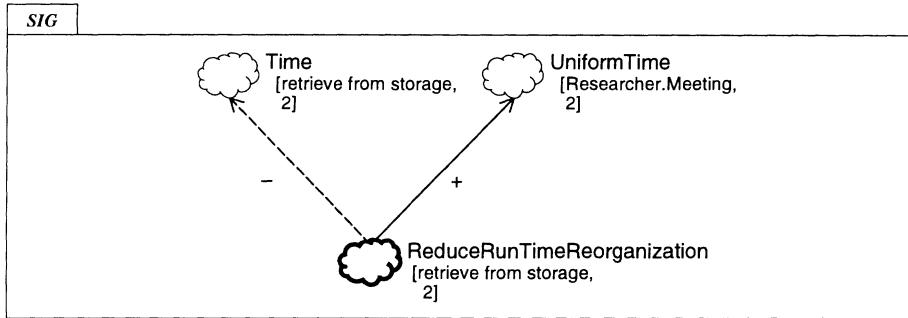


Figure 8.13. Positive and negative impacts of an operationalizing softgoal.

Execution Ordering Methods

To satisfy a time performance softgoal, a developer may selectively state the relative order for the execution of a set of tasks. *Responsiveness* [C. Smith86, 90], the elapsed time as observed by an end-user of a transaction, can be shorter than the total time for a transaction to complete. For example, after registering a researcher, control may be returned to the user even though some statistics on total registrations are still being updated. The elapsed time can be improved by early execution of critical and dominant tasks. Consider the **PerformFirst** method:

PerformFirst[*Operation(Info)*, *Layer*] *HELPS* Time[*Operation(Info)*, *Layer*]

Here operation *Operation(Info)* is performed before other operations on the information item. This helps time performance. Related methods and their contributions are:

PerformEarly	<i>HELPS</i>	Time
PerformLater	<i>HURTS</i>	Time
PerformLast	<i>BREAKS</i>	Time

Here, topics and layers of softgoals are omitted for brevity.

Figure 8.14 presents a catalogue of some operationalization methods for performance requirements. The methods use Software Performance Engineering (SPE) techniques [C. Smith86, 90].

8.5 ARGUMENTATION METHODS AND TEMPLATES

Argumentation methods and templates can provide evidence for a choice of methods. Suppose we know that all references to information item *Info* in a

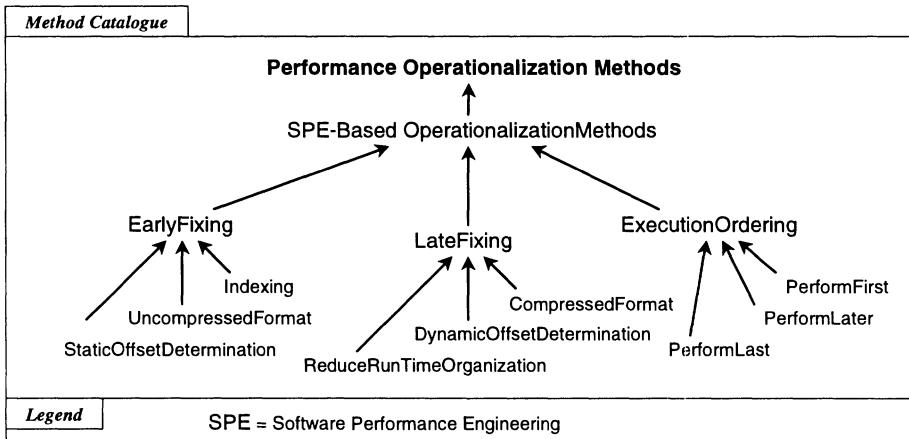


Figure 8.14. Catalogue of Operationalization Methods for Performance.

segment of code are specific, and can be uniquely determined statically, rather than being variable expressions with several possible values. This argument, written

Claim[`explicitReferences(Info, Layer)`],

can be used as evidence that static addressing techniques will suffice a softgoal of fast retrieval.

An argument that an information item is subject to frequent changes in the schema can be written: `Claim[frequentSchemaChanges(Info, Layer)]`.

An argument that access on an information item is frequent can be written: `Claim[frequentAccess(Info, Layer)]`.

As in earlier chapters, *situation descriptors* can be used in claims. Examples are `explicitReferences()`, `frequentSchemaChanges()` and `frequentAccess()`. Situation descriptors can also be used in other softgoals and in correlation definitions. To distinguish them from softgoals, they start with a lower-case letter, and use round parentheses for parameter lists.

Claims can also consist of a textual explanation, e.g., `Claim["Meeting and Name are frequently accessed together", 3]`.

Expected or actual workload statistics, as well as information about a schema, can also be used as arguments. If performance prediction results are available, they too can be used in arguments. Such arguments may be used to support the identification of a softgoal as involving a critical or dominant part of the workload.

<i>Correlation Catalogue</i>			
Contribution of offspring <i>Operationalizing Softgoal</i>	to parent <i>NFR Softgoal</i>		
	Time [Info]	Uniform Response Time [Info]	Space [Info]
<i>Early Fixing Operationalizations:</i>			
EarlyFixing [Info]	HELPS		
Indexing [Info]	HELPS		HURTS
Uncompressed-Format [Info]	HELPS		HURTS
<i>Late Fixing Operationalizations:</i>			
LateFixing [Info]	HURTS		
Compressed-Format [Info]	HURTS		HELPS
Reduce-Run Time-Reorganization [Info]	HURTS	HELPS	

Figure 8.15. A correlation catalogue for Performance.

8.6 CORRELATIONS

Relationships between softgoals can be stated by *correlation rules*, which are collected in *correlation catalogues* (Figure 8.15). This catalogue shows the impact of offspring operationalizing softgoals upon parent NFR softgoals.

For example, using a compressed format to store information is good for space performance but bad for response time, due to the need to uncompress or compress the information. A correlation rule can state that a compressed format operationalizing softgoal *HURTS* a time softgoal.

Early and late fixing have a variety of contributions, positive and negative. Methods may be selected for some of these contributions, and the correlation catalogue will detect others. For example, *LateFixing* might be chosen to help space requirements, and the correlation catalogue will detect a negative impact on time.

In addition to such *general* correlations, *application specific* conditions can be attached to correlation rules. For example, a developer could use domain knowledge to record that the use of a list structure is good for response time *if* a condition holds, such as requiring a list to be suitably small.

Some operationalizing softgoals are effectively mutually exclusive, at least when applied to the same information item. For example, the negative relationship between two kinds of storage format can be written:

```
CompressedFormat[addr, 2] BREAKS UncompressedFormat[addr, 2]
```

On the other hand, mutual exclusion can also create a positive relationship, for example, between operationalizing softgoals which impose a relative order on the execution of two distinct operations, *Operation₁* and *Operation₂*.

```
PerformLater[Operation1, Layerj]
HELPS PerformFirst[Operation2, Layerk]
WHEN distinctOperations(Operation1, Operation2)
```

Here the WHEN keyword introduces a condition needed for the contribution to apply.

8.7 ILLUSTRATION

Let's illustrate how the framework can be applied to the example of the research management system.

Domain and Workload Information

The system is used to administer research projects. Researchers are associated with institutes, and attend meetings. The developer has definitions of the main concepts. For example, Researcher has several attributes including the researcher's name, and the current meeting attended.

The developer also has some information on the workload of the system. This would include the number of researchers, the frequency of meetings, and the typical attendance at meetings. This gives an indication of the frequency of operations. The developer may also have a set of priorities for the system. Here, reimbursing researchers for their travel expenses for meetings is frequent, and is given priority.

We will now develop a SIG. The SIG will deal with two layers, starting at Layer 3. It will show some refinement methods, and the impact of higher-layer softgoals upon lower ones.

Stating an Initial Requirement

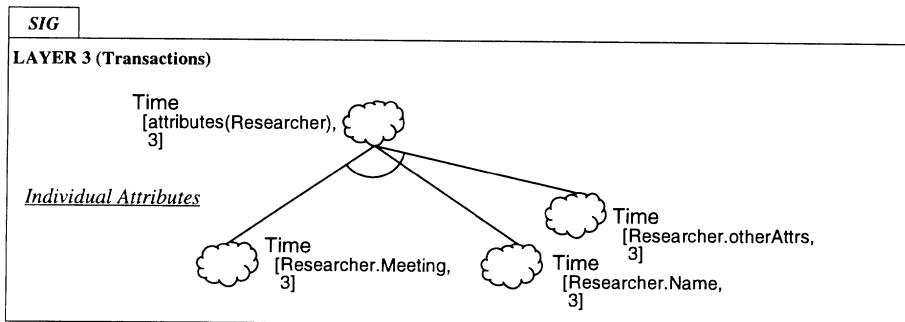


Figure 8.16. A decomposition on attributes.

The developer starts by stating a requirement of good time performance for transactions dealing with the attributes of researchers. This can be represented by the softgoal `Time[attributes(Researcher), 3]` at the top of Figure 8.16.

The developer can choose the layer at which an initial softgoal is stated. One approach is to start with a softgoal addressing the highest layer applicable. Here the developer focusses on *transactions* (operations) dealing with *attributes*. Layer 3 addresses transactions, while Layer 2 handles attributes. The developer decides to state the softgoal at Layer 3. Other approaches (e.g., starting at Layer 2) might be considered.

A Refinement Based on the Topic of a Softgoal

The developer focusses on the time softgoal, and wants to decompose the topic (subject) of the softgoal. This refinement takes the time softgoal for operations on the attributes of `Researcher`, and produces several time softgoals, one for each particular attribute (`Name`, `Meeting`, etc.). This use of the `IndividualAttributes` method is shown in Figure 8.16.

Prioritization Based on Workload

The developer considers the workload and priorities. The developer recalls that travel reimbursement is a frequent and priority operation, and notes that reimbursement for meetings will involve both the researcher's name and the

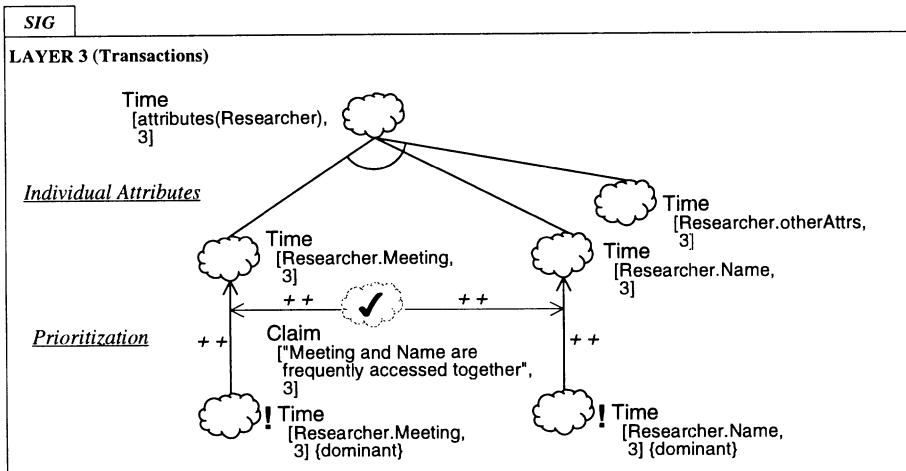


Figure 8.17. A prioritization argument based on workload.

meeting the researcher attended. Thus access to these attributes is recorded as being a dominant part of the workload, and these prioritizations are supported by a claim based on the workload, shown in Figure 8.17.

Choosing an Operationalizing Softgoal

With this domain, workload and prioritization information in mind, the developer considers operationalization methods. One option is to retrieve both attributes, **Name** and **Meeting**, at the same time. By appropriately structuring the data and operations, this may be able to be accomplished in little more time than it takes to retrieve just one attribute. This selection of the **Family-PlanRetrieval** method using the Processing vs. Frequency Tradeoff Principle is shown in Figure 8.18.

The use of **FamilyPlanRetrieval** *HELPS* both of the priority time softgoals. We will discuss evaluation of the graph a little later.

The use of “family plan” results in a target system using grouped retrieval of attributes of **Researcher**. The bottom and the right side of the figure relate the target system to the functional requirements of maintaining the attributes of **Researcher**.

An Inter-Layer Refinement

Having considered operations at Layer 3, the developer now wants to focus on attributes, which are at Layer 2. Note that the operationalizing softgoal

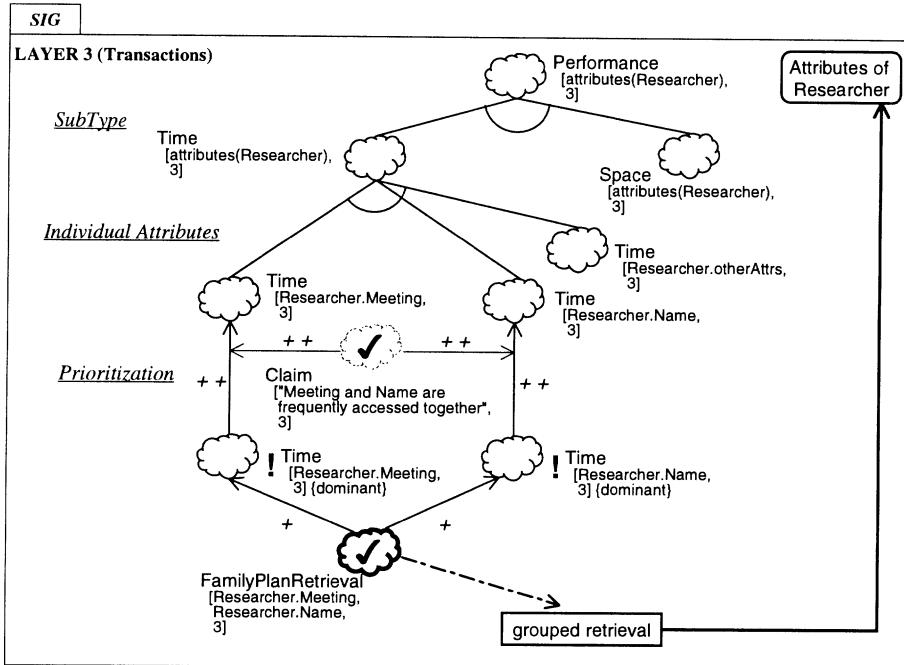


Figure 8.18. Consideration of an operationalizing softgoal.

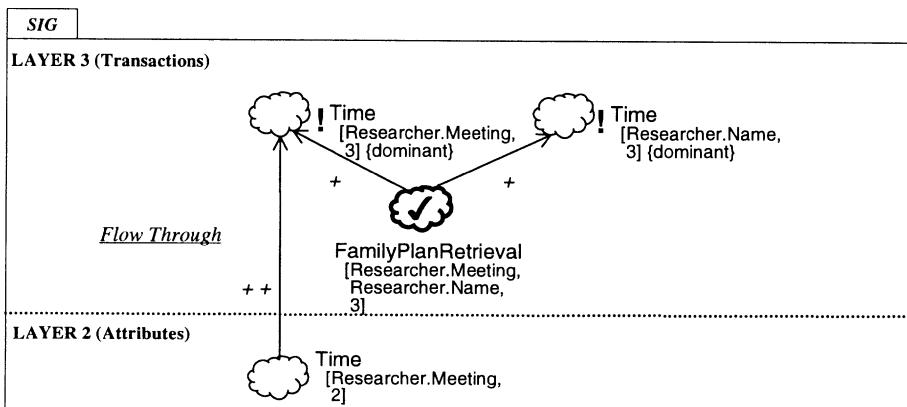


Figure 8.19. An inter-layer interdependency link.

at Layer 3 cannot be refined down into NFR softgoals at lower (or the same) layers. Instead, the Layer 3 dominant time softgoal for **Meeting** is refined to a corresponding softgoal at Layer 2, using the **FlowThrough** method (Figure 8.19).

The **FlowThrough** and subsequent decompositions for **Researcher.Name** are similar to those for **Researcher.Meeting**, but are not shown in the figures.

Considering a Characteristic of Time

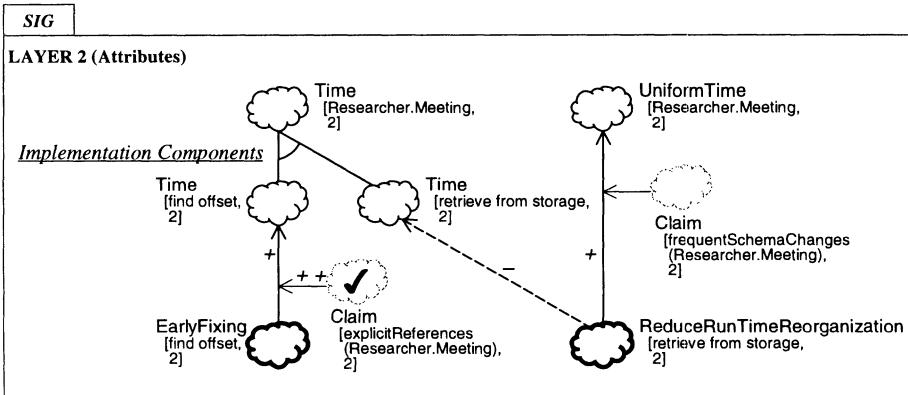


Figure 8.20. Initial operationalizing softgoals.

Now we are at Layer 2. Figure 8.20 shows the portion of the softgoal interdependency graph at that layer. The top softgoals of this layer include a softgoal resulting from an inter-layer refinement, as well as a softgoal initially stated at this layer, but not connected to a higher layer.

We now consider a *characteristic* of the **Time** type (Section 8.1). Figure 8.20 has a **UniformTime** softgoal, which represents a requirement that the time performance be uniform, i.e., with low deviation. There is also a top softgoal for **Time**; here we'll treat this as dealing with *average* time performance. As we will see, these two kinds of softgoals can lead to different outcomes.

If desired, the developer could have instead refined the top **Time** softgoal of this layer using the **SubType** method to decompose it on a statistical characteristic:

```
MeanTime[Researcher.Meeting, 2] AND
UniformTime[Researcher.Meeting, 2]
SATISFICE Time[Researcher.Meeting, 2]
```

In that case, the **MeanTime** softgoal on the left of the figure might be prioritized, and would be refined using the **ImplementationComponents** method. In addition,

the **UniformTime** softgoal on the right side would have an impact on higher layers.

For the softgoal **Time[Researcher.Meeting, 2]**, we consider how operations on the **Meeting** attribute may be implemented. Thus we use the **ImplementationComponents** method (Recall Figure 8.7). Assuming that the attribute is stored as a field within a record, the time softgoal for the attribute is refined into a softgoal for finding the offset of the attribute, and another for retrieving the attribute value from storage.

Two different operationalizations are considered. Each is suitable for different kinds of requirements.

One operationalization is **EarlyFixing** of finding the offset of the attribute. This involves determining the offset as early as possible, for example at compilation time. This helps time performance at run-time by not having to determine the offset during each access. The developer argues that this is the case, as the code explicitly refers to the **Meeting** attribute by name, hence the attribute's offset can be determined statically. This will help provide good *average* time performance.

Another operationalization tries to reduce reorganization of the data at run-time. For example, **LateFixing** can be used, by having more lookups of data locations at run-time. This would avoid having occasional major delays in accessing data while the data is being reorganized. Hence **ReduceRunTimeReorganization** helps meet the softgoal of *uniformity* of time performance. On the other hand, the lack of reorganization may lead to poorer *average* access time. The negative impact of this operationalization on average time for retrieval from storage is shown as a correlation link.

It is interesting to note that consideration of a softgoal for *average* time performance leads to one type of operationalization, while consideration of *uniform* time performance leads to quite another.

If schema changes were frequent, the developer might use this domain information as an argument for trying to reduce run-time reorganization, as this could avoid great variations in response time during schema changes. Here, however, such changes are infrequent, so the claim is not shown as being satisfied.

The developer now moves towards an implementation (Figure 8.21) by refining the operationalizations into specific development techniques. Here the developer considers the method of determining the offset of the **Meeting** attribute.

One form of **EarlyFixing** is **StaticOffsetDetermination**, which helps meet the **EarlyFixing** softgoal. This is in fact chosen, and is indicated by a “ \checkmark ” in the figure. This helps meet the (average) time performance softgoal.

DynamicOffsetDetermination, a form of **LateFixing** would help satisfy **ReduceRunTimeReorganization**, and help meet the uniform time performance softgoal. However, this kind of offset determination is not selected.

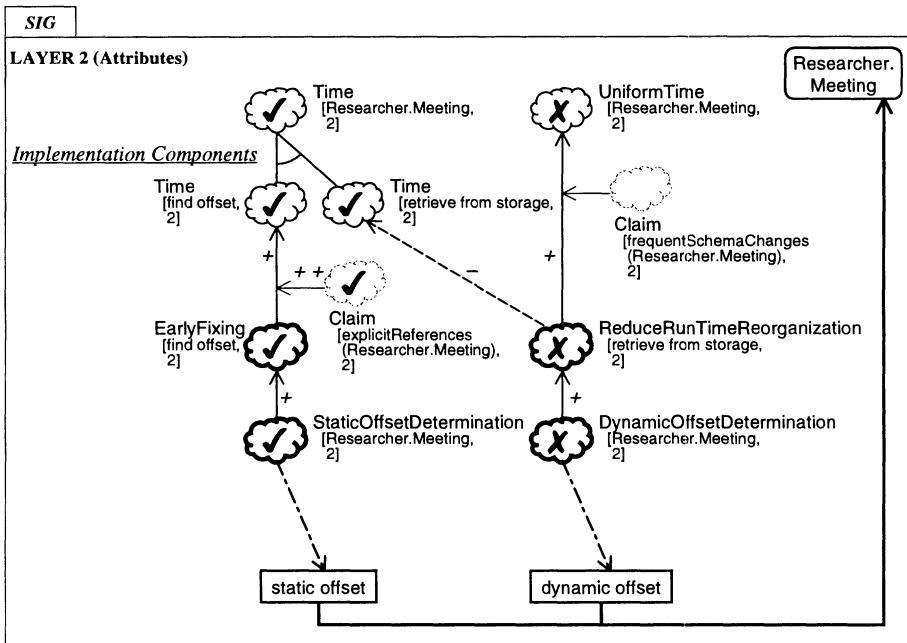


Figure 8.21. Refined operationalizing softgoals.

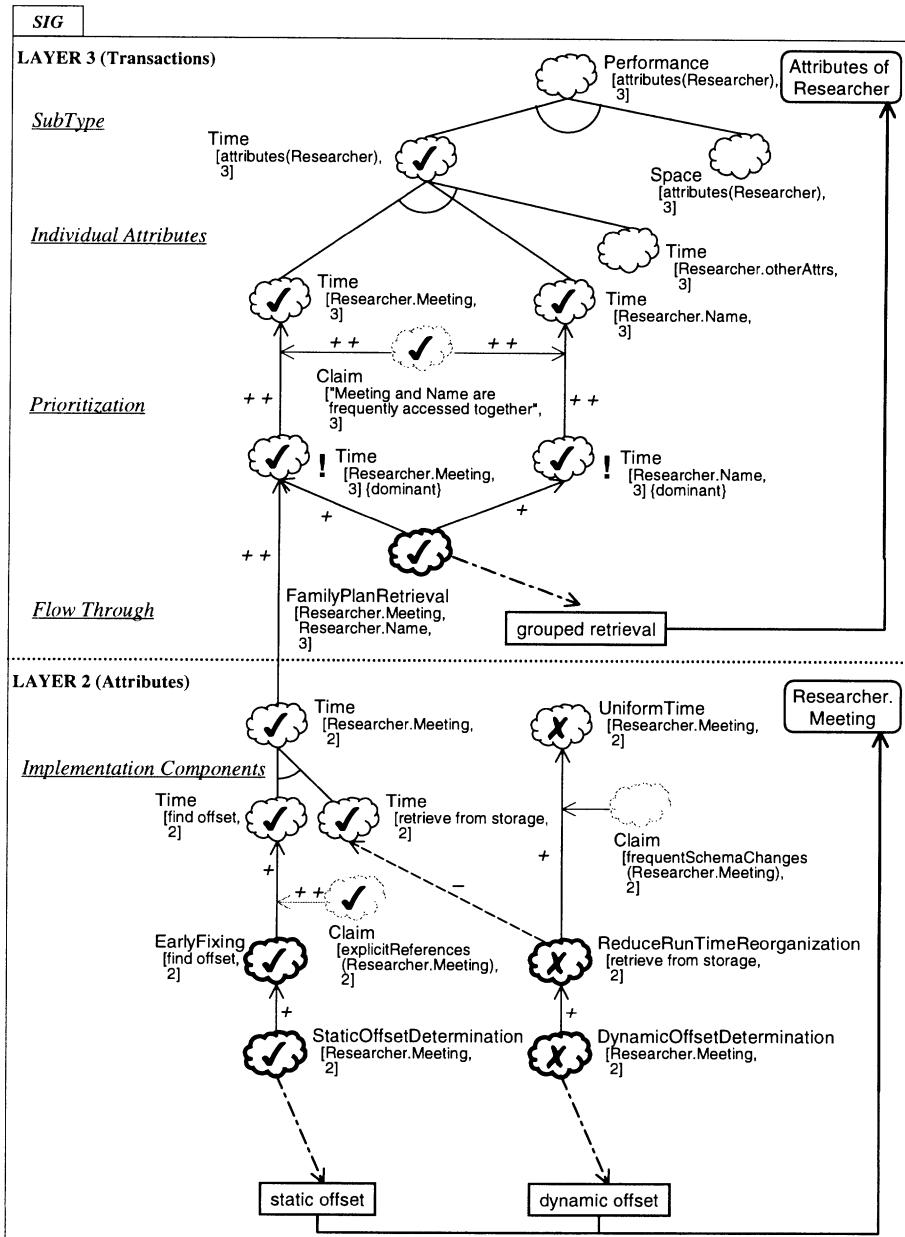
The bottom and right side of the figure relate the possible target systems to the functional requirements of dealing with the **Meeting** attribute of **Researcher**.

Evaluating Different Layers of a Graph

We have refined the softgoal interdependency graph, starting at Layer 3, and then moving down to Layer 2.

Now in Figure 8.22, evaluation works bottom-up, starting at the bottom of the lower layer. We start with the selected and rejected softgoals, including operationalizing softgoals and claim softgoals (These were shown in Figure 8.21 as “ \checkmark ” and “ \times ”).

Choosing **StaticOffsetDetermination** *HELPS* its parent, **EarlyFixing**. Using the evaluation procedure, **EarlyFixing** receives a weak positive label. The developer changes the label to satisfied (“ \checkmark ”). Similarly, the satisficing of **EarlyFixing**, taking into account the accepted claim about references to the **Meeting** attribute, leads to **Time[find offset, 2]** being satisfied.

**Figure 8.22.** Evaluation of the Softgoal Interdependency Graph.

Rejecting `DynamicOffsetDetermination` leads to a weak negative label (W^-) for `ReduceRunTimeReorganization`, which the developer changes to denied (“ \times ”). This leads to denial of the `UniformTime` softgoal, and, via a correlation link, a weak positive contribution to `Time[retrieve from storage, 2]`, which the developer changes to satisfied.

Then a top softgoal of this layer, `Time[Researcher.Meeting, 2]` is satisfied, since its offspring are satisfied. As we mentioned, the other top softgoal, for `UniformTime` is denied.

Now the evaluation moves up a layer, following the inter-layer link from `Time[Researcher.Meeting, 2]` to `!Time[Researcher.Meeting, 3]{dominant}`, using the standard rules for evaluation.

At Layer 3, the `FamilyPlanRetrieval` operationalization helps both dominant time softgoals. In addition, `!Time[Researcher.Meeting, 3]{dominant}` is satisfied by the inter-layer contribution from Layer 2 as well. This leads to the parent being satisfied. The developer also considers `!Time[Researcher.Meeting, 3]{dominant}` and its parent to be satisfied. As the time softgoals have been satisfied for the two priority attributes, `Meeting` and `Name`, the time softgoal for all attributes of researcher is considered satisfied as well.

8.8 DISCUSSION

This chapter has introduced a “Performance Requirements Framework,” which extends the NFR Framework by addressing performance concepts and techniques.

In addition to the generic methods available in the NFR Framework for all NFRs, we have added methods based on principles for building performance into systems. Positive and negative relationships among softgoals are catalogued as methods and correlations.

Layered approaches are often used in performance work. This is adapted to the NFR Framework, organizing softgoal interdependency graphs into layers dealing with particular issues. We illustrated how a softgoal interdependency graph can be developed with more than one layer, and the results evaluated for all layers.

The next chapter continues to look at performance requirements, but focusses on a particular kind of target system, namely information systems.

Literature Notes

This chapter and the next are based on B. Nixon’s thesis [Nixon97a]. They also draw on other publications [Nixon90, 91] [Mylopoulos92a] [Nixon93, 94a, 97b, 98].

Performance requirements were addressed in this chapter, and then performance requirements for information systems are addressed in the next.

Our layered approach was inspired by Hyslop’s [Hyslop91] layered decomposition of performance prediction for relational database management systems.

Each of Hyslop's layers has its own performance model which takes certain input values and produces outputs which are considered at lower layers.

Work on metrics has produced classifications of performance requirements. Jain [Jain91] defines a number of *metrics*: criteria, used to compare the performance of systems, that relate to the speed, accuracy and availability of services. These include response time, throughput, utilization, reliability and availability.

Further literature notes on performance requirements are given at the end of the next chapter.

9 PERFORMANCE REQUIREMENTS FOR INFORMATION SYSTEMS

The previous chapter addressed performance requirements. In order to be more specific about performance requirements, we need to make assumptions about the kind of system under development. This chapter focusses on performance requirements for *information systems*. It continues the presentation of the “Performance Requirements Framework” started in the last chapter.

Performance is a vital quality factor for systems, and this is no less true for *information systems*. Information systems are a cornerstone for the successful working of many organizations. Unlike other types of systems (e.g., operating system software), information systems have to handle incoming data and transactions, provide an interface with external users, and maintain persistent storage. Information systems exhibit a variety of characteristics, ranging from a large to small number of entities, and from large to small volume of transactions, which vary in duration and complexity. Without good performance, such systems would be practically unusable, leading to lack of service, lack of information, clients who are dissatisfied, and corporate loss. Hence it is important to consider performance requirements during information system development.

Recall from Chapter 8 that performance requirements for information systems often focus on response time. They are often stated very briefly, and may be developed for particular application systems.¹

This chapter presents a framework for dealing with performance requirements for information systems. It extends the presentation of the Performance Requirements Framework, which was introduced in Chapter 8. It organizes and catalogues knowledge about information systems, features of their specification languages, their implementation techniques and their performance features. A catalogue of methods is given, which draws on results from databases, principles for building performance into systems, and experience in implementing object-based systems, particularly semantic data models. The layered organization of issues, introduced in Chapter 8, is extended to deal with language features of semantic data models. The framework for dealing with performance requirements for information systems is illustrated using the research administration example.

9.1 LANGUAGE FEATURES AND IMPLEMENTATION TECHNIQUES FOR INFORMATION SYSTEMS

In addressing performance requirements for information systems, we consider the kinds of features found in languages that are used to specify the design on a system. We also consider the kinds of target techniques used to produce an implementation of an information system that is consistent with its design. We then examine the performance characteristics of these target techniques to determine how well they meet a variety of performance requirements.

Our focus, then, is on the body of design language *features* and target *implementation techniques*, rather than on particular languages, *per se*. The emphasis is on selection of data structures and algorithms; hence detailed code is not often shown in this chapter. We feel that the approach should be applicable to a variety of approaches to information system development, since several of the features and techniques are used in a variety of languages and implementations.

To be more specific, we consider performance requirements during the development phase when high-level conceptual source specifications of information systems are translated to target implementations. By considering performance requirements during this phase, we can draw upon implementation experience and techniques.

More specifically, we consider translating a semantic data model (particularly the Taxis language) to a relational database implementation language augmented with application programmes (such as is offered by the DBPL language).

¹Thanks to Michael Brodie for his help concerning the nature of performance requirements for information systems in industry.

In order to naturally and directly model the subject matter of the application domain, semantic data models adopt an entity-oriented framework. These conceptual design specification languages are based on an Entity-Relationship data model [Chen76], and are used to specify and manage a large and complex information base. Entities are grouped into classes, which have associated attributes. Classes are arranged in inheritance (*IsA*) hierarchies: specialized classes inherit attributes from general classes. Taxis [Mylopoulos80] [Borgida90a] [Borgida93] applies inheritance uniformly to all types of classes, including entities, short-term transactions, long-term processes and integrity constraints. TaxisDL [Borgida93] is a refinement of Taxis, offering non-procedural specification facilities.

As a target database implementation language, we focus on DBPL [Matthes92a, 93] [Schmidt88] [Borgida89], a relational database programming language which offers structures and operations for database applications, modularization, and a rich, static type system. Given the abstraction in the language, the DBPL programmer need not consider too many database implementation issues.

Recall from Section 8.2 that there are input factors which apply when dealing with performance requirements. Some factors generally apply to a variety of systems. When we focus on information systems, and choose the source and target languages, we can be more specific about the nature of the *additional input factors*. For example, the *language definition* includes data model features of the source language, here Taxis. *Development techniques* include implementation techniques and tradeoffs for information systems. They also include techniques to deal with features of the source language, e.g., methods for representing inheritance *IsA* hierarchies in the target system. The *source specification* includes definitions of source concepts such as researchers, meetings and travel authorization; here, they would be specified in Taxis, e.g., as entities and transactions.

Even if the particular source or target language were changed to other languages with similar features, we would expect that the basic principles and issues described in this chapter would still apply. In part, this is because Taxis shares features with object-oriented systems including entities (objects) with identity, inheritance and persistence. Taxis has also influenced the design of data models for object-oriented databases [Fishman87].

Some Lessons from Implementation Experience

Experience with implementation of semantic data models gives some motivation for using an approach such as the NFR Framework.

There is a variety of implementation techniques for features of semantic data models. In developing an implementation, performance requirements can be considered, along with a number of other NFRs, such as reliability and safety. These NFRs can interact with each other and with the implementation techniques. We observed [Nixon90] that when applying “off-the-shelf” technology from areas such as compilers and databases to the implementation of seman-

tic data models, *integration problems often arise due to interactions among data model features, implementation techniques and NFRs.* Such interactions need to be considered, as they often have global impact, and lead to tradeoff dilemmas and sub-optimal implementations.

Accordingly, we do not feel that automatic compilation using generic algorithms and structures (“canned alternatives”) alone suffices for efficient implementation of information systems. Rather, the developer should be allowed to exploit characteristics of the particular application by choosing an implementation in an interactive, developer-directed, way.

Hence, in dealing with NFRs in general, and in particular for dealing with performance requirements for information systems, we need to deal with tradeoffs and conflicts among performance requirements, and with the many implementation alternatives which have varying performance features. Furthermore, we want to be able to deal with interactions which are discovered when development is already underway. In addition, there is a great deal of knowledge to consider. Thus, there is a need to organize issues and structure the development process.

In view of these needs, and our experience, we feel that the NFR Framework, as specialized in the Performance Requirements Framework, allows developers to interactively use their expertise to produce customized solutions by building performance into information systems. It offers a good basis for cataloguing knowledge about performance of information systems, and for dealing with interactions.

9.2 EXAMPLE: A RESEARCH MANAGEMENT SYSTEM

As an illustration, we continue to consider a research administration system. We give an outline of the source specification of the system. At the same time, we illustrate some of the data model features of Taxis which would be used in the source specification.

This hypothetical system is intended to model the management of expenses for scientists, drawn from several institutes in several countries, who are engaged in a related set of research projects. Participating scientists register for, and attend, group meetings, then submit reports on meetings and expenses. Reports are reviewed and expenses are reimbursed. Scientists then plan and execute their work. This cycle is repeated periodically.

The developer can start by defining some *classes of entities*. Figure 9.1 shows definitions of the class `Employee`, along with its specialization `Researcher`.

Attributes can be defined for each class. Entities (such as `Jacob`, an instance of class `Employee`) will have a value for the attributes defined on the class. For example, `Jacob` might have a `BirthDate` of `14May1948`.

Classes can be related by *inheritance*. Specialized classes (e.g., `Researcher`) are formed by refining definitions of general classes (e.g., `Employee` in Figure 9.1).

As a consequence, all `Researchers` are also `Employees`. Furthermore, specialized classes inherit attributes from their superclasses. For example, Re-

```

entityClass Employee with
  unchanging
    BirthDate: Date
    DateStartedWork: Date
  changing
    Name: String
    Dept: Department
    Inst: Institute
    Mgr: Manager
end Researcher

entityClass Researcher isA Employee with
  changing
    Meeting: ResearchMeeting
    NumOfPapers: NonNegativeInteger
end Researcher

```

Figure 9.1. Definitions of the Employee and Researcher classes.

searcher inherits all attributes of **Employee**, such as **Name**, are inherited from **Employee**. In addition, new attributes can be defined for **Researcher**. An example is **Meeting**, a meeting attended by a researcher.

Transactions can also be defined. Typically, these are short-term operations on entities. For example, a transaction to register a given researcher for a meeting can be defined. It includes a header indicating the transaction and topic **RegisterForMeeting(Researcher)**, and also include details of the operations to be performed.

A number of *data management facilities* will be automatically pre-defined for the entity classes. These facilities include operations to insert and remove entities, update attribute values, and retrieve values.

Further specializations can be defined. For example, **ComputerResearcher** is a specialization of **Researcher** (Figure 9.2). New attributes can be defined for **ComputerResearcher**. An example is **System**, the operating system used by the researcher. Of the inherited attributes, some (e.g., **DateStartedWork**) retain the initial definition, and some of them can be specialized. For example **Salary** is inherited from **Researcher**. While not shown in the figure, the salary for computer researchers could be restricted to a certain range.

Classes are arranged in *IsA (inheritance) hierarchies*. Specialized classes are placed under general ones. Figure 9.3 shows such an arrangement of entity

```

entityClass ComputerResearcher isA Researcher with
    changing
        ComputerType: Computer
        System: OperatingSystem
end ComputerResearcher

```

Figure 9.2. Defining ComputerResearcher as a specialization of Researcher.

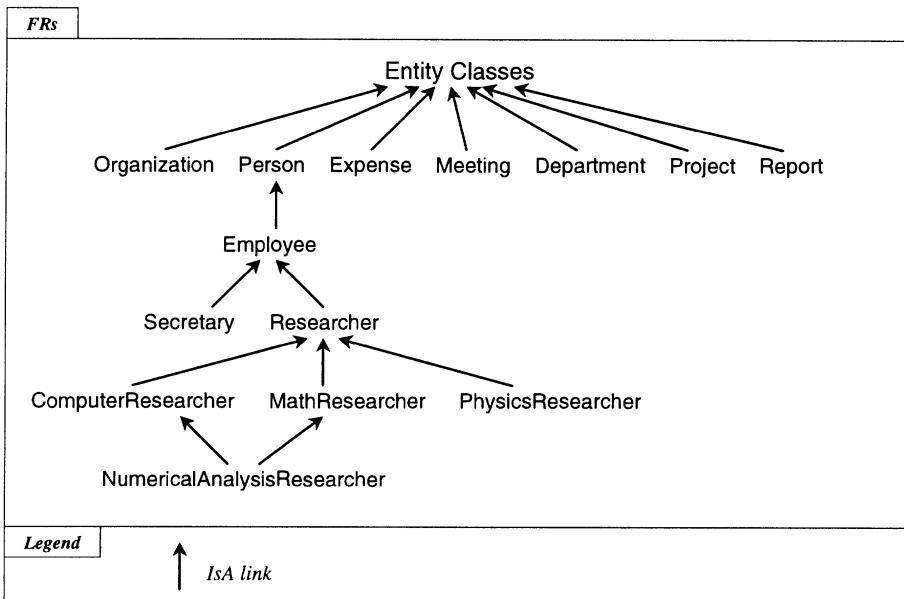


Figure 9.3. Entity (data) classes in the functional requirements for the research administration example.

classes for the example. With *multiple inheritance*, a class may simultaneously be a specialization of two or more classes; e.g., a NumericalAnalysisResearcher is both a ComputerResearcher and a MathResearcher.

All types of classes in Taxis (such as entity classes, transactions, etc.) are arranged in IsA hierarchies. For example, Figure 9.4 shows the main transactions arranged in an IsA hierarchy. For example, the general transaction

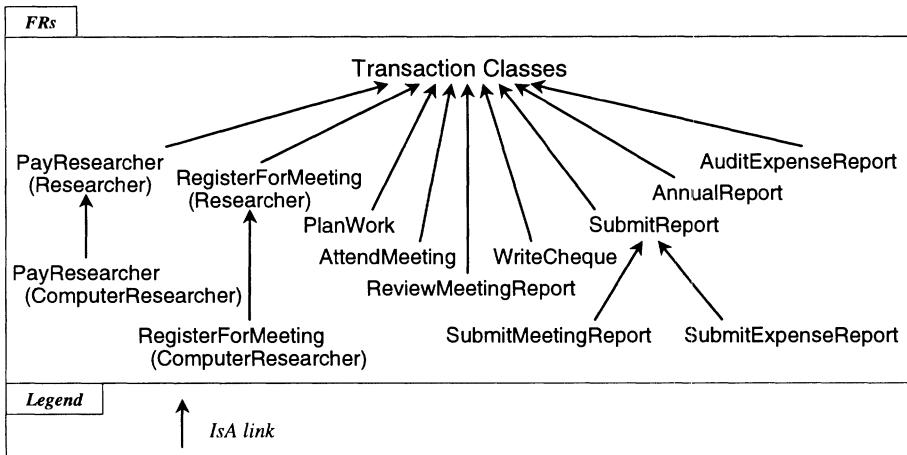


Figure 9.4. Transaction classes in the functional requirements.

`RegisterForMeeting(Researcher)` can be specialized to `RegisterForMeeting(Computer Researcher)`. Here the specialized transaction has specialized parameters. Specialized transactions inherit the actions of general transactions. They can also have specialized actions and additional new ones. For example, when any researcher registers for a meeting, a meeting room can be booked, but when computer researchers register, computer equipment is also booked.

A number of *integrity constraints* are specified. For example, the cost of a trip cannot exceed the budget of the researcher's department. Another example, a temporal constraint, is that a researcher must register for a meeting before attending it.

Figure 9.5 outlines, the long-term process of research administration. It takes the perspective of an individual researcher. This long-term process can be modelled by a Taxis *script* [Chung84, 88] [TDL87].

The process starts at the top, where some initialization is done. Then, moving down the figure, the researcher registers for the meeting and then attends it. This is an example of a temporal constraint, requiring registration before attendance. Note that there may be long periods of time between such activities. In addition, confirmations may be sent to the system, e.g., to indicate that the researcher attended the meeting, so that control may flow to the next step. This is an example of communication, which may involve systems and people.

After the meeting, two series of events can occur in parallel. In one, an expense report is submitted, and then the researcher is reimbursed and

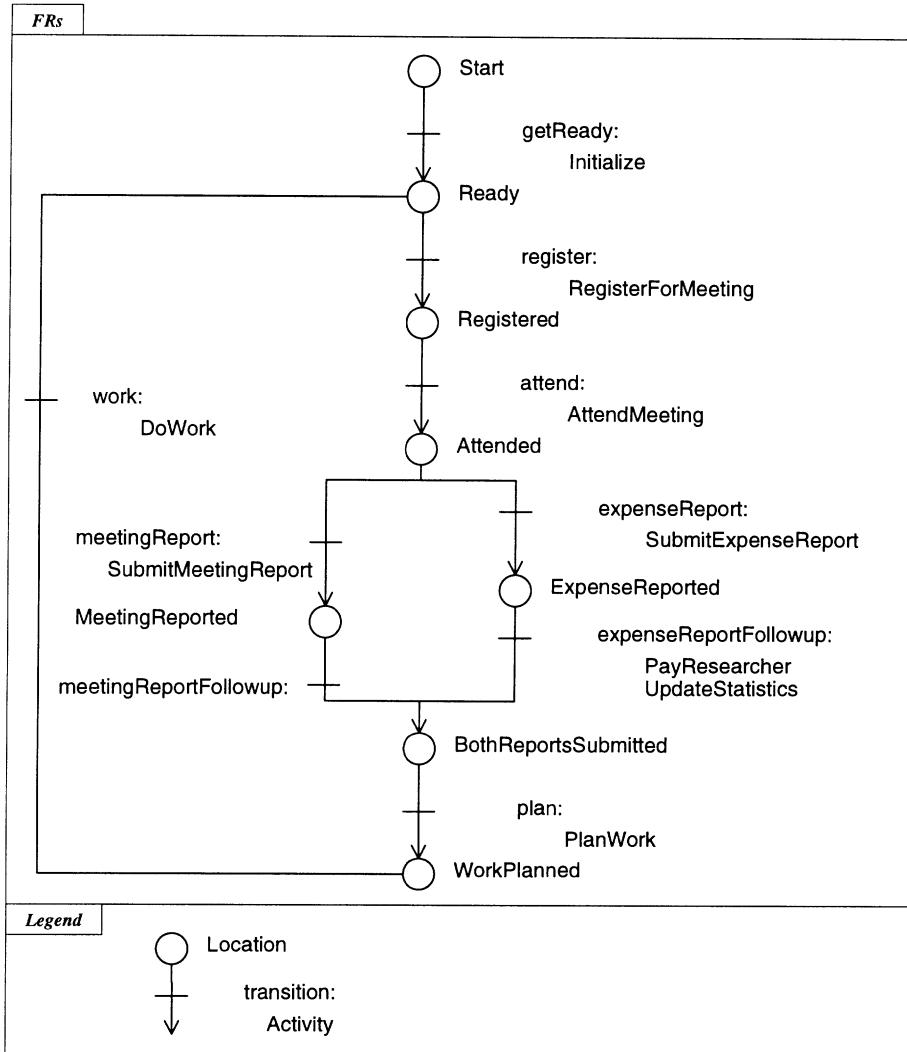


Figure 9.5. A long-term research administration process represented as a script.

statistics are updated. In the other, the researcher can submit a report on the meeting. When both series of events are completed, the system synchronizes, and the researcher can plan future work, and then perform it. Then the entire process can be repeated.

A Taxis script is built around a Petri net skeleton, inspired by [Zisman78]. Control flows between *locations*, which are nodes or states, such as **Start** and **Ready** in Figure 9.5. *Transitions*, which are arcs such as **getReady**, which join locations. Along a transition arc, *operations* are performed. Examples are **Initialize** and **RegisterForMeeting**. In TaxisDL [Borgida93], operations are specified non-procedurally, in terms of *givens* (*preconditions*) and *activities* (*postconditions*).² *Givens* are conditions for activating a transition. *Activities* specify the activity to be performed if a transition is activated. Some of the activities of the script in Figure 9.5 are transactions from the functional requirements (Figure 9.4). For example, the transition **register** has a precondition (given) that the researcher be employed, and a postcondition (activity) that the researcher be registered for the meeting (which can be accomplished by the transaction **RegisterForMeeting**).

Scripts also have interprocess communication primitives, based on Hoare's communicating sequential processes [Hoare78]. This allows scripts to invoke other scripts, communicate with them, and reference any entity. Detailed code is not shown in the figure.

Organizational Workload

The framework allows the developer to consider priorities as well as workload for the organization and system. The basic Performance Requirements Framework in Chapter 8 considered workload in general. Now we also consider *organizational workload*. We use this term to deal with workload aspects more related to the particular domain, organization, applications and operations. Examples of the kind of workload measure include the number of researchers, and how frequently they attend meetings. In comparison, "system workload" expresses workload more in detailed implementational terms, such as the number of input-output operations.

To meet performance requirements, organizational workload is used to help deal with tradeoffs and select among appropriate implementation alternatives. For example, we might expect there to be 2000 employees, of whom 1000 are researchers. On average, each researcher attends 4 meetings per year. In total, 100 meetings are held per year. The 1000 researchers consist of 100 physicists, 700 computer scientists and 300 mathematicians. Counted in each of the last two groups are 100 numerical analysis researchers. There are 30 managers. In addition, from these figures one can draw a number of implications. For example, at a given time, there can be 1000 script instances at a given time, one per researcher. Per year, there are 4000 script *cycles* through 9 transitions each, and therefore 36 000 transitions. Several transactions (e.g., **SubmitExpenseReport**) are invoked 4000 times per year, while some functions

²In Taxis scripts, postconditions of transitions are called "goals." These functional goals are different from the softgoals of the NFR Framework. For consistency, this book uses the terms "givens" for preconditions, and "activities" for postconditions.

(e.g., `ForeignExchange`) might be invoked less frequently. The entities are arranged in classes with inheritance. Some attributes are inherited and others are not. For the `Researcher` class, 50% of the non-inherited attributes are frequently accessed.

9.3 EXTENDING THE PERFORMANCE TYPE

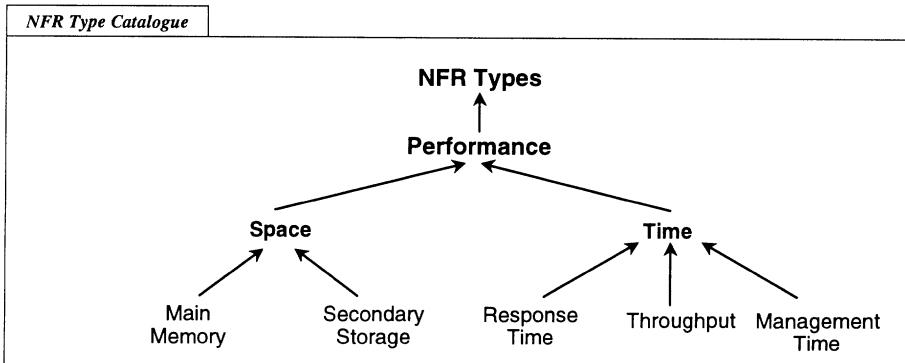


Figure 9.6. Adding ManagementTime to the performance type.

Observe that our sample script has a long-term process which may last several months, during which time the researcher attends a meeting, prepares reports, and so on. However, the computer system will not be continuously performing operations on behalf of this script. Rather, there will be a number of relatively short “bursts” of processing time when script-related operations will be executed. This includes time managing the script, such as determining which givens have been met, and which transactions to activate. We call this time spent managing long-term processes and integrity constraints `ManagementTime`, a subtype of `Time`. Figure 9.6 extends the performance type of Chapter 8 to deal with `ManagementTime`.

We distinguish `ManagementTime` of scripts from `ResponseTime` of transactions because they deal with activities with vastly different time durations. In addition, reducing these two types of `Time` has different kinds of impact. For example, improving the `ResponseTime` of the `PayResearcher` transaction will shorten the duration of this transaction. However, speeding up the activation of the transition which contains `PayResearcher` (i.e., improving the `ManagementTime` of `expenseReportFollowup`) will not significantly decrease the duration of the script. This is because the overall script duration is primarily determined by domain factors, such as the frequency of meetings. To put it another way,

quarterly reports will end up being issued quarterly, no matter how little ManagementTime is used to administer the reporting process.

We use ManagementTime to refer to the time spent managing semantic integrity constraints as well as scripts. This is because semantic integrity constraints and scripts share some implementation considerations [Chung84]. ManagementTime was called “Process Management Time” in Figure 2.1.

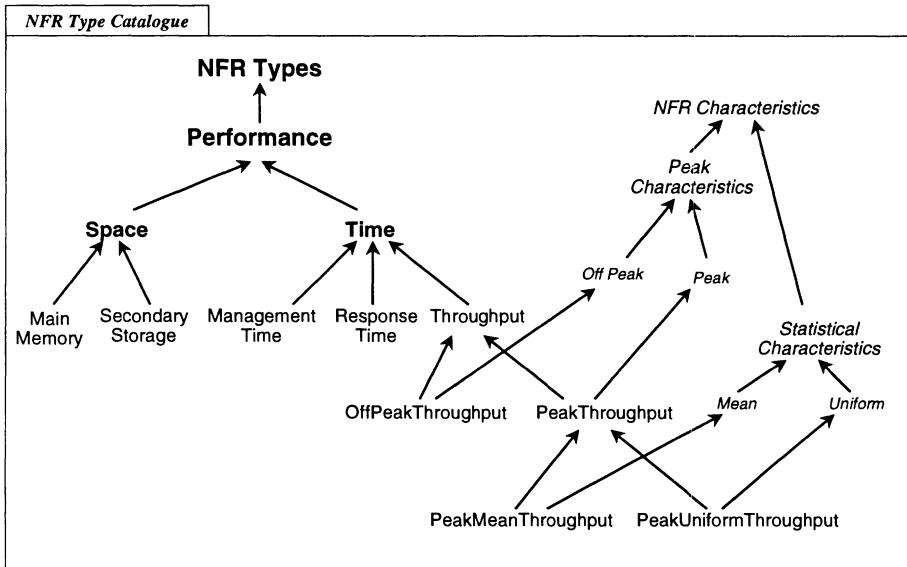


Figure 9.7. Adding ManagementTime and characteristics to the performance type.

In previous chapters we have extended performance and other NFR types to deal with characteristics of NFRs. Now the performance type with management time (Figure 9.6) is extended in Figure 9.7 to deal with characteristics of the performance type. Thus we can express requirements for mean management time, peak-period management time, etc.

9.4 ORGANIZING ISSUES VIA LANGUAGE LAYERS

We can impose additional structure in the presentation and representation of performance issues. This is accomplished through a series of *language layers*. This generalizes the layering of Section 8.1. It organizes information about *interactions* among data model features, implementation techniques and performance aspects of specification languages [Nixon90]. This layering helps control the number of interacting implementation concepts to consider at a time, by

structuring the introduction of domain knowledge associated with specification components.

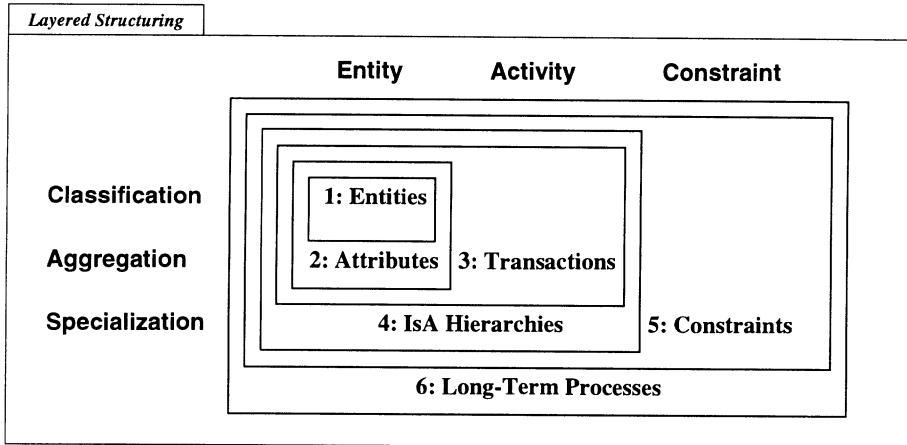


Figure 9.8. Performance issues arranged in a grid.

Consider a grid with two axes, as shown in Figure 9.8. On the horizontal axis, we have the conceptual *linguistic features* (relating to the *ontology*, i.e., what can be expressed about the world) of entities, activities and constraints (based on Greenspan's [Greenspan82] specification units for requirements modelling). On the vertical axis are the *organizational (epistemological) features* of classification, aggregation and specialization [J. Smith77] offered by semantic data models. This grid helps organize issues into a series of language subsets, where larger subsets deal with a larger number of issues.

Now we can express the issues of the grid in terms of layers, corresponding to language features of semantic data models. This is done by selecting a small portion of the grid for lower layers, and then forming higher levels to deal with additional features by enlarging the selected area, extending the selected portion along one axis at a time.

Figure 9.9 shows this layering. Each layer deals with a language *feature* supported by semantic data models. The layering uses a series of language subsets, where higher-level languages introduce additional features and include the features of lower levels. Note that the lowest level language, the target, is numbered 0, and higher-level languages have larger numbers.

The languages and *features* added at each layer are as follows.

Layer 0 is the *target language*, namely the *relational data model*. We consider the database facilities offered by the DBPL language [Borgida90a].

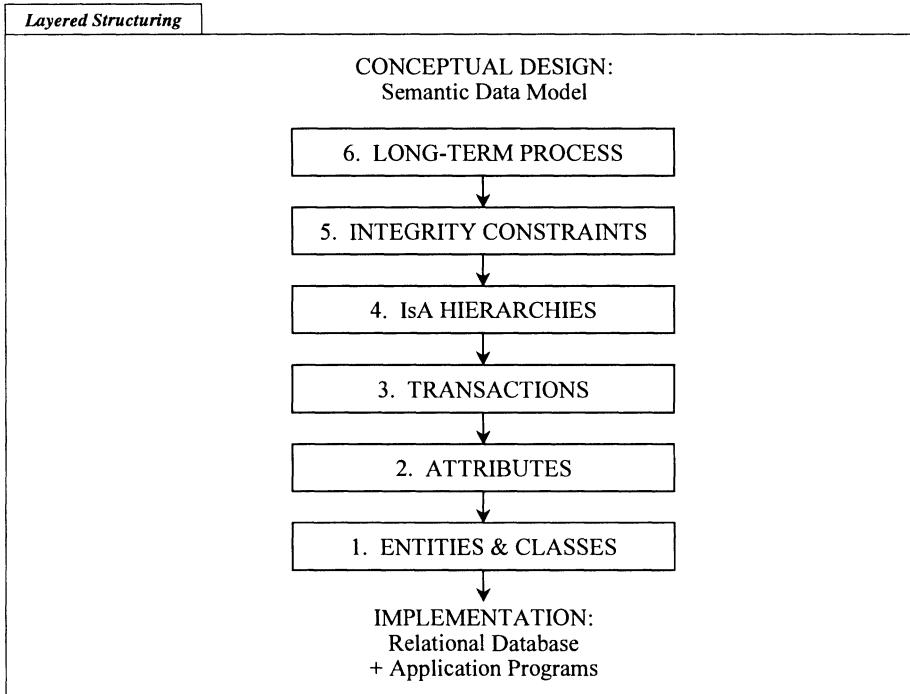


Figure 9.9. Layered organization of performance knowledge.

Layer 1 is the relational model, extended with *entities*, both persistent data entities (such as *John*, an instance of *Researcher*), and finite entities (e.g., integers), arranged in *classes*;

Layer 2 consists of Layers 0 and 1, plus *attributes*, defined on entity classes. Layers 1 and 2 roughly correspond to the Entity-Relationship Model [Chen76].

Layer 3 has the lower layers plus *transactions*, modelled as classes with attributes and instance entities.

Layer 4 includes entities and transactions with attributes, and classes arranged in *IsA hierarchies*. Layers 1 through 4 roughly correspond to the Taxis subset described in [Nixon87].

Layer 5 has the above Taxis subset, extended with *constraints*.

Layer 6 includes the lower layers plus *long-term processes*, whose nature has aspects of entities, activities and constraints [Chung84].

Layers 1 through 6 comprise the *source* specification language, namely Taxis [Chung88] and its successors, TDL³ [Borgida90a] and TaxisDL [Borgida93].

Recall the simple layering of Chapter 8, which organized issues into 3 layers. Now to deal with information systems, we start with comparable layers at the bottom. That is, Layer 1 deals with entities and classes. Layer 2 deals with attributes, so that the bottom 2 layers essentially correspond to the Entity-Relationship model. Layer 3 deals with transactions.

We then extend the layering to consider additional language features. Layer 4 deals with inheritance (IsA) hierarchies, which apply to a variety of language features such as entity classes, but also transactions. Layer 5 deals with integrity constraints (e.g., requiring that researchers' salaries are less than their department's budget). Finally, Layer 6 deals with long-term processes, which unlike transactions, may last several months, and may also involve interaction with other processes and people.

We work top-down, starting with the source (top layer) semantic data model. Successive layers deal with and eliminate data model features such as long-term processes and integrity constraints. This is done by expressing features of higher layers in terms of simpler concepts at lower layers. In the other direction, bottom-up, we consider smaller languages and then larger ones.

This approach helps a developer focus on smaller sets of issues at a time, when desired. In addition, some of the intermediate languages correspond approximately to previously-studied models, including a Taxis subset [Nixon87] and the Entity-Relationship model [Chen76]; this facilitates the review of intermediate results in terms of known characteristics of those languages.

From a *performance* viewpoint, implementation issues are organized in a layered manner similar to what has been used in the study of database performance [Casas86] [Hyslop91].

From a *language* and *data model* viewpoint, issues are organized to consider a series of subsets of data model features and languages.

The framework uses the same layered organization for *selection* among implementation alternatives (while dealing with performance requirements), *prediction* of performance of a selected implementation (which is outlined in [Nixon91]), and in a (partial) performance requirements tool. By sharing a common organization, results should be more easily shared among components and future extensions. In addition, this layered approach should be applicable to other data models, e.g., object oriented models [Kim89].

³Note that the acronym "TDL" is also used in a different context, as the Type Definition Language of the VBASE object-oriented system [Andrews87].

Some Issues for Information System Development

There are many performance concerns for information systems. These concerns are addressed in the framework by the provision of decomposition and operationalization methods, correlation rules, etc.

<u>SOURCE</u>	<u>TARGET</u>
<u>Conceptual Specification</u>	<u>Implementations</u>
6. Long-term processes	- Database transaction - Trigger code
5. Constraints	- Cyclic checking - Trigger code
4. IsA hierarchies: · representation · inherited attributes · transaction body	- Tree - Matrix - Horizontal splitting - Vertical splitting - Static inheritance - Dynamic inheritance
3. Transactions	- Database transaction - Function
2. Attributes	- Static offset fixing - Dynamic offset fixing
1. Entities	- Surrogate identifiers - Physical addresses

Figure 9.10. Space of implementation alternatives arranged by layer.

One class of issues relates to techniques used to implement the data model features of the source specification language. Experience in implementing semantic data models reveals that there are a number of issues and implementation alternatives. Part of the large space of implementation alternatives is shown in Figure 9.10, which organizes the issues using the layering structure.

The figure shows different techniques that can be used to implement a particular source feature. For example, at Layer 2, two different methods are shown for determining the offsets of an attribute of an entity.

Other issues and sources of methods arise from database design and implementation, and from techniques for building performance into systems.

Now we will consider methods for dealing with information system performance. This extends the presentation of the Performance Requirements Framework of Chapter 8. The presentation here is organized by language features, starting generally at lower layers. It is illustrated by the research administration example.

9.5 DECOMPOSITION METHODS FOR HANDLING DATA MANAGEMENT

We consider a number of decomposition methods for data management operations (and developer-defined transactions) for entities, classes and attributes. These are considered at the lower 3 layers.

Recall from Chapter 8 that *Info* is an information item (or an operation on an item), *Operation* is an operation on an item, and *Layer* is a layer.

- **Operational method:**

A performance softgoal for an information item *Info* (such as a class or an attribute of a class) at a particular *Layer* is helped (but not necessarily satisfied) by a softgoal for operations on the item:

$$\begin{aligned} & \text{Performance[operationsOn(Info), Layer]} \\ & \text{HELPS Performance[Info, Layer]} \end{aligned}$$

In turn, the performance softgoal for the set of operations on the item is refined into a set of performance softgoals, one for each *operation* on the item:

$$\begin{aligned} & \text{Performance[Operation}_1(\text{Info}), \text{Layer}] \text{ AND } \dots \text{ AND} \\ & \text{Performance[Operation}_n(\text{Info}), \text{Layer}] \\ & \text{SATISFICE Performance[operationsOn(Info), Layer]} \end{aligned}$$

- **Primitive—Developer-DefinedOperations method:**

This method refines a performance softgoal for operations on an item into softgoals for its primitive (pre-defined) and its developer-defined operations:

$$\begin{aligned} & \text{Performance[primitiveOperationsOn(Info), Layer]} \text{ AND} \\ & \text{Performance[developerOperationsOn(Info), Layer]} \\ & \text{SATISFICE Performance[operationsOn(Info), Layer]} \end{aligned}$$

In turn, the softgoal for primitive operations is refined into softgoals for each primitive operation, using the **Primitive Operations** method:

$$\begin{aligned} & \text{Performance}[\text{primitiveOperation}_1(\text{Info}), \text{Layer}] \text{ AND } \dots \text{ AND} \\ & \text{Performance}[\text{primitiveOperation}_n(\text{Info}), \text{Layer}] \\ & \text{SATISFICE Performance}[\text{primitiveOperationsOn}(\text{Info}), \text{Layer}] \end{aligned}$$

Similarly, the softgoal for developer-defined operations is refined into softgoals for each developer-defined operation, using the **Developer-Defined Operation** method:

$$\begin{aligned} & \text{Performance}[\text{developerOperation}_1(\text{Info}), \text{Layer}] \text{ AND } \dots \text{ AND} \\ & \text{Performance}[\text{developerOperation}_n(\text{Info}), \text{Layer}] \\ & \text{SATISFICE Performance}[\text{developerOperationsOn}(\text{Info}), \text{Layer}] \end{aligned}$$

- **Individual-BulkOperations** method:

This method refines a performance softgoal for operations into softgoals for operations manipulating one or many items.

$$\begin{aligned} & \text{Performance}[\text{individualOperationsOn}(\text{Info}), \text{Layer}] \text{ AND} \\ & \text{Performance}[\text{bulkOperationsOn}(\text{Info}), \text{Layer}] \\ & \text{SATISFICE Performance}[\text{operationsOn}(\text{Info}), \text{Layer}] \end{aligned}$$

For example, operations on individual researchers (such as registering for a meeting) can be distinguished from operations involving all researchers (such as printing expense summaries for all staff). This allows us to distinguish two groups of operations, whose performance can have quite different natures.

In turn, the softgoal for operations on an individual information item is refined into softgoals for each such operation:

$$\begin{aligned} & \text{Performance}[\text{individualOperation}_1(\text{Info}), \text{Layer}] \text{ AND } \dots \text{ AND} \\ & \text{Performance}[\text{individualOperation}_n(\text{Info}), \text{Layer}] \\ & \text{SATISFICE Performance}[\text{individualOperationsOn}(\text{Info}), \text{Layer}] \end{aligned}$$

Similarly, the softgoal for bulk operations is refined into softgoals for each such operation:

$$\begin{aligned} & \text{Performance}[\text{bulkOperation}_1(\text{Info}), \text{Layer}] \text{ AND } \dots \text{ AND} \\ & \text{Performance}[\text{bulkOperation}_n(\text{Info}), \text{Layer}] \\ & \text{SATISFICE Performance}[\text{bulkOperationsOn}(\text{Info}), \text{Layer}] \end{aligned}$$

- **EntityManagement** method:

A performance softgoal for operations on an information item is refined into softgoals for the primitive data management operations on the softgoal.

$$\begin{aligned} & \text{Performance[entityOperationsOn(Info), Layer]} \\ & \text{HELPS Performance[operationsOn(Info), Layer]} \end{aligned}$$

Good performance for entity operations will help, but not necessarily satisfy, good performance for all the operations on the item.

The performance softgoal for entity operations can be satisfied by satisfying performance softgoals for all the primitive data management operations: creation, retrieval, update and removal of entities:

$$\begin{aligned} & \text{Performance[create(Info), Layer] AND} \\ & \text{Performance[retrieve(Info), Layer] AND} \\ & \text{Performance[update(Info), Layer] AND} \\ & \text{Performance[remove(Info), Layer]} \\ & \text{SATISFICE Performance[entityOperationsOn(Info), Layer]} \end{aligned}$$

An abbreviation for `entityOperationsOn` is `access`.

- **Single-MultipleAttribute** method:

Suppose a performance softgoal for operations on an information item has been refined by the individual attribute method to a softgoal for operations on the attributes of the item.

$$\begin{aligned} & \text{Performance[operationsOn(attributes(Info)), Layer]} \\ & \text{HELPS Performance[operationsOn(Info), Layer]} \end{aligned}$$

This softgoal can then be refined into three softgoals. One softgoal addresses the set of operations on precisely one attribute, another addresses the set of operations on all attributes, and the other addresses the set of operations on an intermediate number of attributes.

$$\begin{aligned} & \text{Performance[operationsOn(oneAttribute(Info)), Layer] AND} \\ & \text{Performance[operationsOn(someAttributes(Info)), Layer] AND} \\ & \text{Performance[operationsOn(allAttributes(Info)), Layer]} \\ & \text{SATISFICE Performance[operationsOn(attributes(Info)), Layer]} \end{aligned}$$

- **SchemaChange** method:

We can also consider the impact of schema changes on the storage and manipulation of entities. While the conceptual specification (or schema) of an information system may be expected to remain constant in some cases, in other cases it may be expected to change. For example, it may be desired to be able to add new specializations of **ComputerResearcher** over time, without requiring the entire system to be shut down and restarted. This method refines a performance softgoal for an information item, on the basis of whether the schema is expected to change. One offspring deals with performance for the case of a static schema; the other handles schemas which change.

```

Performance[inStaticSchema(Info), Layer] AND
Performance[inDynamicSchema(Info), Layer]
SATISFICE Performance[Info, Layer]

```

9.6 METHODS FOR HANDLING INHERITANCE HIERARCHIES

IsA hierarchies have an impact on many data model features, including entity classes and transaction classes. Each of these has a variety of implementation alternatives.

There are several operationalizing and decomposition methods from semantic data model implementation experience. Several of the operationalization methods relate to implementation techniques for inheritance (**IsA**) hierarchies, which are considered at Layer 4.

An important observation is that **IsA** hierarchies result in collections of attribute values whose appearance is more like a “staircase” than a grid (Figure 9.11). Here **Researcher** has attribute **NumOfPapers**, which is inherited by **ComputerResearcher** and **NumericalAnalysisResearcher**, the specializations of **Researcher**. Similarly **ComputerResearcher** has a non-inherited attribute **System**, which is inherited by **NumericalAnalysisResearcher**, which has a non-inherited attribute, **MathPackage**. In the illustration, not all attributes are shown.

Data Hierarchies

Let's consider **IsA** hierarchies for the case of storage of entity (data) classes.

When storing large amounts of persistent data, a natural first choice is relational database technology. If we can represent attribute values within a relational schema, we can exploit the efficient storage and access techniques already developed for relational databases. However, the collection of attributes does not appear like a relational table. As a result, a simple relational representation may waste space

For example, one alternative for representing all attributes is to use a universal relation which has one tuple per entity, one column per attribute, and null values for attributes not applicable to an entity. Obviously, such a representation would be highly inefficient with respect to space usage, although it allows static determination of attribute offsets. Let's consider some alternatives.

	NumOfPapers	System	MathPackage
Num.AnalysisResearcher			
ComputerResearcher			
Researcher			

Figure 9.11. Arrangement of attribute values in the presence of inheritance hierarchies.

	entity	NumOf Papers	System	Math Package
OnlyNum.AnalysisResearchers	entity	Num0036	UNIX	Maple
		Num0035	OS/2	MatLab
OnlyComputerResearchers	entity	Com0034	Windows	
		Com0033	Win98	
OnlyResearchers	entity	Res0032	20	
		Res0031	22	

Figure 9.12. Horizontal splitting of attributes.

All Researchers		AllComputer Researchers		AllNum.Analysis Researchers	
entity	NumOf Papers	entity	System	entity	Math Package
Num0036	24	Num0036	UNIX	Num0036	Maple
Num0035	28	Num0035	OS/2	Num0035	MatLab
Com0034	19	Com0034	Windows		
Com0033	23	Com0033	Win98		
Res0032	20				
Res0031	22				

Figure 9.13. Vertical splitting of attributes.

A second, more interesting, alternative involves using one relation per class [J. Smith77], while another uses one relation for each generalization sub-lattice that is part of the conceptual schema [Zaniolo83].

When using one relation per class, one may store all attributes (newly defined or inherited) of a particular class in the corresponding relation (**HorizontalSplitting**, Figure 9.12), or only the newly defined attributes, as done in [J. Smith83] (**VerticalSplitting**, Figure 9.13). In the examples, we assume that entities are implemented by assigning a unique internal identifier to each entity. Also note that only some attributes are shown in the diagrams.

Some implementations give the system developer more flexibility in selecting storage mechanisms to meet more selectively the needs of a particular application. ADAPLEX [Chan82] supports a form of both vertical and horizontal partitioning of entities and their attribute values. In fact, arbitrary predicates, specifying which entities to include in which partition, are available to the system developer.

A variant of vertical splitting, which does not necessarily deal with inheritance, is **SelectiveAttributeGrouping**. It stores together attributes chosen by the implementor (not necessarily on the basis of inheritance).

<i>entity</i>	<i>attribute</i>	<i>attributeValue</i>
Num0036	NumOfPapers	24
Num0036	System	UNIX
Num0036	MathPackage	Maple
Num0035	NumOfPapers	28
Num0035	System	OS/2
Num0035	MathPackage	MatLab
Com0034	NumOfPapers	19
Com0034	System	Windows
Com0033	NumOfPapers	23
Com0033	System	Win98
Res0032	NumOfPapers	20
Res0031	NumOfPapers	22

Figure 9.14. Attributes stored as “triples.”

Another alternative (which was considered for Taxis [Nixon87]) stores a tuple of “triples”: for every attribute value of every entity, one stores the entity identifier, the attribute name, and the attribute value. The example would be represented as in Figure 9.14. The “triples” method does not offer the best performance, but does permit translation to a schema with a small,

fixed number of relations, that requires no refinement or reformatting, even when additional classes and attributes are defined.

Another option is a modified universal relation approach. One issue is to determine the field in the relation where a given attribute will be stored. Observe that typical schemata have many general classes which have disjoint extensions (e.g., integers, strings, exceptions, data classes). Consider the IsA sub-lattice underneath each such general class: the set of attributes used in each sub-lattice will typically be much smaller than the set corresponding to the most general class (say, Any). A separate “universal” relation can be created for each general class. This can be repeated at lower levels where extensions are disjoint, for example, for data classes Person and Building. As a result, space can be saved by allocating space for only the attributes used in each sub-lattice, while still offering static determination of offsets, *provided* the translator can determine which general class is to be accessed. Note however that if static analysis of an expression can only determine that it refers to an overly general class (e.g., AnyData which includes Person and Building) the field (or offset) may have to be calculated dynamically, necessitating some run-time conversions between attributes of a general class and fields of a particular relation.

Relevant parameters for selecting among alternatives include the relative proportions of data management operations (entity creation, entity removal, attribute retrieval, attribute modification, and bulk retrievals), as well as the expected frequency of schema modifications over time. An analysis of the different alternatives for attribute storage is presented in [Nixon87]. While there are tradeoffs, it is clear that one needs to go beyond a simple relational representation in order to obtain efficiency.

Tuple Storage: Operationalization Methods.

To implement hierarchies of entity classes, we consider the storage of tuples in a relational database.

- Using **FewAttributesPerTuple**, if a tuple contains few attributes (e.g., vertical splitting), some time softgoals can be positively satisfied, since a unit of storage will contain several tuples, which can be retrieved together. However, with this information alone, we can't necessarily determine the impact on space softgoals.

FewAttributesPerTuple[Info, Layer] *HELPS* Time[Info, Layer]

- Using **SeveralAttributesPerTuple**, if a tuple contains several attributes (e.g., horizontal splitting), some space softgoals can be positively satisfied. For example, if this reduces the total number of tuples needed to store information about a class, we may decrease the total space needed for overhead for each tuple (for internal identifiers, pointers, etc.).

SeveralAttributesPerTuple[Info, Layer] *HELPS* Space[Info, Layer]

We can't in general determine the impact on time.

- Using **ReplicateDerivedAttribute**, explicitly storing one attribute which is defined as being derived from another, will have a negative impact on space. The impact on time softgoals may vary. For example, it may be positive for retrievals, but negative for updates, as more than one attribute may need to be updated.

ReplicateDerivedAttribute[Info, Layer] HURTS Space[Info, Layer]

We can extend this method to deal with data management operations.

ReplicateDerivedAttribute[Info, Layer] HELPS
Time[retrieve(Info), Layer]

ReplicateDerivedAttribute[Info, Layer] HURTS
Time[update(Info), Layer]

For these two methods, we can't in general determine the impact on space.

Tuple Manipulation: Operationalization Methods.

For hierarchies of entity classes, we also need to consider operations on attributes. The effectiveness of operations will greatly depend on the storage method chosen.

- Using **AccessManyAttributesPerTuple** (e.g., accessing information stored using horizontal splitting), if many of the attributes in a tuple will frequently be accessed, time softgoals can be positively satisfied. This is because one access will retrieve several values.

AccessManyAttributesPerTuple[Info, Layer] HELPS Time[Info, Layer]

- Using **AccessManyTuplesPerRelation** (e.g., accessing information stored using vertical splitting), if many of the tuples in a relation will be accessed, time softgoals can be positively satisfied. For example, if one needs to access several attributes of an entity, this method is helpful.

AccessManyTuplesPerRelation[Info, Layer] HELPS Time[Info, Layer]

Transaction Hierarchies

Some data models (such as Taxis) offer transactions arranged in hierarchies. For example, if the entity class `Researcher` has a specialization `ComputerResearcher`, the transaction class `RegisterForMeeting(Researcher)` can have a specialization `RegisterForMeeting(ComputerResearcher)` (Figure 9.4). And in executing a call to the `RegisterForMeeting` transaction (e.g., `r.RegisterForMeeting`) for a given researcher `r`, the system needs to know the “minimum class(es)” of the parameter (e.g., is `r` a computer researcher, or just a researcher?) in order to invoke the “most specialized transaction version.” For example, if `r` is a computer researcher, then `RegisterForMeeting(ComputerResearcher)` will be called; otherwise `RegisterForMeeting(Researcher)` will be. Let’s consider some issues which arise from this feature.

- *Transaction Calls:* Based on the minimum class(es) of the actual arguments, the “most specialized transaction version” must be selected for invocation. The difficulty of this selection task grows with the number of parameters, the number of specializations of each parameter class, and the associated topology of the IsA hierarchy (multiple inheritance, depth of hierarchy, etc.). Depending on the degree of complications, different techniques, such as table look-up, hashing, and inverse indexing of attributes may be selected. In addition, for frequently called transactions, it may be desirable to use more space to minimize execution time.
- *Code Inheritance:* Another issue is whether inherited code should be replicated. There is a spectrum of options. *Dynamic inheritance* (performed at each transaction invocation) may be useful when the schema is continuously being modified, at least for infrequently-called transactions. The system could monitor calls to determine which transactions should be re-compiled using another technique. *Static inheritance*, with full replication of code, may be suitable for transactions which are called very frequently, or for which fast response time is paramount. Some *intermediate (hybrid) methods* are reviewed in [Nixon89]: they statically prepare tables of code locations (offsets in activation records), without replicating code, but requiring extra de-referencing of code locations during execution. One can imagine an hybrid method, whereby large inherited code segments are not replicated (saving space), while smaller ones are copied (saving de-referencing time).

There are several relevant parameters for choosing among implementation alternatives. They include the topology of the transaction hierarchy, the relative size of non-specialized inherited code in the schema and the relative frequency of using it, the expected frequencies of invocation of transactions, and limitations on space available for code and allowable response times.

Decomposition Method for Transaction Hierarchies.

Consider an IsA hierarchy of transactions. For example, `Reimburse(ComputerResearcher)` is a specialization of `Reimburse(Researcher)`.

- **CommonOperations** method: By the **CommonOperations** method, operations on the general transaction may be refined into operations which are common to all versions, and operations which are not.

Time[commonOperations(Operation(Info)), Layer] AND
 Time[nonCommonOperations(Operation(Info)), Layer]
 SATISFICE Time[Operation(Info), Layer]

This is a specialization of the **ImplementationComponents** method.

Operationalization Methods for Transaction Hierarchies.

Some operationalization methods for dealing with transaction hierarchies draw on the software performance engineering principles for building performance into systems. They include specializations of early and late fixing.

- **StaticCodeInheritance** is a specialization of **EarlyFixing**. Due to replication of code, there is a negative impact on space.

StaticCodeInheritance[Info, Layer] HELPS Time[Info, Layer]
 StaticCodeInheritance[Info, Layer] HURTS Space[Info, Layer]

- **DynamicCodeInheritance** is a specialization of **LateFixing**. By avoiding replication of code, there is a positive impact on space.

DynamicCodeInheritance[Info, Layer] HELPS Space[Info, Layer]
 DynamicCodeInheritance[Info, Layer] HURTS Time[Info, Layer]

- *Execution Ordering Methods*, such as **PerformFirst** and **PerformLater** were presented in Section 8.4. They can be applied to the implementation of conceptual specification languages. For example, TaxisDL [Borgida93] does not require the specification of the order of execution of components of a transaction.

9.7 METHODS FOR HANDLING INTEGRITY CONSTRAINTS AND LONG-TERM PROCESSES

Semantic integrity constraints and long-term processes are language features found in some data models, including Taxis. They share some implementation considerations [Chung84].

Integrity Constraints

A large information system can have many entities, and many integrity constraints associated with each entity. Efficient enforcement of integrity constraints is essential, as the system cannot check every constraint at every point

in time. Hence, one must consider how and when the system searches among all the constraints which may be violated, when implementing semantic integrity constraints. Integrity constraints are considered at Layer 5.

Since the system will be continually looking for unsatisfied constraints, an important performance issue is whether the search is *exhaustive* or *selective*. Consider a constraint that every researcher must have a salary which does not exceed the budget of his or her department. *Exhaustive search* involves the repeated checking of every constraint after any change to the system. In the example, every researcher's salary could be examined, and this entire process might be done frequently. It would be better to analyse the condition at compilation, and produce code with *selective checking*; in the example, a compiler would emit code which does not check this condition for any salary *decrease*. Here, a particular constraint may be satisfied by simply checking the direction of change. In other cases, modified values may be compared to aggregate values (such as maxima and minima) to determine constraint satisfaction. These are some of the cases where a constraint can be implemented by examining only one or two values, which are selected at compilation time.

It turns out that there are tradeoffs among the different techniques for selective checking [Chung88]. For example, if attribute values are frequently modified, overhead for maintaining maxima and minima will also increase. The form of the constraint, and the expected patterns of modifications will be factors in determining which compilation techniques should be used. There are additional options concerning integrity constraint enforcement.

One option is for enforcement of integrity constraints on entity classes to be *centralized*. Using this approach, all updates are handled by a central mechanism which examines the relevant definitions to determine which constraints to check, e.g., as in Taxis [Chung88]. Another option is *application-based* (decentralized) enforcement, where the developer chooses which application transactions will have code to check particular constraints. Centralized checking imposes some structure on enforcement, while decentralized checking offers flexibility and individual optimization, possibly at the cost of consistency. For example, if a certain constraint can *only* be violated from execution of *one* transaction, the constraint can efficiently and safely enforced directly in that one transaction. Suppose the **Researcher** class has the constraint that a researcher registering for a meeting must have a good record of submitting timely reports. By using knowledge of where this constraint is used, it could be directly checked in the **RegisterForMeeting** transaction, and nowhere else, thus reducing overhead.

For operationalizations of integrity constraints, there are alternatives with varying performance. These include **CyclicChecking** [Zisman78] (exhaustive circular examination of all entities with a particular constraint) and a **TriggerMechanism** (event-driven selective checking [Chung88] as in active databases) which can often be more efficient. Temporal integrity constraints can be arranged in a list ordered by the time applicable to the constraint; this can avoid exhaustive checking of the list.

Handling Long-Term Processes

Another, related, issue is the efficient implementation of a data model's facilities for concurrency and long-term processes. In Taxis, they are modelled by *scripts*.

Operationalization Methods.

There are several problems in the management of the type of concurrency introduced by a modelling construct such as that of scripts. In general, there will be a large number of active script instances, states and transitions at any one time, and it is important to use a strategy for process scheduling that is both fair and efficient. The key to any such strategy is the determination and management of transitions that are eligible for activation or are currently active. One needs to consider techniques for both scheduling (such as those available from systems programming) *and* optimized detection of condition satisfaction. If an implementation uses an "off-the-shelf" technique which *only* provides *scheduling* (such as monitors provided by an operating system), but does not incorporate optimizing techniques (such as selective checking) then performance will be poor. For example, if the condition that a particular researcher has been hired is translated to a monitor-like "wait" construct, the result could be very inefficient, as the condition would be evaluated repeatedly. Now if there are a thousand instances of the script, (one for each researcher), the large amount of checking may slow down the system altogether. Again *selective checking* will improve the situation. For example, the condition could be checked for a particular researcher when that researcher is registering for a meeting.

CyclicChecking involves a sequential scan, but only of transitions whose input states are active.

Another approach, **Triggering**, is to only examine transitions whose conditions may have now become satisfiable and have all input states active.

Cyclic checking may actually provide reasonable performance when there are few transitions, and when every transition whose input states are satisfied also has its conditions satisfied. In most other situations, however, triggering is much more efficient. One exception is when many transactions are de-triggered, i.e., true conditions become false.

Decomposition Methods.

There are a number of types of decomposition methods to deal with long-term processes. Some of them involve data model features of scripts.

Let's consider **DataModelFeature** decomposition methods. These decompose a softgoal by considering the data model features used in the topic of the softgoal.

- **ScriptComponents** method:

A softgoal for a script *Script* is refined into softgoals for its locations and for its transitions.

$$\begin{aligned} & \text{Performance[locations}(Script), \text{ Layer]} \text{ AND} \\ & \text{Performance[transitions}(Script), \text{ Layer]} \\ & \text{SATISFICE Performance[}(Script), \text{ Layer]} \end{aligned}$$

- **TransitionComponents** method:

This method refines a softgoal for a script's *Transition*, e.g., `RemindToContact`, into softgoals for its constituent constructs, namely givens and actions:

$$\begin{aligned} & \text{Performance[givens}(Transition), \text{ Layer]} \text{ AND} \\ & \text{Performance[actions}(Transition), \text{ Layer]} \\ & \text{SATISFICE Performance[}(Transition), \text{ Layer]} \end{aligned}$$

The `ScriptComponents` and `TransitionComponents` methods are shown in Figure 9.16, along with a number of other methods for scripts and integrity constraints.

Another group of methods are the `IndividualComponents` methods. These methods refine a performance softgoal for a particular feature (e.g., transitions of a script) into softgoals for each individual component of the feature (e.g., each transition).

For long-term processes (Layer 6), specializations of the Individual Components method include the `IndividualLocations`, `IndividualTransitions`, `IndividualGivens`, and `IndividualActivities` methods. Let's consider one of these methods.

- **IndividualActivities** method:

This method refines a performance softgoal for the activities (postconditions) of a transition into softgoals for each of the activities of the transition:

$$\begin{aligned} & \text{Performance[Activity}_1(Transition), \text{ Layer]} \text{ AND } \dots \text{ AND} \\ & \text{Performance[Activity}_n(Transition), \text{ Layer]} \\ & \text{SATISFICE Performance[activities}(Transition), \text{ Layer]} \end{aligned}$$

Refining Management Time into Time Softgoals

After `ManagementTime` softgoals are addressed at Layers 6 and 5, there may be remaining time-related development to be done at these and lower layers. Since aspects relating to long-term process and integrity constraints may have been addressed at the higher layers, the remaining softgoals at these and lower layers

may address other **Time** softgoals, such as **ResponseTime** or **Throughput**. The **ManagementTime** to **Time** method allows this kind of *inter-layer refinement*:

$$\begin{aligned} \text{Time[Info, Layer}_m\text{]} \text{ HELPS MgmtTime[Info, Layer}_n\text{]} \\ \text{WHERE } 1 \leq \text{Layer}_m \leq 6 \text{ AND } 5 \leq \text{Layer}_n \leq 6 \\ \text{AND } \text{Layer}_m \leq \text{Layer}_n \end{aligned}$$

Unlike the **FlowThrough** method, this method allows the types of the parent and offspring softgoals to differ.

9.8 ORGANIZING PERFORMANCE METHODS

We consider how to organize the large number of performance methods.

A premise of this framework is that *performance and development knowledge must be organized*. As we have seen earlier, there are many types of knowledge which must be considered when addressing performance requirements and there is knowledge about a large number of information system implementation alternatives, whose performance can interact in complex ways. Accordingly, the framework represents and organizes performance concepts and knowledge of information systems, their specification, implementation and performance. It represents and organizes results from semantic data models and databases, as well as principles for building performance into systems [C. Smith86]. This knowledge can be organized using a powerful object-oriented knowledge representation language such as Telos [Mylopoulos90]. Developers can use this knowledge, along with characteristics of the system (e.g., domain knowledge, organizational workload, and priorities) and their own expertise, to deal with the particular performance requirements of the particular system under development, and produce customized solutions.

There are a number of performance refinement methods. How do we organize them? Refinement methods are divided into decomposition, operationalization and argumentation methods (Section 4.1). They can be further grouped by the types of the parents and offspring. For example, all operationalization methods produce operationalizing softgoals, but some have an NFR softgoal as a parent, while others have an operationalizing softgoal. They can also be grouped into predefined- and developer- defined methods.

Figure 9.15 organizes decomposition methods (“DM”). (In Figures 9.15 and 9.17 the “top” level is displayed at the left of the figure.) Since softgoals have types, topics, layer topics, and additional attributes (such as priorities, discussed below), methods can refine softgoals along each of these dimensions. We can form broad groups of methods for each of these dimensions, which are shown as the second level of decomposition methods.

These groups of methods can be further sub-divided. For example, the layer-based methods can be specialized into groups of methods for each of the layers. Likewise, topic-based methods can be specialized: there are groups of

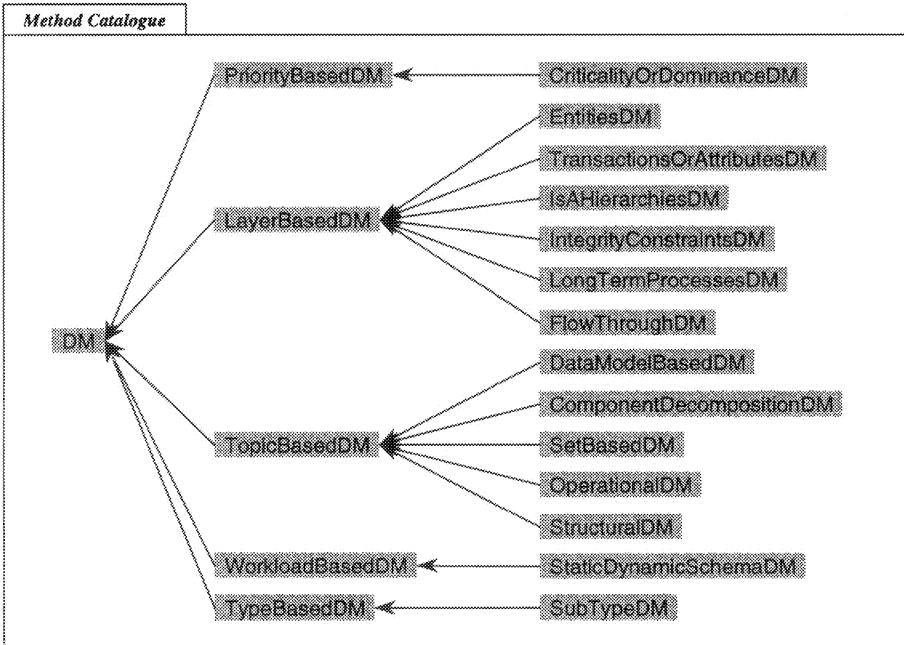


Figure 9.15. Organization of Performance decomposition methods.

methods which produce offspring based on operations on the topic, data model features of the topic, etc.

Most of the actual decomposition methods are specializations of the groups shown in Figure 9.15. In fact, many of the methods are actually formed by multiple inheritance, from two or more such groups. For example, the **TransitionComponents** method (lower right of Figure 9.16) is one of the methods which is a specialization of both the **LayerBased** method for a particular layer, Layer 6 (long-term processes), and the **TopicBased** method for data model feature component decomposition. Similarly, the **IndividualTransitions** method (lower left of Figure 9.16) is a specialization of the method for Layer 6 and the topic-based method for individual components decomposition.

Similarly, *operationalizing methods* ("OM" at the "top" of Figure 9.17, where the "top" is displayed at the left of the figure) are arranged into broad groups, which are then used to define more specific methods via multiple inheritance. Here, the broad groups again include types, topics and layer topics. However, there are additional groups which represent sources of operationalization methods. These include implementation techniques from semantic data

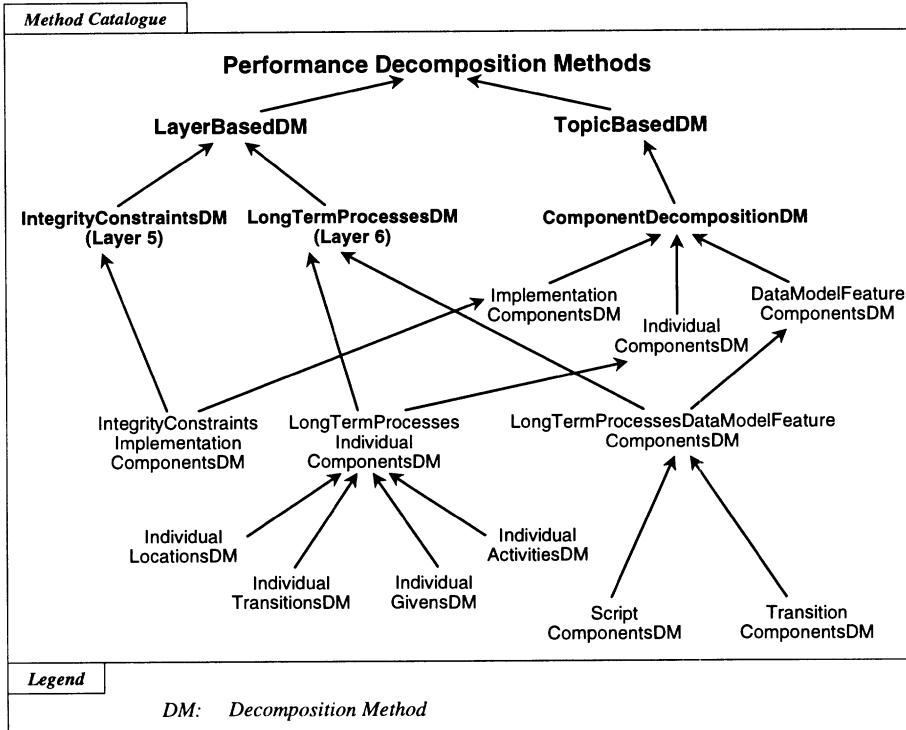


Figure 9.16. Some Performance decomposition methods for Layers 6 and 5.

modelling, object-oriented systems, and database systems, as well as software performance engineering principles for building performance into systems.

By arranging methods into groups, particular refinement methods can be defined via multiple inheritance from the (somewhat orthogonal) broad groups of methods. This arrangement should help the developer in understanding, representing and using a fairly large number of methods, for several reasons. First, the groups should help a developer search a catalogue. In addition, the use of the groupings can help a developer focus attention on a smaller number of methods. A developer who understands aspects of a group of methods will understand some aspects of more specific groups and their individual methods. In addition, a developer will understand some aspects of methods formed by multiple inheritance from two or more general methods, if they are already understood. Finally, the groupings can provide some guidance for cataloguing new methods when they are defined.

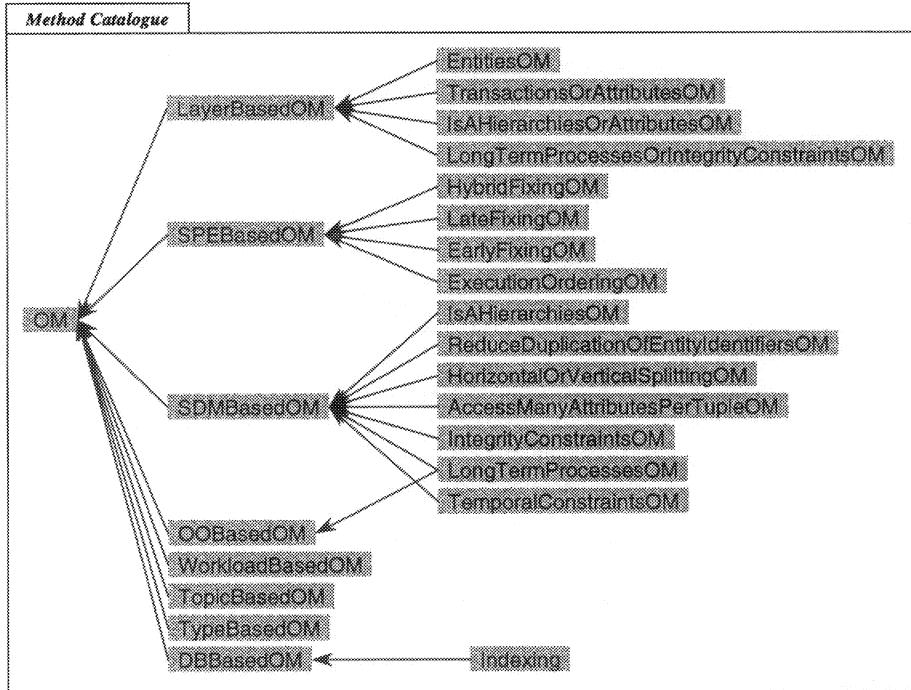


Figure 9.17. Organization of Performance operationalization methods.

The same organization of knowledge can be made available to help a developer use tool support for the framework. Tools can catalogue and organize performance concepts and development methods, including results from semantic data models and principles for developing responsive systems. Tools can also record knowledge of NFR Framework concepts, the source language, and the particular domain being considered.

9.9 ORGANIZING CORRELATIONS

Correlations show positive and negative interdependencies among softgoals. Figure 9.18 shows the impact of some operationalizations for information system development. Here some operationalizations deal with tuple storage for data hierarchies, and others deal with transaction hierarchies. In addition to standard time and space softgoals, the performance softgoals deal with data management operations, including retrieval and updating. Interestingly, some operationalizations have both positive and negative impacts.

Correlation Catalogue		to parent NFR Softgoal			
Contribution of offspring <i>Operationalizing Softgoal</i>		Time [Info, Layer]	Time [retrieve(Info), Layer]	Time [update(Info), Layer]	Space [Info, Layer]
<i>Tuple Storage Operationalizations:</i>					
FewAttributes PerTuple [Info, Layer]	HELPS				
SeveralAttributes PerTuple [Info, Layer]					HELPS
Replicate DerivedAttribute [Info, Layer]			HELPS	HURTS	HURTS
<i>Transaction Hierarchy Operationalizations:</i>					
StaticCodeInheritance [Info, Layer]	HELPS				HURTS
DynamicCodeInheritance [Info, Layer]	HURTS				HELPS

Figure 9.18. A performance correlation catalogue for information system development.

9.10 ILLUSTRATION

We now illustrate some of the issues presented in this chapter. We continue with the research administration example. We focus on issues related to IsA hierarchies, at Layer 4.

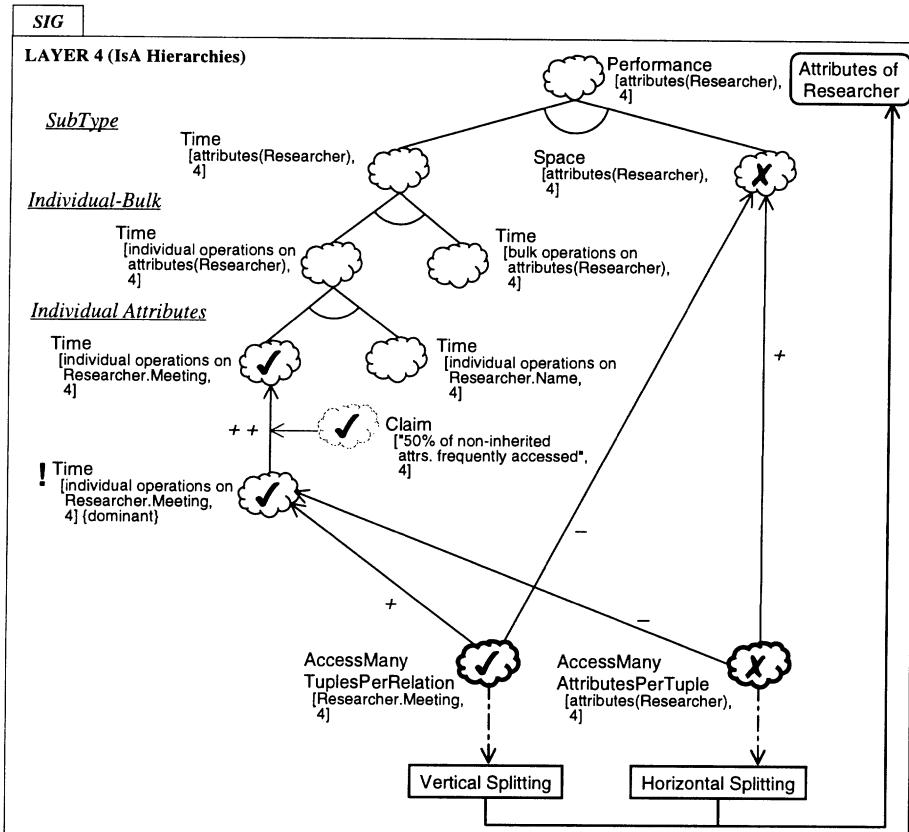


Figure 9.19. Dealing with inheritance hierarchies.

The initial requirement is for good performance for the attributes of **Researcher**. This is shown as the softgoal **Performance[attributes(Researcher), 4]** at the top of Figure 9.19.

Using the **SubType** method, this softgoal is decomposed into time and space softgoals for the attributes of **Researcher**.

The developer focusses on the time softgoal. To explore the different kinds of operations on researchers, the time softgoal is refined, using the **Individual-Bulk Operations** method.

The developer focusses on the time softgoal for individual operations on the attributes of **Researcher**. To consider individual operations on the various attributes of **Researcher**, the **IndividualAttributes** method is used.

The developer focusses on individual operations on the **Meeting** attribute. The developer observes that attributes such as **Name**, which **Researcher** inherits from its parent class **Employee**, are not frequently accessed. However, 50% of non-inherited attributes, including **Meeting**, are frequently accessed. Hence **Time[individual operations on Researcher.Meeting, 4]** is prioritized as being dominant.

Using this information about access patterns, the developer realizes that time performance can be enhanced. The developer arranges the storage so that the non-inherited attributes are accessed together, but without accessing other attributes in that operation. Vertical splitting groups the non-inherited attributes together, so the developer decides to use it in the target system to address the functional requirements of storing and manipulating the attributes of **Researcher**. This is done by selecting the operationalization **AccessManyTuplesPerRelation**.

The result of using vertical splitting is that the dominant time softgoal is satisfied. However, the space softgoal for attributes is denied. This is because vertical splitting can use extra storage, requiring several entity identifiers per entity. For example, a particular researcher entity can require up to three such entity identifiers (the *entity* key attributes) in relations **AllResearchers**, **AllComputerResearchers** and **AllNumericalAnalysisResearchers** in Figure 9.13.

On the other hand, horizontal splitting requires only one entity identifier per entity (Figure 9.12), so it helps the space softgoal. This operationalization is rejected, however, as it hurts the dominant time softgoal.

The developer can continue consideration of performance requirements at lower layers. Figure 9.20 shows an inter-layer link connecting Layers 4 and 3.

This example has addressed one layer, dealing with **IsA** hierarchies, particularly data hierarchies. However, transaction hierarchies and a variety of other data model features are considered when treating performance requirements in the case studies of Chapters 11 and 12.

9.11 DISCUSSION

This chapter and the previous one have presented the Performance Requirements Framework. It offers developers a systematic approach for dealing with performance requirements, by applying the NFR Framework to performance requirements. This chapter has extended the framework to deal with performance requirements for information system development.

The developer draws on a body of knowledge about performance and information systems, organized in catalogues. This knowledge is used by de-

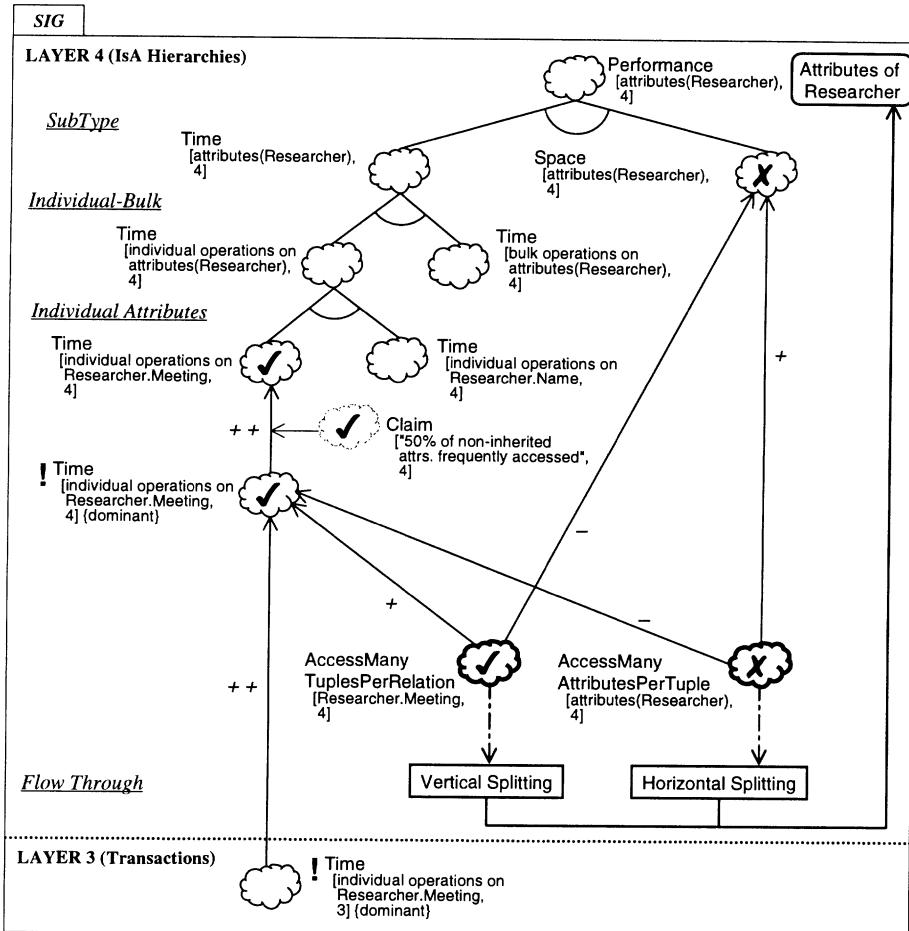


Figure 9.20. Linking another layer to the graph.

velopers, along with their own expertise and domain information, to produce customized solutions that address the performance requirements.

Knowledge of information systems development, their specification, implementation and performance aspects, is represented and organized. This draws on experience in translating the Taxis semantic data model, a conceptual specification language for information systems, into an implementation specified using a relational database implementation language, DBPL. As a result we have catalogued implementation techniques for a number of data model features used in a variety of design specification languages.

The use of C. Smith's principles for building performance into systems helps developers choose the focus of their attention. This prioritization helps a developer deal with tradeoffs.

As well, knowledge is further organized by a layered structure, which can help reduce the number of concepts a developer deals with at a time during the development process. The layering helps the decomposition of problems to simpler, smaller ones. In our layering, based on subsets of specification languages, language features are expressed in terms of simpler features. In addition, the layering is applied to softgoal interdependency graphs. NFRs, which have a global impact, are made more "localized" in their impact as one goes down a graph and operationalizes the NFR softgoals. By going down through layers of graphs, there is further localization of concerns.

Approaches to Treating Performance

We have taken a substantially qualitative approach to dealing with performance requirements. Why have we done this, when performance is generally considered quantitative? In addition, why focus on the quality of the development process, when the performance of the software product is so important?

We feel that qualitative and process-oriented aspects are needed, along with quantitative and product-oriented ones, but different aspects will be stressed at different stages of development. We feel that a qualitative approach is helpful, at least during early phases of development, and when considering combinations of performance and other non-functional requirements (e.g., security) which may be more qualitative in nature.

At the *early* stages of development, it seems helpful to start *qualitatively*. We start with *brief*, high-level requirements. We want to deal especially with broad overall *relationships* and conflicts among NFRs, and between NFRs and implementation techniques. Initial efforts may focus on direction, e.g., moving towards or away from a requirement (e.g., response time should be fast).

At *late* stages of development, a *quantitative* approach will certainly be needed. There, the detail can be analysed with an appropriate degree of quantitative detail. We can quantitatively evaluate the product, e.g., by predicting its performance, to see if it will meet performance requirements. If it does not (or if it meets current requirements, but does not meet new, more stringent, ones), we can again go through the (initially qualitative) process of selection among alternatives.

During *intermediate* stages of development, we would expect both *qualitative* and *quantitative* approaches to be of help. We could use a qualitative approach to select just a few implementation alternatives to evaluate quantitatively. Thus a qualitative approach could decrease the number of alternatives whose performance would be predicted quantitatively. Quantitative performance prediction gives numerical results to evaluate the software product; the prediction results can also indicate trends, which can help as the qualitative decision-making process continues. Note that the layered approach may help quantitative prediction of the performance of intermediate products. The in-

termediate product can be evaluated at a particular layer, by treating it as the “end” product of a smaller step in development.

An early approach to quantitative, product-oriented prediction of performance for semantic data models is presented in [Nixon91]. It uses the language layering presented in this chapter. Detail needs to be added to produce a full prediction model. In addition, a predictor tool should be prepared, and linked to tools for the NFR Framework. Such a performance predictor would be helpful without needing to modify the current framework. For example, quantitative performance prediction results can be used to provide claims for decisions. Also, a performance predictor would be helpful in estimating the performance of target systems selected at various layers using the framework.

In the longer term, more quantitative aspects could be added to the framework, and linked to its qualitative aspects.

Literature Notes

This chapter and the previous one are based on B. Nixon’s thesis [Nixon97a]. They also draw on other publications [Nixon90, 91] [Mylopoulos92a] [Nixon93, 94a, 97b, 98].

This work on performance requirements for information system development was motivated by earlier work on aspects of information system development: information system specification languages, their implementation and performance. Our particular experience [Nixon83] [Chung84] [Nixon87, 89, 90] [Chung88] considered the compilation of specifications expressed in a semantic data model (e.g., Taxis [Mylopoulos80] [Wong81]) into a relational database programming language (e.g., DBPL). See [Borgida85a, 90b], [Albano85, 89], [Atkinson87], [Hull87] and [Peckham88] for overviews of semantic data models.

As a source specification language, this chapter has used Taxis [Mylopoulos80] and its successors, the Taxis Design Languages, TDL [Borgida90a] [TDL87] and TaxisDL [Borgida93]. Various strategies for implementing Taxis long-term processes are discussed in [Chung88]. See [Chung84, 88] and [Nixon90] for details of several compilation techniques for reducing run-time costs for integrity constraint enforcement.

The presentation of the space of alternatives (previously described in [Nixon90] which discussed intergrations issues for implementations) is organized here and in [Nixon97a] by the language layering. Our layered organization of performance issues is somewhat driven by data model features. It is noteworthy that object-oriented database languages can be placed in a comparable hierarchy, starting with object identity, then adding inheritance, and then encapsulation [Cruz90]. Besides structuring selection among implementation alternatives for the framework, the layering is also helpful for structuring performance prediction for semantic data models (an initial approach is presented in [Nixon91]), by adding detailed inputs and outputs at each layer.

The organization of methods draws on [Nixon94a].

The research management example is based on a common example [Olle82] used in the DAIDA project [Borgida89].

Information System Development.

The NFR Framework was influenced by the DAIDA environment for information system development [Jarke92a, 93b] [Borgida90a]. The DAIDA project also addresses implementation of conceptual designs, but focusses on correctness rather than performance. It also deals with the “mapping” of conceptual designs to database implementations. No mapping steps are generated automatically; rather, there is an emphasis on helping the developer establish the *correctness* of refinement steps. This offers great flexibility in considering new implementation techniques. For the future, one can envision an hybrid implementation assistant, which would address the complementary issues of performance and correctness.

A framework for performance engineering for information systems is presented in [Opdahl92]. Focussing on the prediction and improvement of performance of information systems during development, it incorporates models of software, hardware, the organization and the applications. It offers sensitivity analysis and a number of tools, and is integrated into an environment for a larger performance evaluation project [Brataas92]. As part of the same project, a method for designing workflow systems which satisfy performance requirements has been produced [Brataas96]. This project draws on, and extends the work of C. Smith. Opdahl [Opdahl94] argues for quantifying performance demands during requirements specification.

A knowledge-based approach for software performance and capacity planning engineering is proposed in [Molnár93] for the case of a particular systems analysis and design methodology.

It is interesting to contrast the treatment offered in the Framework with other research based on the transformational approach, such as the TI system [Balzer85]. TI, like its transformation-based peers, focusses on correctness requirements, i.e., making sure that the generated implementation is consistent with the original specification. Performance, if treated at all, is treated as a selection criterion among alternative transformations. Kant’s early work [Kant79, 81, 83], on the other hand, does address performance requirements. Her framework, however, focusses on conventional programming-in-the-small rather than information system development, relies on quantitative performance measures (which are available for her chosen domain but are, unfortunately, not always available for information systems because of their complexity) and assumes an automatic programming setting rather than the semi-automatic, dialectical software development process adopted here.

Also in the automatic programming area is Tan’s design assistant [Tan89]. Given a high-level design specification, it aids the programmer with automatic selection of correct implementations using knowledge of algorithms and data structures. This is part of a multi-layer software apprentice project. The Requirements Assistant [Reubenstein91] is at the top, the Design Assistant is

in the middle, and the Programmer's Apprentice (which has KBEmacs [Waters85] as one realization) is at the bottom.

Implementation Techniques for Data and Knowledge Bases.

For automating physical database design, for relational and post-relational databases, Rozen [Rozen93] offers a framework and a tool. Sets of design options which do not conflict with each other are considered, and low-cost designs are found by an approximation algorithm.

Batory [Batory88] addresses the implementation of relational databases. Batory decomposes implementation issues into a set of distinct issues, such as implementation of indexing. Higher-level decisions are made by the developer; the system then handles lower-level decisions. Each decision (building block) has a set of pre-defined implementation alternatives, and should fit together with solutions for other issues. By combining “plug-compatible” solutions, the resulting implementation should be *correct* by construction.

For knowledge base management systems [Brodie86], implementation techniques have been considered [Mylopoulos96] for such issues as query processing [Jarke89], concurrency control [Chaudhri92], constraints, and storage management.

For an object-oriented database with inheritance hierarchies, [Benzaken90] considers placement of entity attributes and functions on secondary storage. Clustering of entities on secondary storage is used to decrease input-output activity for data-intensive applications. Benzaken's work addresses performance-oriented prediction and selection, for a particular problem.

For the particular issue of maintaining integrity constraints, one approach is to automatically repair inconsistent database states via production rules [Ceri90]; the system determines which operations can cause a constraint violation.

III Case Studies and Applications

10 INTRODUCTION TO THE STUDIES AND APPLICATIONS

In Part III, we look at particular case studies and applications of the NFR Framework, having already presented the Framework and its specializations for particular types of NFRs.

We will now study the use of the NFR Framework for a variety of information systems and domains. We present two such case studies, for credit card and administrative systems, which address non-functional requirements for security, performance and accuracy.

We will also present applications of the NFR Framework, to two particular areas of application, software architecture, and business process redesign.

Chapter 11 is the first of two studies of NFRs for information systems. The chapter studies credit card systems with respect to the NFRs of performance, accuracy and security. Domain information, workload and information flow are considered. A variety of data model features are dealt with.

In Chapter 12, an administrative system is studied. It deals with income tax appeals, a government system involving long-term, consultative processes. Performance requirements are considered, using descriptions of the organization, its workload and procedures.

Chapter 13 is the first of two applications of the NFR Framework. The chapter applies the Framework to a particular phase in Software Engineering, namely, software architectural design. Using a standard problem in software architecture, we describe how to use the NFR Framework to systematically guide a software architect in selecting among architectural alternatives. We show how

NFRs such as modifiability, performance, reusability, comprehensibility and security, can be addressed early in the software lifecycle, and reflected in a software architecture before a commitment is made to a specific implementation.

Chapter 14 applies the NFR Framework's concept of softgoal to modelling and reasoning about organizations. This links organizational objectives and strategies with the work processes and technologies that support them. In redesigning an organization, many of the design objectives, such as better customer service and faster turn-around have tradeoffs and can be modelled as softgoals to be satisfied. This is an application of the NFR Framework not limited to the design and development of software systems. The application is illustrated using an hypothetical example from the literature.

In Chapter 15, the NFR Framework and some of our studies are evaluated. Feedback is presented from the viewpoint of domain experts who are familiar with the kinds of systems and organizations studied, but were not involved in the development of the Framework. Methodological considerations for studies are also discussed.

10.1 INTRODUCTION

This chapter introduces the two studies and two applications of the NFR Framework, which are presented in Part III.

The *studies* show the use of the NFR Framework to address NFRs for software systems. We address the NFRs of security, performance and accuracy, which were presented in detail in Part II. We study two information systems, one for credit cards authorization, the other for income tax appeals.

The *applications* show the use of the NFR Framework in two particular areas of application. The NFR Framework is applied to software architectural design, and to business process redesign. The applications are illustrated for standard problems. They address a variety of NFRs, some of which have not been analyzed using the NFR Framework at the level of detail presented in Part II.

Studies

We have presented the NFR Framework and its specializations for particular types of NFRs. Now we turn to putting the Framework to use, by studying its use in dealing with NFRs during development of a variety of systems.

To consider the applicability of the NFR Framework to a *variety* of types of information systems, we conducted empirical studies of portions of some different information systems. These studies address a variety of domains: a credit card authorization system (Chapter 11, based on [Nixon93, 97a] [Chung93a,b, 95b]), and a governmental administrative system for income tax appeals (Chapter 12, based on [Nixon94a, 97a] [Chung95b]). The studies dealt with three important classes of NFRs, namely *accuracy*, *security*, and *performance*. The use of systems which do not meet these NFRs may lead to loss of money and trust,

incorrect treatment, and inefficient administration. The studies also exhibit a variety of organizational workloads, priorities, and other characteristics.

Sections 10.2 through 10.4 outline the characteristics of the domains studied, describe how we conducted the studies, and introduce feedback from domain experts. Feedback is discussed further in Chapter 15.

Applications

The NFR Framework is applied to two application domains.

Chapter 13 applies the NFR Framework to architectural software design. That chapter (based on [Chung95c,d]) applies the NFR Framework to a particular phase in Software Engineering, namely *software architectural design*. It describes how to use the NFR Framework to systematically guide a software architect in selecting among architectural alternatives. This provides goal-driven, process-oriented architectural design. The approach is illustrated with a standard example from the software architecture field, and is applied to a variety of NFRs, including modifiability, performance, security and comprehensibility.

Chapter 14 applies the NFR Framework to business process reengineering and organization modelling. The chapter, based on [Yu94c] [Mylopoulos97], helps modellers understand how an organization operates, and help redesign organizational processes. The chapter links organizational objectives and strategies to the work processes and the technologies that support them. One important way of characterizing an organization is in terms of the goal-oriented behaviour of organizational “actors.” The chapter incorporates the NFR Framework’s concept of softgoal as a knowledge representation construct to enrich the expressiveness of modelling and reasoning about organizational relationships. The approach is illustrated with an hypothetical example from the literature. In redesigning an organization, many of the design objectives, such as better customer service and faster turn-around, can be represented as softgoals that need to be traded-off against each other and ultimately satisfied.

10.2 CHARACTERISTICS OF DOMAINS STUDIED

In Chapters 11 and 12, we present studies of two kinds of systems (See Figure 10.1 for a summary):

- a *credit card authorization system*, [Nixon93, 97a] [Chung93a,b, 95b] which authorizes transactions and cancels stolen cards. Security, performance and accuracy are all important NFRs for these systems.
- an *administrative system* for government: an *income tax appeals system*, [Nixon94a, 97a] [Chung95b] which tracks the progress of tax appeal cases. Performance requirements are considered.

In addition, our studies of other kinds of systems are reported elsewhere (See Figure 10.2 for a summary).:

- another *administrative system* for government: a *Cabinet document management system*, [Chung93a, 95b] which tracks the progress of documents

Study	Credit Cards Chapter 11 [Nixon93, 97a] [Chung93a,b, 95b]	Administration: Tax Appeals Chapter 12 [Nixon94a, 97a] [Chung95b]
Domain	Commercial	Government
<i>Characteristics</i>	interactive	long-term, consultative
<i>Functional Requirements</i>	— authorization — cancellation	— maintain status — statistics
NFRs Considered	time, space, security, accuracy	management time, space, (actually accuracy)
<i>Priorities</i>	— cancellation — authorization	— timely reminders — fast access to info
<i>Tradeoffs</i>	— time-space	— time-space — (actual: time-accuracy)
Organizational Workload:	— very many short transactions — many cardholders	— long-term processes — fairly few dossiers
<i>Data Model Features Considered</i>	IsA hierarchies, transactions, attributes	long-term processes, integrity constraints, attributes
Matters Illustrated	— hybrid methods for priorities & tradeoffs — use tool to build security SIG — information flow — centring principle (order of operations)	— deal with (assumed) priorities & tradeoffs — use tool to represent domain info — centring principle (focus on priorities)
Domain Documents	— annual reports — statistical summary	— operations policy manual — detailed statistics
Feedback	initial interview only	full interview

Figure 10.1. Overview of studies presented in Part III.

Study	Administration: Cabinet Documents [Chung93a, 95b]	Health Insurance [Chung93a, 95b]	Bank Loans [Chung95a, 96]
Domain	Government	Multi-sector	Commercial
<i>Characteristics</i>	long-term, consultative	batch	batch, interactive, consultative
<i>Functional Requirements</i>	— monitor progress	— calculate payments — statements	— calculate rates & balances — statements
NFRs Considered	confidentiality, informativeness	confidentiality (accuracy, user-friendliness)	informativeness, time, accuracy, confidentiality
<i>Priorities</i>	— confidentiality	— confidentiality	— change rate — enhanced info.
<i>Tradeoffs</i>	— security-time	— time-security	— time-accuracy — replicated data vs. central storage
Organizational Workload	— long-term processes — fairly few files	— large volume — many patients	— interest & balance calculations — many statements
<i>Data Model Features Considered</i>	long-term processes, constraints	transactions, attributes	transactions, attributes
Matters Illustrated	— detect correlations — deal with priorities	— complex specification — disambiguate requirements	— dealing with change — centring principle (focus on priorities)
Domain Documents	— proposed sys. specification	— govt. reports — system docs.	— policy manuals — annual reports
Feedback	full interview	full interview	none

Figure 10.2. Overview of studies presented elsewhere.

considered by a government Cabinet. Security requirements are especially important for this kind of system.

- a *health insurance system* [Chung93a, 95b], which maintains information on patients, doctors and reimbursements. Interestingly, this deals with several sectors (professional, governmental and commercial). Security is a major concern.
- a *bank loan system* [Chung95a, 96], which maintains information on loans outstanding and interest rates. This study also dealt with *changes* in non-functional requirements, workload, etc.

The systems, studied in Part III and elsewhere, exhibit a variety of characteristics (See Figures 10.1 and 10.2 for summaries). They include commercial, governmental and multi-sectoral domains. Operations may be short-term and interactive, long-term and consultative, or a combination of these. The studies address a variety of NFRs, especially accuracy, security and performance. Priorities (critical requirements, and dominant parts of the workload) and tradeoffs among requirements vary from domain to domain.

In presenting the two studies (Figure 10.1) in Chapters 11 and 12, we describe the domain, the functional requirements, organizational characteristics (such as workload and information flow), and non-functional requirements.

Domain information was drawn from documents which varied in their nature and degree of detail. They included system specification documents, policy manuals, statistical reports, and annual reports. For each study, we had access to only some of the document types.

Where possible, information is taken from documents from the organizations concerned. We tried to be faithful to the real requirements and workload statistics in such documentation. In some cases we supplemented the documentation with creative imagination, based on our understanding of the domain.

With this information, the studies then illustrate how a developer could deal with NFRs by using the NFR Framework and its components to build softgoal interdependency graphs (SIGs) to record the analysis, rationale and development decisions that might be made in developing such systems.

We used this information to conduct the studies, and found that a number of matters were illustrated. We were able to use several different types of refinement methods and operationalization methods, thus illustrating the NFR Framework's components, and the use of a selection of its associated method catalogues. We were able to detect and deal with defects, such as ambiguities, omissions and conflicts. Interactions among NFRs and development decisions were recorded and handled. We were able to use knowledge of the organizations' workload to deal with priorities and tradeoffs. In the case of performance requirements, we were able to use principles for building performance into systems [C. Smith90], and were able to organize the process using layered structures.

10.3 OUR APPROACH TO CONDUCTING THE STUDIES

We now describe how we carried out the studies using the NFR Framework. This approach (from [Chung95b] and [Nixon97a]) could be used when employing the NFR Framework in a particular domain.

Before presenting the steps in detail, we give an outline.

Step A: Knowledge acquisition.

- **Step A.1: Acquiring knowledge specific to NFRs.**
- **Step A.2: Acquiring domain knowledge.**

Step B: Using of the NFR Framework:

- **Step B.1: Identifying NFR-related concepts.**
 1. *Identifying important NFR softgoals.*
 2. *Identifying development (design, implementation) methods.*
 3. *Identifying design rationale (arguments).*
- **Step B.2: Refining and interrelating NFR concepts.**
 1. *Refining and relating softgoals.*
 2. *Identifying and dealing with priorities.*
 3. *Providing design rationale.*
 4. *Evaluating softgoal achievement.*

These are not necessarily sequential steps, and one may also need to iterate over them many times during the design process. A developer may choose refinements, having operationalizations in mind; thus the development process may move up and down, rather than being strictly top-down.

In more detail, here are the steps used to conduct the studies.

Step A: Knowledge acquisition.

Acquisition of knowledge about particular NFRs and the domain was done in this Step.

- **Step A.1: Acquiring knowledge specific to NFRs.**

First we obtained knowledge about the particular type of NFR. Our sources of knowledge included industrial experience and academic literature. We encoded and catalogued this knowledge using the NFR Framework and its components. We produced a terminology for particular NFRs, such as the Accuracy Type and the Performance Type. We also developed catalogues of methods for information system development, as well as methods for operationalizing NFRs. Tradeoffs and interactions among methods and requirements were noted, e.g., as correlation rules. Typical design rationale were represented as generic arguments.

After this Step was done essentially once for a particular NFR, the catalogued results were applied to several domains.

- **Step A.2: Acquiring domain knowledge.**

For each domain, we obtained documents from (or about) the organizations whose systems we were studying. As noted earlier, the nature of the documents varied depending on the organization. In one study we obtained workload information, and information on the organization and its operations (including partial schema information). In some studies, we received only general, public information (e.g., annual reports). We then conducted an initial review of the documents, to understand the domain characteristics, obtain as much NFR-related information as possible, and get an initial idea of some of the main requirements. We tried to make minimal assumptions, and be faithful to the source documents, which varied in the amount of detail provided. Due to the varying amounts of information available to us, we sometimes had to provide or augment information on schema, requirements, and priorities, based on our own experience and intuition, but without violating available domain knowledge. This Step was done once per domain, but could be done more interactively, using ongoing contact with the organizations over a period of time.

Step B: Using of the NFR Framework:

Detailed development via use of the NFR Framework involved iterative use of this Step.

- **Step B.1: Identifying NFR-related concepts.**

This Step took domain information and arranged it into categories along the lines of the main components of the NFR Framework.

1. *Identifying important NFR softgoals.*

We came up with an initial estimation of priorities for the organization and system. This included critical NFRs and dominant parts of the workload.

2. *Identifying development (design, implementation) methods.*

These included methods from the area of concern, such as accuracy or performance (from Step A.1), and those described in domain documents. These were then used as operationalization methods.

3. *Identifying design rationale (arguments).*

We identified and catalogued the relevant arguments. These too could originate from generic knowledge of systems, particular NFRs, and the type of system (e.g., information systems), or from domain-specific information.

- **Step B.2: Refining and interrelating NFR concepts.**

In this Step we interrelated NFR softgoals, refining softgoals, handling priorities, providing design rationale, and evaluating achievement of softgoals. To do this, we used and linked the concepts identified in

Step B.1, producing Softgoal Interdependency Graphs, while endeavouring to meet the stated NFR softgoals during the development of a system.

1. Refining and relating softgoals.

- (1.a) We refined softgoals, often into simpler ones. We often selected methods from catalogues. Catalogues were extended to address new methods.
- (1.b) We also refined softgoals to help clarify the meaning of softgoals, which may be initially stated briefly, and may be ambiguous. For example, sometimes softgoals had to be refined before the relationship among them could be established.
- (1.c) We related development (design, implementation) methods to NFR softgoals.
- (1.d) We identified tradeoffs, interdependencies among softgoals, conflict and synergy, and factors underlying correlations.

2. Identifying and dealing with priorities.

- (2.a) This involved identifying priority softgoals, and critical and dominant parts of the workload.
- (2.b) Tradeoffs were often handled by considering tradeoffs. For performance softgoals, principles for providing good response time [C. Smith90] were used.
- (2.c) We noted the impact of those methods which were stressed in the domain documents.

3. Providing design rationale.

- (3.a) Rationale for decisions was obtained from a number of sources, including workload descriptions, other documentation, catalogues of generic rationale, and arguments based on observations about the particular system.
- (3.b) On the basis of development tradeoffs present in the SIG under construction, we provided some reasons why particular methods were chosen.

4. Evaluating softgoal achievement.

We applied the evaluation procedure to determine whether the stated NFRs, particularly the priority ones, were achieved. Part of the procedure involves examination of the nature and strength of relationships between softgoals.

Step B.2, and sometimes Step B.1, were applied repeatedly, not necessarily in the order stated.

The studies dealt with only small portions of systems. They were based on descriptions of existing systems, and in some cases considered a proposed extension to an existing system.

The studies were carried out “off-line,” without our involvement as actual participants during the development process. We ourselves did not develop the systems, but did consider NFRs for such systems.

We had contact with the organizations at the start of the studies, when we obtained their documentation, and then at the end, when we obtained feedback. During the actual studies, we examined their documentation, but generally did not communicate with the organizations. Thus our approach was done in a kind of “archaeology mode” [Chung95b] in which we attempted to “dig” into the documents producing SIGs without being in contact with the organizations.

10.4 OBSERVATIONS FROM STUDIES

Feedback from people familiar with these kinds of systems (Chapter 15) provides a valuable “reality check” in examining the NFR Framework and its applicability to a variety of systems. Some of the studies were reviewed by domain experts. They offered some preliminary support for the usefulness of certain aspects of the framework, while pointing out room for improvement, and raising some important open issues.

The studies were based on the domains, and illustrated our approach. However, the experts found that the detailed conclusions in the studies are not necessarily what they would have concluded. This seems mainly due to our use of source documents without ongoing contact with domain experts during our study.

For example, in some studies, we were able to deal with tradeoffs, by using domain information (e.g., workload) and hybrid combinations of implementation methods. This allowed us to meet priority requirements, at the expense of non-priority requirements. In some studies, however, experts noted that tradeoffs did not reflect priorities in the domains, hence indicating the need for more domain knowledge. In retrospect, we would have done studies more interactively. This was a shortcoming in the study methodology rather than of the Framework. See Chapter 15 for details of feedback.

10.5 LITERATURE NOTES

This chapter is based on portions of [Chung95b], [Nixon97a] and [Chung93a]. The detailed sources of individual studies and applications are given in the literature notes of each chapter in Part III.

11 A CREDIT CARD SYSTEM

In this chapter, credit card systems are studied. We consider an information system for a bank's credit card operation. A body of information on cardholders and merchants is maintained. In this highly competitive market, it is important to provide fast response time and accuracy for sales authorizations. To reduce losses due to fraud, lost and stolen cards must be invalidated as soon as the bank is notified.

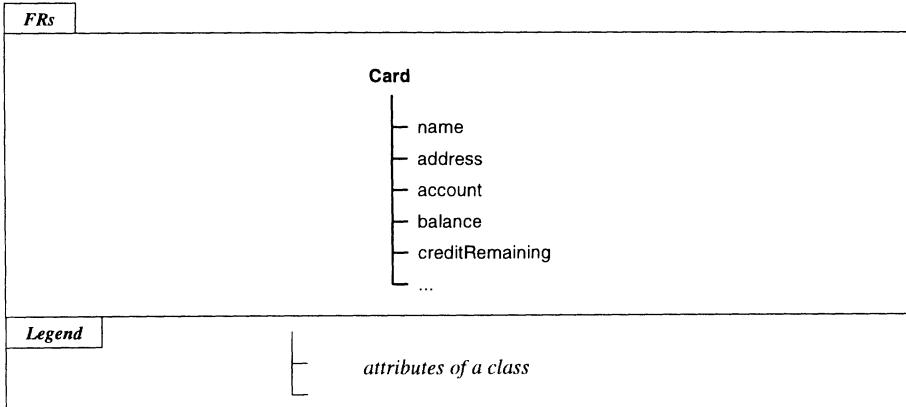
Interestingly, this chapter considers several of the NFRs we have studied — performance, security and accuracy requirements — for one system. Domain information and organizational workload are considered.

We describe the domain and requirements, and then develop softgoal interdependency graphs (SIGs) to address NFRs.

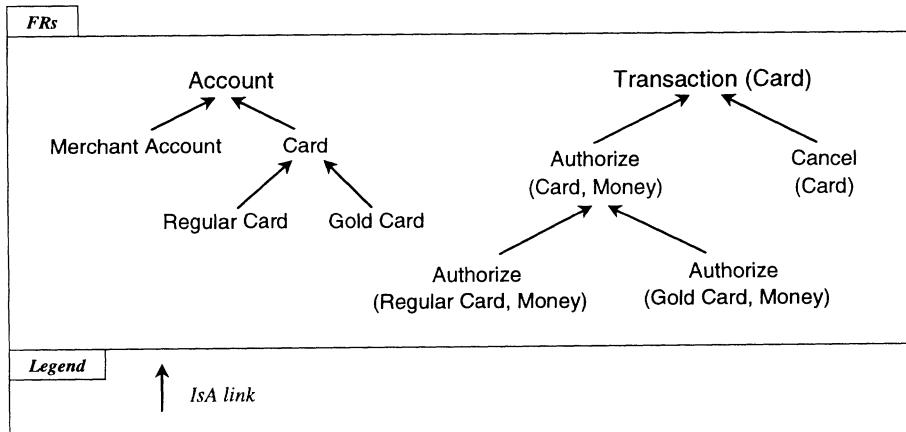
11.1 DOMAIN DESCRIPTION AND FUNCTIONAL REQUIREMENTS

The functionality of a bank's credit card system includes maintaining information on sales, cardholders and merchants. Transactions are authorized, and accounts are updated. Stolen cards are cancelled.

This study draws on actual statistics [Canadian Bankers91] and annual reports from industry associations [Visa Canada90] [Visa International91] [MasterCard91]. In conducting studies, we have tried to remain faithful to the information available. In this study, where information was unavailable to us, we supplemented the published information with some creative imagination

**Figure 11.1.** Attributes of credit cardholders.

based on general knowledge of the domain. From the statistics for Visa and MasterCard usage in Canada for the year ended 31 October 1991 [Canadian Bankers91], we estimate workload for an hypothetical large bank. Assuming

**Figure 11.2.** Classes in the credit card system.

that the 5 large banks have approximately 75% of the market, we allocate 15% of the national market to our hypothetical bank. We further assume that

there are approximately 300 retail sales days per year. Accordingly, a bank would have 3 600 000 cards in circulation and 90 000 merchants. Each day it would deal with 217 000 sales slips, 218 lost or stolen cards, and 19 cards used fraudulently.

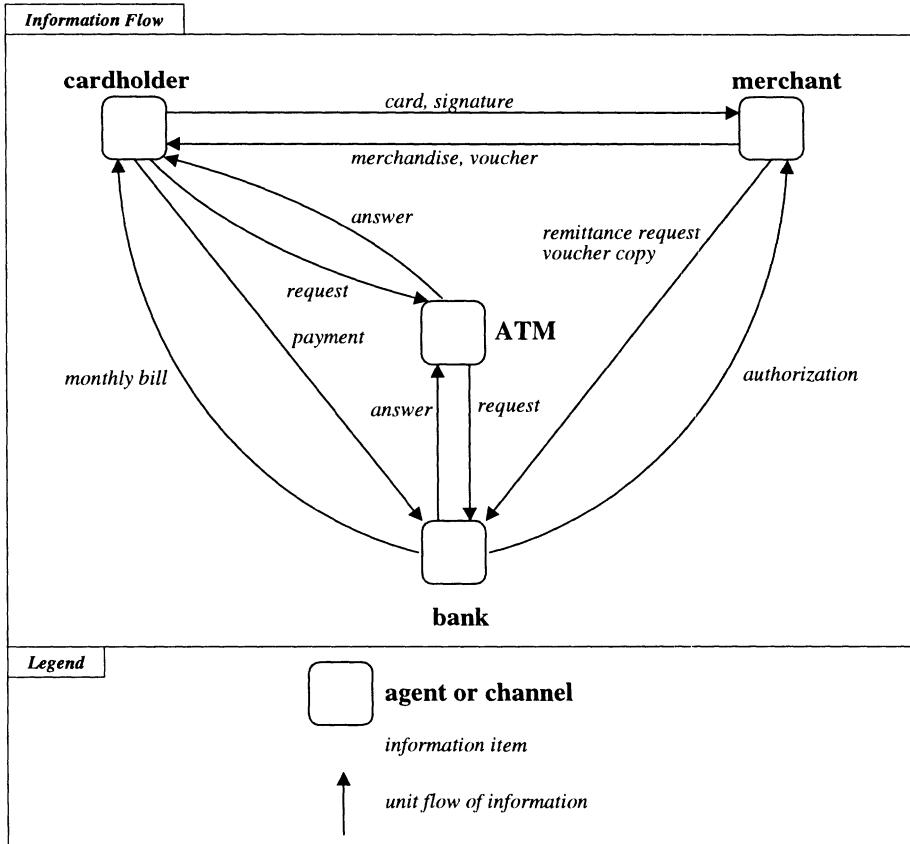


Figure 11.3. Information Flow for the credit card system.

A credit card system specification would include a design schema. Based on general knowledge of the domain, we assume that each cardholder has a number of attributes, as shown in Figure 11.1.

In addition, the market may be somewhat segmented. For example, cards may be categorized as “regular” or “gold” (Figure 11.2). This segmentation can also be reflected in activities. For example, credit authorization for gold cards may be more flexible and also involve additional steps (e.g., calculation

of travel bonuses payable to cardholders based on the amount of the sale). This is reflected in the figure showing part of an hypothetical schema, by the use of IsA (inheritance) hierarchies for data and transactions.

Figure 11.3 shows a simplified description of the *information flow* for the credit card system. Information flows between the bank, cardholders, merchants and automated teller machines (ATMs).

11.2 NON-FUNCTIONAL REQUIREMENTS

We examined [Nixon93, 97a] [Chung93b, 95a] some organizations' documents [Canadian Bankers91] [Visa Canada90] [Visa International91] [MasterCard91]. Response time and security were considered important NFRs. For a commercial credit card system, quality is very important, as the market is highly competitive.

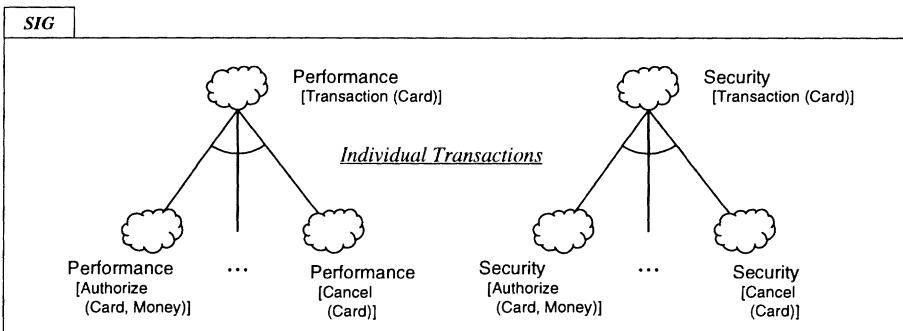


Figure 11.4. Some main NFRs for the credit card system.

We now consider some hypothetical non-functional requirements which a developer might state for this system. This is done on the basis of the above domain information, and an everyday knowledge of the domain.

Figure 11.4 shows some top-level NFRs for performance and security for card transactions. These are refined into NFRs for specific transactions, such as authorizing a sale for a particular card and money amount (*Authorize(Card, Money)*), and cancelling a card (*Cancel(Card)*). For these broad softgoals, the performance softgoals do not have a layer topic specified. Some additional broad softgoals are not shown in the figure. Layers will be introduced for offspring of these broad softgoals.

For performance requirements, time performance is an important NFR for credit cards, due to the high transaction volume, and the need to cancel stolen cards immediately. We considered the following performance requirements.

1. The cancellation of lost and stolen cards requires very fast response time. To minimize financial loss, this transaction is given highest priority.
2. The second performance requirement is that authorization of sales, a dominant portion of the transaction volume, should be done with fast response time. In practice, this kind of performance is considered important; for example one credit card network reported that it has an international authorization average response time of 1.9 seconds [Visa Canada90].
3. Also considered was minimizing secondary storage requirements for the cardholder information, due to the large number of cards in circulation.
4. In addition, minimizing secondary storage for the daily individual sales is another, less important, requirement.

For security requirements, we require good security for authorizing sales.

This leads us to consider more specific security requirements for authorizations — addressing integrity, confidentiality and availability.

Accuracy is an important aspect of integrity. As account information can change rapidly, it is important that cardholder accounts have information that is both accurate and timely. Here timely accuracy is a critical NFR for authorization of sales.

11.3 DEALING WITH PERFORMANCE REQUIREMENTS

We now illustrate how a developer of the credit card system could address the performance requirements (involving time and space) stated above. We show some softgoal refinement methods and the impact of higher-layer softgoals upon lower ones. We deal with the *prioritization*: of performance requirements, some for *dominant* parts of the workload, and others for *critical* parts. C. Smith [C. Smith90] points out the need to address both, in order to effectively deal with requirements. We consider first a critical time requirement, then a time requirement for a dominant part of the workload, and then some space requirements. We also consider softgoals at different layers, involving various data model features.

In dealing with performance softgoals for this study, we repeatedly followed a pattern of first decomposing a softgoal based on its components, and then identifying and focussing on *priority* component softgoals. This is in keeping with principles for building performance into systems [C. Smith90]. For example, we decomposed a softgoal of good response time for cancelling lost and stolen cards into softgoals for the operational components. We focussed first on accessing the attributes of a credit card, then on a particular attribute which is needed for cancelling a stolen card, and finally on updating it. This was satisfied by updating the attribute at the *start* of the operation, which effectively cancels the card, even though the non-critical tasks may continue in the background.

We considered a *time-space tradeoff*. Space performance softgoals sometimes interacted negatively with the above treatment of time softgoals. For

example, we sought good space performance for storing the attributes of a card. As there are many cards, using a compressed format would save space, but compressing the particular attribute needed for cancellation would cause a conflict by slowing down the cancellation due to processing delays. So we satisfied the space softgoal for the dominant case, by compressing the other attributes. Even though we did not meet a non-dominant storage softgoal, we avoided a negative impact on the critical time softgoal.

Fast Card Cancellation: A Critical Time Softgoal

As soon as the bank is notified that a card has been lost or stolen, the card must be cancelled, to disallow unauthorized purchases. To reduce fraud, this infrequent transaction is treated by a developer as *critical*, and is given top priority for attaining fast response time. While there may be lengthy delays *outside* the system (e.g., the time it takes for people to discover lost cards and report them), the system should cancel the card as soon as possible, to avoid authorizing subsequent fraudulent requests.

For the design, we assume that given a *Card*, the *Cancel(Card)* transaction must update the value of its *Status* attribute, to indicate cancellation of the card. In addition, several other attributes of the card (Figure 11.1) will also be accessed. For example, the cardholder's name, address, etc., could be printed in a log.

Stating an Initial Softgoal.

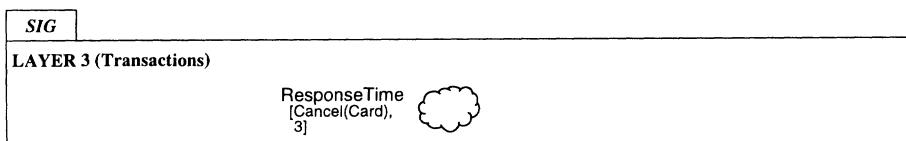


Figure 11.5. Initial softgoal for fast cancellation of credit cards.

The performance softgoal *Performance[Cancel(Card)]* was shown at the bottom of Figure 11.4. Using the *SubType* decomposition method (not shown in a figure), it is refined into softgoals *Time[Cancel(Card)]* and *Space[Cancel(Card)]*. Then *Time[Cancel(Card)]* is refined into softgoals for response time and throughput. So far we have not indicated layers for these broad softgoals.

We now focus on good response time for this transaction and specify a layer. This softgoal is stated at Layer 3, which deals with transactions, and is shown in Figure 11.5

Softgoal Decomposition and Prioritization.

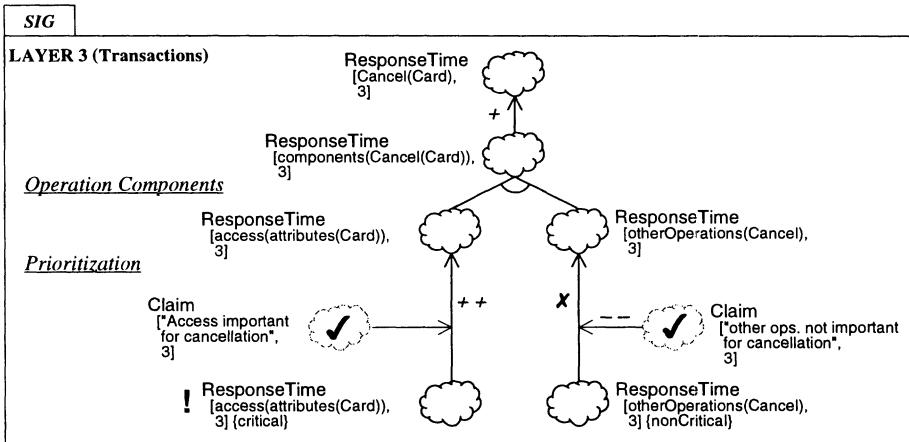


Figure 11.6. Softgoal decomposition and prioritization with argumentation.

In Figure 11.6 the developer focusses on the components of the Cancel transaction, resulting in a softgoal for the components of the operation. The **OperationComponents** method is used to decompose the softgoal, resulting in response-time softgoals:

1. for accessing the attributes of a card, and
2. for other operations.

Now the developer identifies one offspring as **critical**, another as **nonCritical**, and uses domain information to support these actions. Response time for accessing the attributes of the card, **ResponseTime[access(attributes(Card)), 3]**, is identified as **critical**, because the access is important. On the other hand, response time for the other operations are identified as **nonCritical** for cancellation, and are not considered further.

Now the developer will consider the parent softgoal, **ResponseTime [components(Cancel(Card)), 3]**, to be satisfied if its critical offspring is satisfied, even if the nonCritical one is denied. This is done by using the prioritization templates from Figures 4.30 and 4.31 and Section 8.3. This has the effect of “softening” AND contributions.

Note that some relationships among softgoals are not precisely represented by an **AND** contribution. For example, developers may use their expertise to state that certain offspring are more (or less) important than others, for the purposes of satisfying a parent softgoal. In the NFR Framework, this

flexibility can be accomplished by placing *HELPS* contributions below or above an *AND* link. This approach, shown in Figures 4.32 and 4.33, and discussed in Section 8.3, was used in earlier versions of this study and the next. It is omitted from Chapters 11 and 12, to simplify the presentation.

Some *HELPS* links are not discussed. When the contribution of a Claim is not shown, the default is *MAKES*.

Further Decomposition and Prioritization.

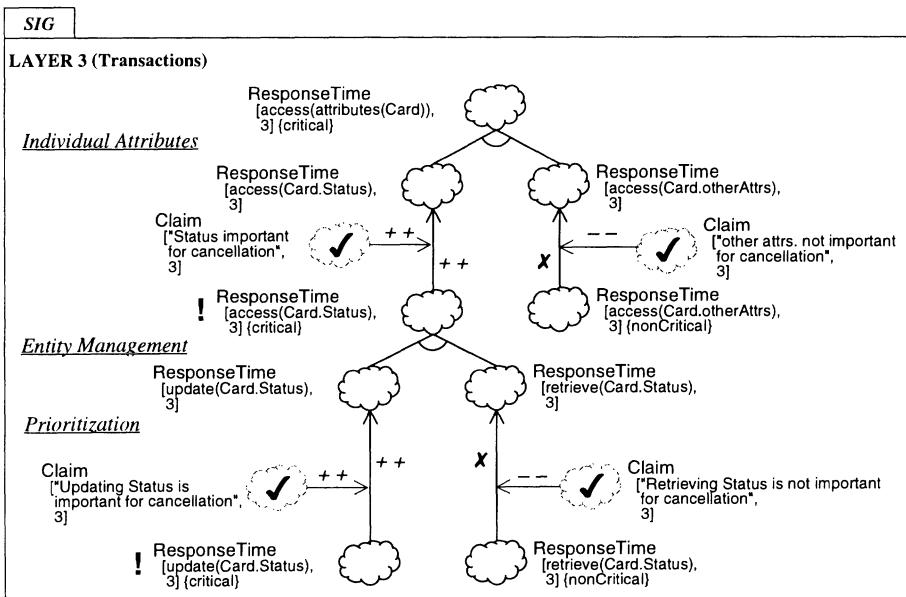


Figure 11.7. Further softgoal decomposition and prioritization.

The **IndividualAttributes** method refines the critical softgoal into softgoals for each of the attributes of **Card**. For presentation, Figure 11.7 groups the attributes into **Status**, whose access is critical, and the remaining non-critical ones, represented by **otherAttrs**.¹ Here the contribution type is *AND*.

Now the most critical part of the **Cancel** transaction is accessing the **Status** attribute, and in particular, updating it. The **EntityManagement** method decomposes the softgoal for accessing **Status** into softgoals for retrieval and updating. Of the two, updating is argued as being critical, since it effectively

¹In Figures 11.16 through 11.18, **otherAttrs** also excludes **CreditRemaining**.

cancels the card, blocking further sales authorizations. A prioritization argumentation method records that access to this attribute is critical.

The other softgoal, retrieval, is not critical here, but is considered further in Figure 11.13.

Consideration and Selection of Operationalizing Softgoals.

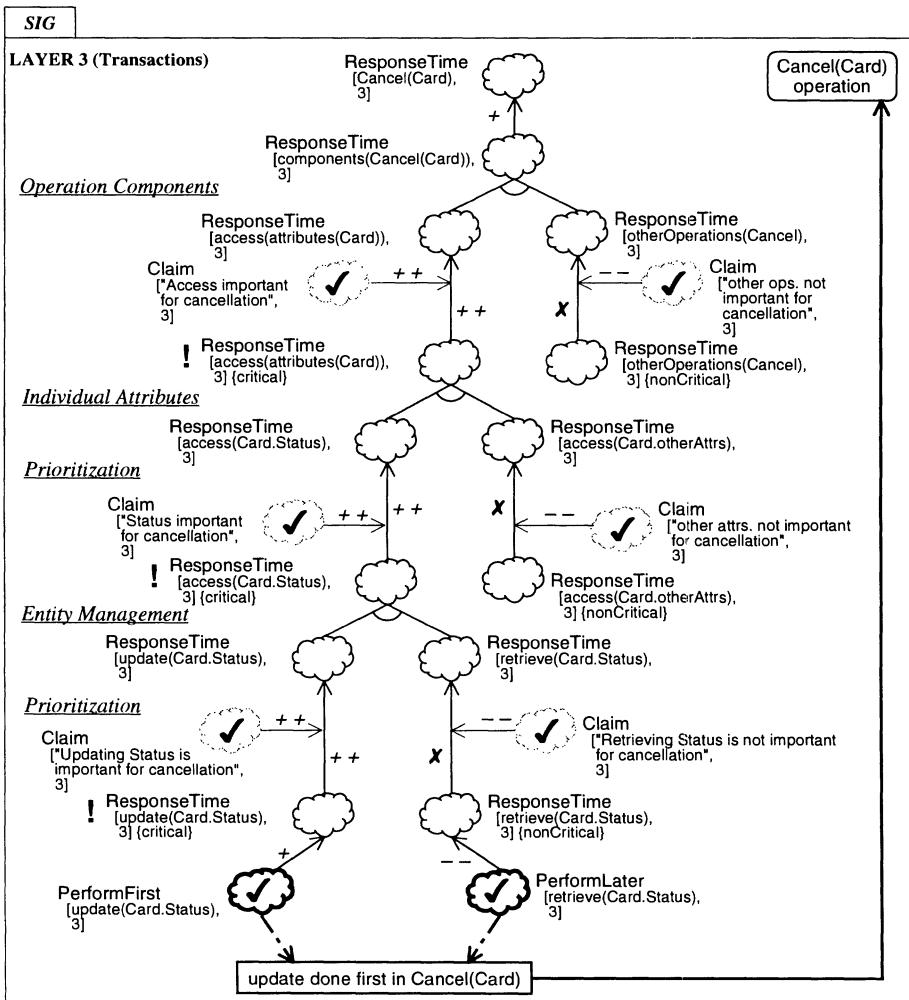


Figure 11.8. Selection of Operationalizing Softgoals.

C. Smith [C. Smith90] advocates achieving good response time as seen by users of the system. Accordingly, a bank officer who requests cancellation of a card can receive confirmation from the system as soon as **Status** has been updated, even though other tasks (such as updating other attributes and printing a log) may continue in the background.

Some design languages (including TaxisDL) do not require specification of the order of performing individual operations within a transaction. In Figure 11.8, the developer can exploit this underspecification in the design by choosing an order of execution of operations in the target implementation language. This leads the developer to select performing this update first in the transaction. This “satisfices” the response-time softgoal, and the **PerformFirst** operationalizing softgoal is recorded.

Figure 11.8 shows the developer for all of Layer 3, which deals with transactions. The selection of operationalizations **PerformFirst[update(Card.Status), 3]** and **PerformLater[retrieve(Card.Status), 3]** is shown at the bottom of the figure, and is indicated by “ \checkmark .” The satisficing of claims is also shown by “ \checkmark .” The right side of the figure also links the target implementation to the functional requirements.

Linking Layers.

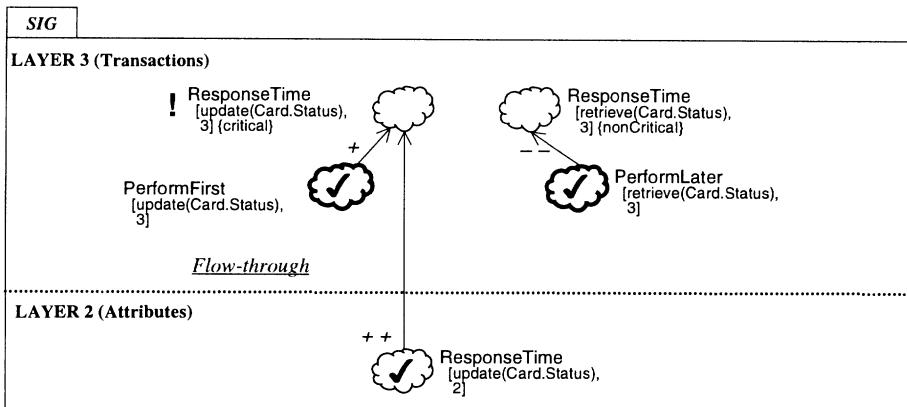


Figure 11.9. Using inter-layer interdependency links.

The developer refines the Layer 3 softgoal for updating **Status** into a softgoal at Layer 2, which deals with attributes. This is done by the **FlowThrough** method. Softgoals at the two layers are connected by an *inter-layer interdependency link*, shown in Figure 11.9.

Decomposition on Implementation Components.

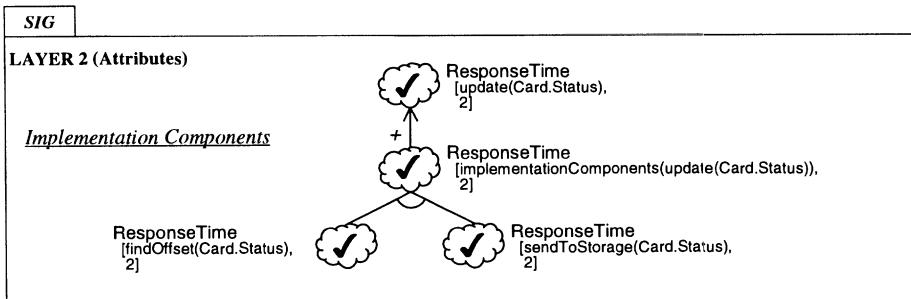


Figure 11.10. Decomposition based on implementation components.

The Layer 2 softgoal is then decomposed into softgoals for the two component operations. These are finding the attribute's offset, and sending the new attribute value to storage. This is done by the `ImplementationComponents` method in Figure 11.10.

Selecting Operationalizing Softgoals.

The softgoal of quickly finding the offset of the attribute within a tuple can be satisfied by statically determining the offset. This is recorded by an operationalizing softgoal. The other softgoal — that of quickly sending the updated attribute value to secondary storage — may be satisfied by storing Status values of all cards together. By storing few other attributes, if any, per tuple, time may be reduced by accessing a smaller structure. Both operationalizing softgoals (bottom of Figure 11.11) *HELP* their parents. As a result of choosing these operationalizations, the target system stores Status values in a separate relation via `SelectiveAttributeGrouping` (a method which does not deal with inheritance of attributes).

Evaluating the Impact of Decisions.

Now that we've constructed the SIG from top to bottom, let's consider evaluating the impact of decisions upon main softgoals. The evaluation procedure (Section 3.3) does this by labelling softgoals, starting at the bottom of Figure 11.11. Initially, all softgoals are labelled as undetermined (U). Selected operationalizing softgoals are labelled as satisfied ("✓"), while rejected ones are denied ("✗"). Generally, *interdependency links* resulting from application of catalogued methods are satisfied (i.e., have "✓" as their labels).

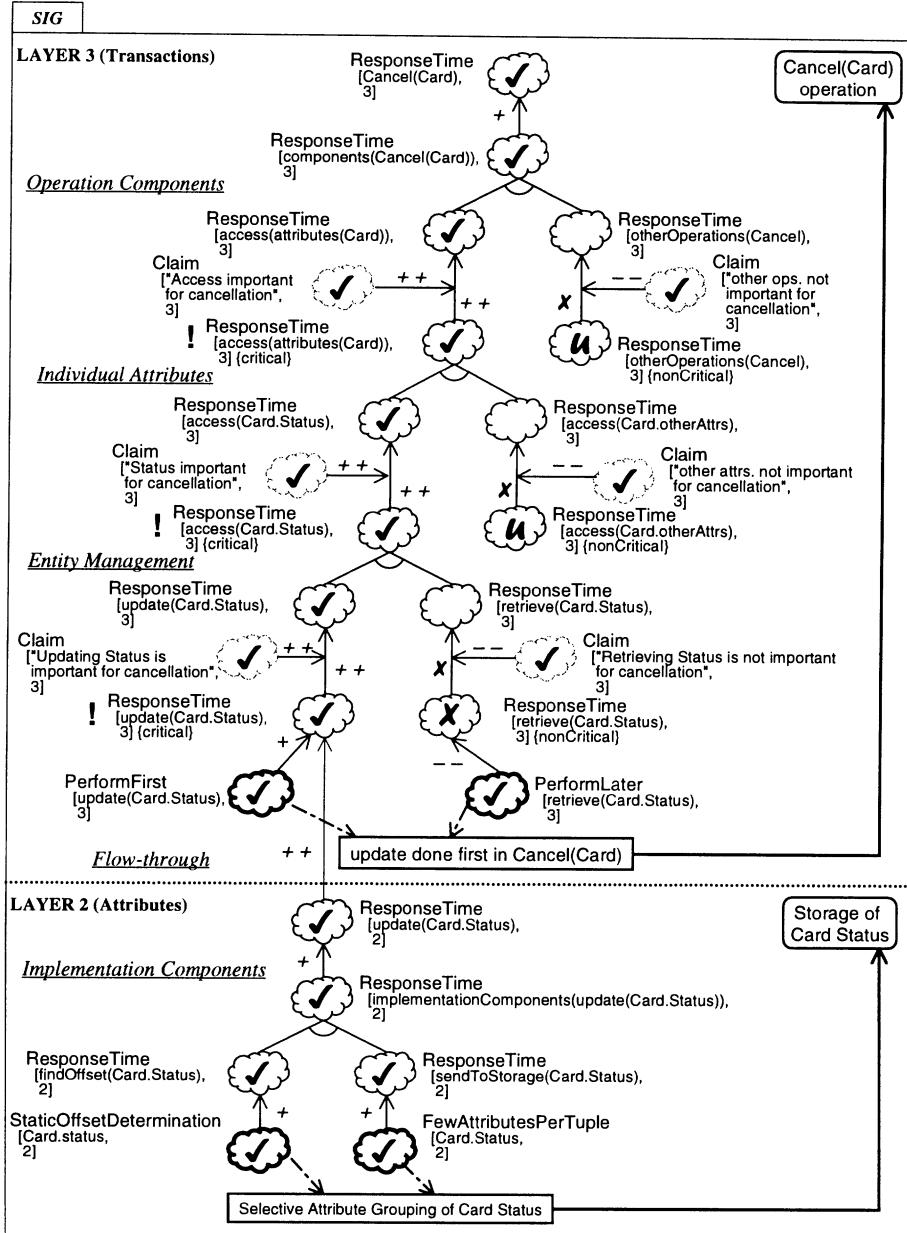


Figure 11.11. Evaluating the impact of decisions after selecting operationalizing softgoals.

Evaluation starts at the bottom of the SIG for the lowest layer, then up through the graphs of each layer. By selecting `SelectiveAttributeGrouping` [`Card.Status`, 2], both operationalizing softgoals `StaticOffsetDetermination` [`Card.Status`, 2] and `FewAttributesPerTuple` [`Card.Status`, 2] are labelled as satisfied (“ \checkmark ”). By the rule for *HELPS* contributions (Table 3.2), each “ \checkmark ” contributes weak positive support (“ W^+ ”) to its parent. The developer, who must change the W^+ values, feels that the softgoal refinements are satisfactory, and changes the W^+ values of `ResponseTime[findOffset(Card.Status), 2]` and `ResponseTime[sendToStorage(Card.Status), 2]` to “ \checkmark ”. Several similar changes from W^+ to “ \checkmark ” are not explicitly mentioned in the remainder of this chapter. Now using the rule for *AND* contributions (Section 3.3), the two “ \checkmark ” values are combined into an “ \checkmark ” value for `ResponseTime[implementationComponents(update(Card.Status)), 2]`, which is propagated to its parent.

Up at Layer 3, `!ResponseTime[update(Card.Status), 3]{critical}` receives an “ \checkmark ” value via an inter-layer *HELPS* link, and an “ \checkmark ” value from `PerformFirst[update(Card.Status), 3]`, resulting in an “ \checkmark ” value for it and its parent.

For `!ResponseTime[access(Card.Status), 3]{critical}`, we encounter a frequently occurring pattern where there is a conjunction of a critical operation and a non-critical one (Recall Figures 4.30 and 4.31 and Section 8.3). The developer argues that one of its offspring, `ResponseTime[retrieve(Card.Status), 3]`, is *not* critical, and a *nonCritical* offspring is produced. The claim denies that parent-offspring *interdependency link*, so that `ResponseTime[retrieve(Card.Status), 3]` is not considered in the conjunction. Now the parent, `!ResponseTime[access(Card.Status), 3]{critical}`, is labelled as satisfied, due to satisfying its critical offspring alone. Other non-critical offspring are similarly eliminated. This same pattern is repeated resulting in satisfying `ResponseTime[access(attributes(Card)), 3]` and then `ResponseTime[components(Cancel(Card)), 3]` even though their non-critical offspring have not been satisfied. Finally, the developer labels the top-level softgoal `ResponseTime[Cancel(Card), 3]` as satisfied.

Importantly, the developer has satisfied the *critical* operations of this dominant transaction. Moreover, response-time softgoals were denied primarily for those operations which can be performed after responding to the user. These results are in keeping with principles for providing good response time [C. Smith90]. When these less-important negative results are aggregated with the very important positive ones, the overall result is that steps have been taken towards achieving good response time for users.

Fast Sales Authorization: A Dominant Time Softgoal

Authorization of sales is a *dominant* part of the workload. The developer desires good response time for the `Authorize(Card, Money)` transaction, whose topics include `Card`, and the amount of `Money` involved in the sale. The *critical* parts of the transaction are (1.) ensuring that the card is valid (i.e., not reported as stolen) and (2.) ensuring that the amount of the proposed sale does not exceed the balance of credit remaining for the card. Other tasks can be done after

authorizing the sale, such as creating a record of the sale, and recording travel bonus points (if applicable to the particular card type).

We will consider operationalizations for the critical and dominant softgoals, and will structure them in a SIG. For reasons discussed below, it happens in this particular case that the dominant softgoal is addressed at Layer 4 (Figure 11.12) and the priority softgoal is addressed at Layer 3 (Figure 11.13).

Dealing with Transaction Hierarchies.

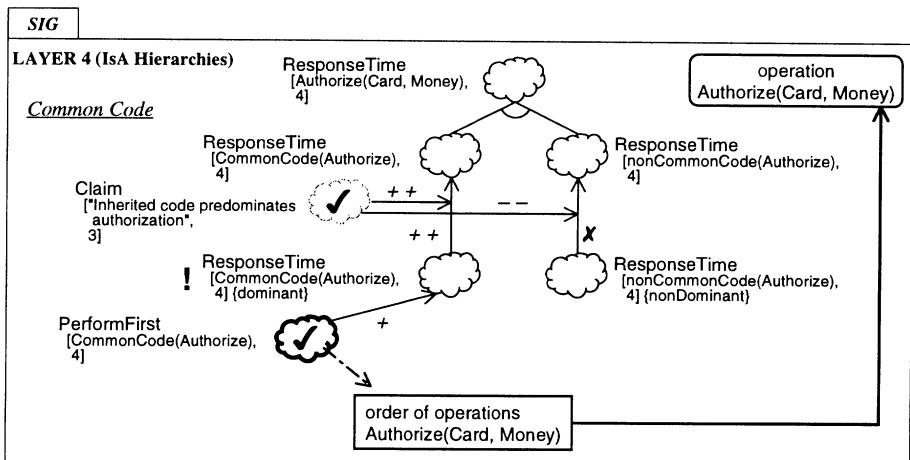


Figure 11.12. Dealing with transaction hierarchies at Layer 4.

Just as card types (regular, gold, etc.) can be arranged in an IsA hierarchy, so can the corresponding versions of the **Authorize(Card, Money)** transaction (Recall Figure 11.2). The specialized transactions can have some additional actions depending on the type of card involved, such as recording travel bonus points for gold cards. Since we are dealing with *IsA hierarchies*, the softgoal for good response time for the transaction (shown at the top of Figure 11.12), is at Layer 4. This softgoal, **ResponseTime[Authorize(Card, Money), 4]** contributes to the broad softgoal **Performance[Authorize(Card, Money)]**, which was shown at the bottom of Figure 11.4.

An important observation is that the *critical* parts of the authorization process are independent of the card type (gold, regular, etc.). That is, authorization is basically independent of the options for travel bonuses, service charges, etc., which vary with the type of card. Thus code can be structured so that operations which are common to all card types are done first, and the

authorization decision is returned to the user *before* the card type is identified, and additional, type-dependent operations, are performed.

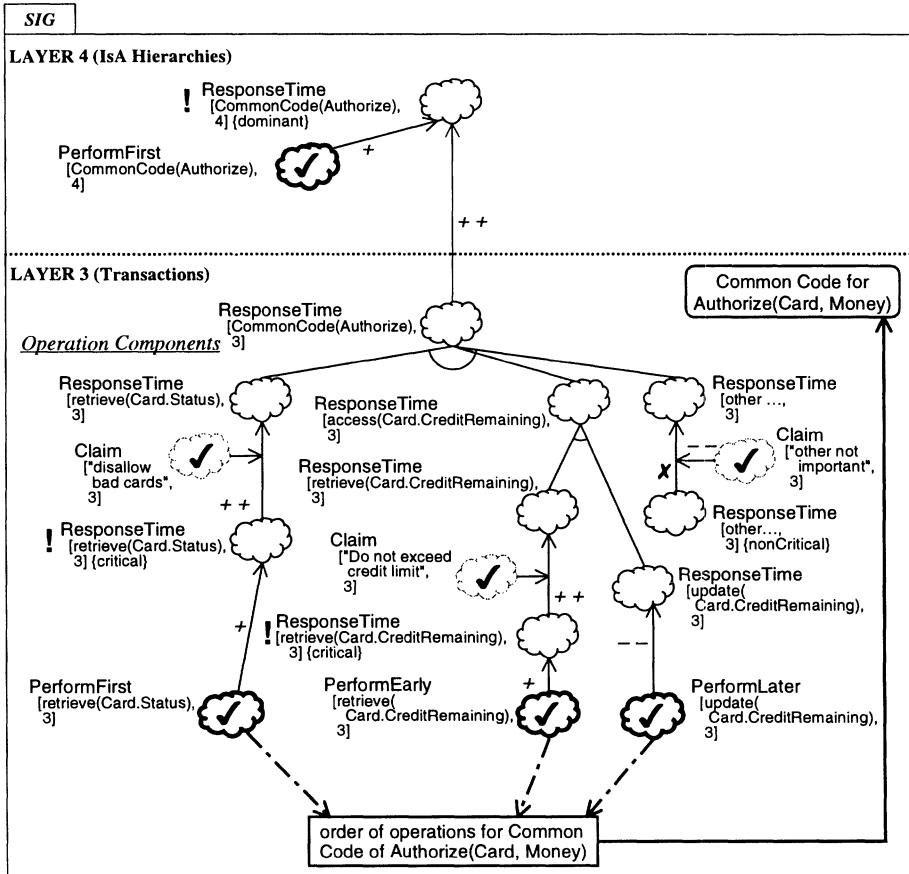


Figure 11.13. Dealing with priority operations at Layer 3.

In Figure 11.12, the developer applies the **CommonCode** (or **CommonOperations**) method, which decomposes the response-time softgoal into Layer 4 softgoals for the operations common to all cards, and for the non-common operations. The common operations are inherited from **Card** by its specializations including **GoldCard** and **RegularCard**. The common operations (inherited code) are identified, in an argument, as a dominant portion of the workload. The softgoal for common operations is satisfied by performing them before other operations. As the common code need not distinguish specializations of **Card**,

the developer will subsequently deal with the code at Layer 3, which deals with transactions without inheritance in Figure 11.13. On the other hand, the “non-common operations,” i.e., those operations which differ for the specializations of **Card**, are considered at Layer 4. There are different techniques available (e.g., [Nixon87]) to deal with issues arising in IsA hierarchies of transactions, such as code inheritance. We do not discuss these further here, as the softgoal is not dominant. Layer 4 (IsA hierarchies) has dealt with the overall relative order of operations of the transaction.

Dealing with Priority Operations.

The developer returns to the common operations, but now at Layer 3 (Figure 11.13). The developer uses the **OperationComponents** method to produce softgoals for the sub-tasks. These include retrieving the status of the card (which is critical here, and is therefore performed first), accessing the credit remaining on the card, and performing other operations. **CreditRemaining**, the credit limit remaining on the account, is the credit limit less the current balance outstanding. The softgoal of fast access to **CreditRemaining** is decomposed by the **EntityManagement** method into softgoals for retrieving the attribute and updating it. The retrieval subgoal is identified as critical and is satisfied by being performed early. Updating the credit remaining and the account balance, can be performed later. At Layer 3, we have dealt with the relative order of execution of the common operations of the transaction.

Note that a softgoal may be critical in one situation, but not in another. For example, recall that in the context of cancellation (Figure 11.7), retrieval of **Card.Status** was not critical. Here, however, it is critical in the context of authorization (Figure 11.13).

Dealing with Credit Card Attributes.

Now Layer 2 deals with representation of attributes (Figure 11.14). So far, we have assumed that **CreditRemaining** is a separate, explicitly-stored attribute. What would happen if it were in fact defined in the design schema as being *derived* (e.g., as **Card.CreditLimit** – **Card.BalanceOwing**)? To avoid having to retrieve and process several attributes during this critical part of a dominant procedure, the developer may decide to replicate the derived attribute and store it separately. This involves a *change to the source schema* (Cf. [Goldman83]) which might normally be beyond the purview of the developer; but is done for the sake of “satisficing” a critical performance softgoal. A **HELPS** link indicates that the replication helps retrieval.

Decisions and their Impact.

As was mentioned for Figure 11.11, the good response-time softgoal for updating the **Status** attribute can be satisfied by storing few attributes per

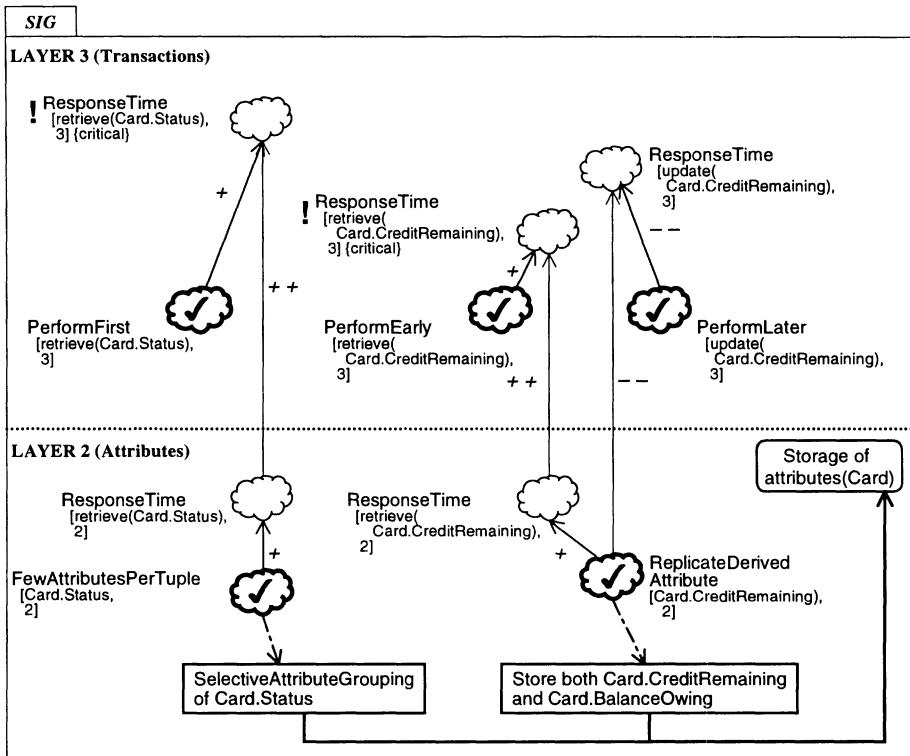


Figure 11.14. Dealing with credit card attributes at Layer 2.

tuple, by using selective attribute grouping. This operationalizing softgoal can be used here for the other critical task — retrieving the Status attribute. The softgoal is labelled as being satisfied (“✓”) as we start the bottom-up evaluation process (Figure 11.15).

Thus both critical retrieval operations have been satisfied. Now recall the softgoals for other operations, ResponseTime[other..., 3] and ResponseTime [update(Card.CreditRemaining), 3]. The good news is that these tasks can be done after the authorization response has been given to the user. The bad news is that the updating of Card.CreditRemaining will trigger additional work (such as the updating of Card.BalanceOwing), which is due to the replication of the formerly-derived attribute. Thus the satisfying of the critical fast-retrieval softgoal *BREAKS* the fast-update softgoal. In addition, the use of PerformLater [update(Card.CreditRemaining), 3] gives further evidence for denying the fast-update softgoal. This is not so bad, since fast-update is not a priority. By

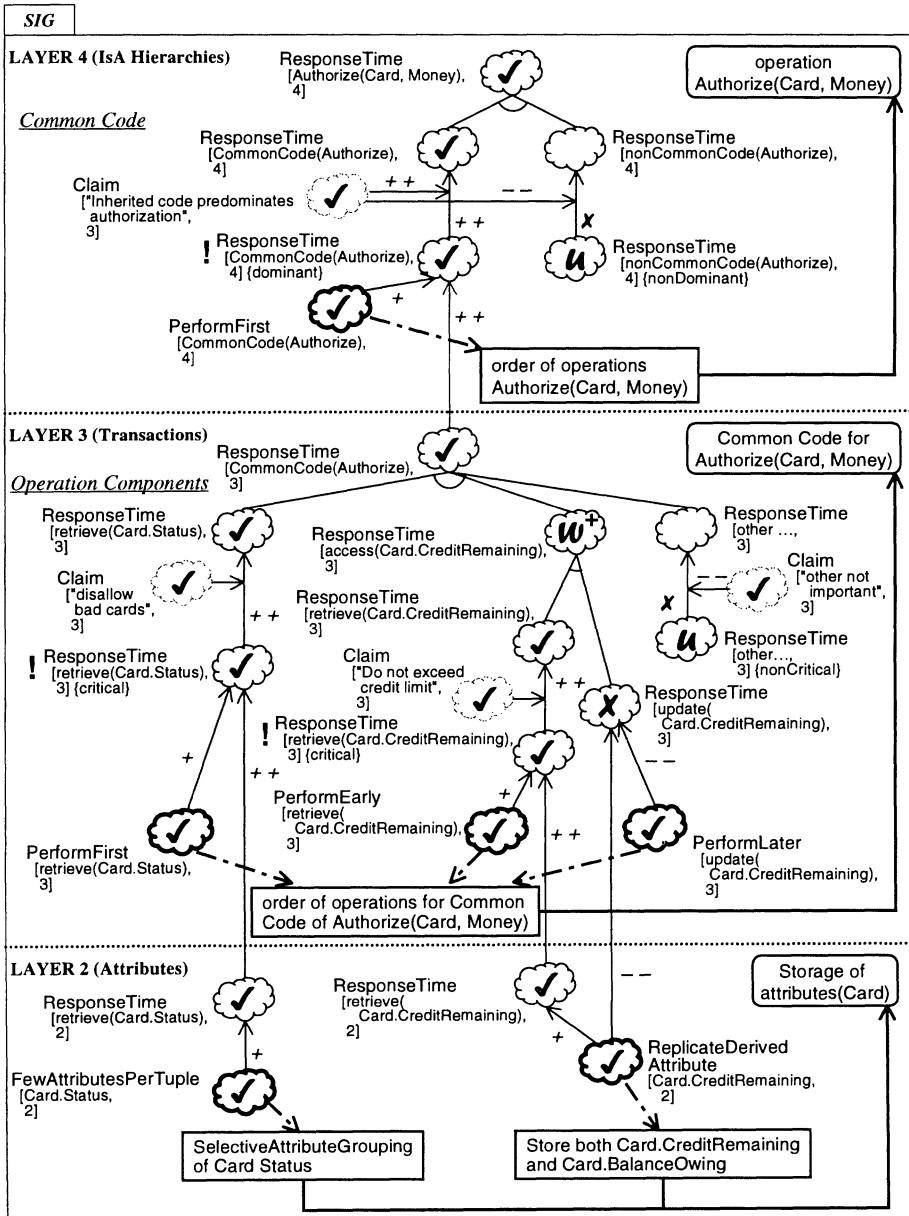


Figure 11.15. Evaluating the impact of decisions.

similar use of propagation rules and consideration of priorities, the top-level softgoals at Layers 3 and 4 are seen to be satisfied. The developer has made arguments that the critical operations of this dominant transaction have been satisfied. Moreover, response-time softgoals were denied primarily for operations which can be performed after responding to the user. Both these results are in keeping with C. Smith's [C. Smith90] principles for building performance into software.

Storage of Cardholder Information

Now let's consider a space requirement. As indicated in Section 11.1, a typical bank might have over 3 million cardholders. Information on these cardholders may include several attributes, and should be stored efficiently.

Inheritance Hierarchies and Priorities for Attribute Storage.

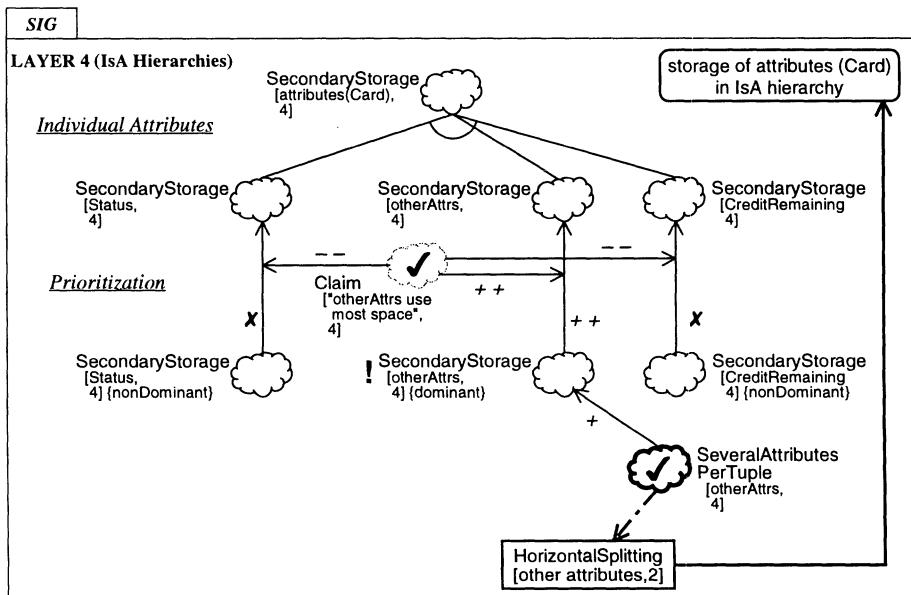


Figure 11.16. Considering inheritance hierarchies at Layer 4.

The softgoal `Performance[Cancel(Card)]`, shown at the bottom of Figure 11.4, was refined using the `SubType` decomposition method (not shown in a figure), into softgoals `Time[Cancel(Card)]` and `Space[Cancel(Card)]`. The top

softgoal of Figure 11.16, for good `SecondaryStorage` of the attributes of `Card`, i.e., the cardholder information, contributes to `Space[Cancel(Card)]`.

Since the `Card` class has specializations, e.g., `GoldCard` which has additional attributes related to travel bonus plans, the softgoal `SecondaryStorage[attributes(Card), 4]` is stated at Layer 4 which deals with `IsA` hierarchies.

The developer uses the `IndividualAttributes` method to decompose the softgoal into softgoals for each of the attributes. To simplify the discussion, we show attributes `Status` and `CreditRemaining` separately (as we have been dealing with them in previous parts of the example), and show all the remaining attributes together as `otherAttrs`. These other attributes are noted as using most of the space, hence their storage is identified as being `dominant`, while `Status` and `CreditRemaining` are `nonDominant`. The space requirement for the other attributes can be satisfied using `HorizontalSplitting` (Section 9.6) on these attributes, which reduces the numbers of tuples used per cardholder.

Using Domain Information to Select Attribute Storage Methods.

At Layer 2 (attribute storage), the developer no longer considers the impact of `IsA` hierarchies. Attribute values can be stored in compressed (encoded) form (bottom of Figure 11.17). This saves space, but incurs extra time when updating or retrieving values. Recall that during critical operations, the other attributes are not used. This argument supports their compression, which satisfies their space softgoal. Chosen operationalizations and satisfied claims are shown in the figure by “ \checkmark .”

Let’s return to the Layer 4 secondary-storage softgoal for the `Status` attribute. As status values do not depend on the `Card` type hierarchy, we refine this softgoal to one at Layer 2, using the `FlowThrough` method (Section 8.3). Recall from Figure 11.13 that it was critical to quickly retrieve `Status` for fast sales authorization. Using a compressed format for this attribute would slow down this dominant transaction. This is an example of a negative interaction between time and space, i.e., a time-space tradeoff. So for this attribute, the developer opts for an uncompressed format a form of *early fixing*. In general, this makes an `UNKNOWN` contribution for `Space` (including `SecondaryStorage`). But here the developer changes the `UNKNOWN` contribution to `BREAKS` to show that satisfying early fixing leads to denial of the storage softgoal.

The refinement for `SecondaryStorage[CreditRemaining, 4]{nonDominant}` at Layer 2 is not shown, but is similar to that of `Status`, since both attributes are critical in the (dominant) authorization transaction. This example thus shows that non-dominant softgoals may need further refinement, due to interaction with other softgoals.

Evaluating the Impact of Decisions.

The result of the evaluation (Figure 11.18) is that the storage softgoals are satisfied for the dominant case, the “other” attributes — which form the

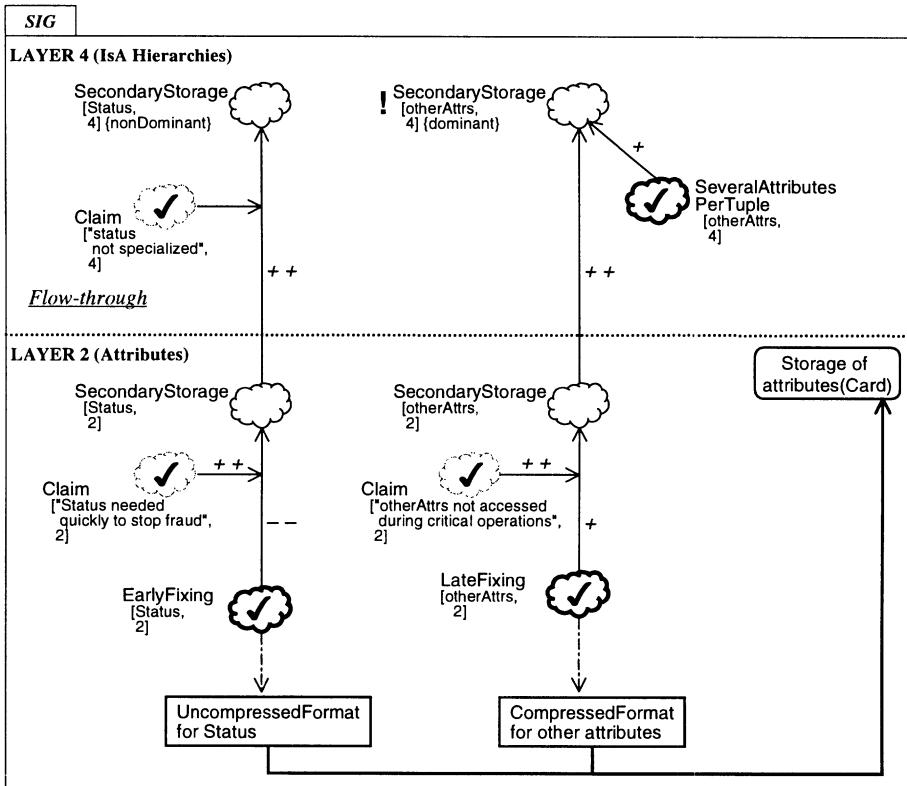


Figure 11.17. Selecting attribute storage methods at Layer 2.

bulk of the attributes of *Card* — while they are denied for only the two attributes whose fast access is very important for satisfying other softgoals. Again, in keeping with C. Smith's [C. Smith90] principles, we have satisfied the softgoals for the dominant requirements (here, for storage). We also have justifications recorded for the softgoals denied.

Storage of Sales Information

In Figure 11.19, the top softgoal is good space performance for sales information, which is kept in secondary storage. While sales records are independent of card type, they could be specialized for each type of merchant (e.g., restaurants, airlines). Hence we do not deal with IsA hierarchies here, so the softgoal is stated at Layer 2.

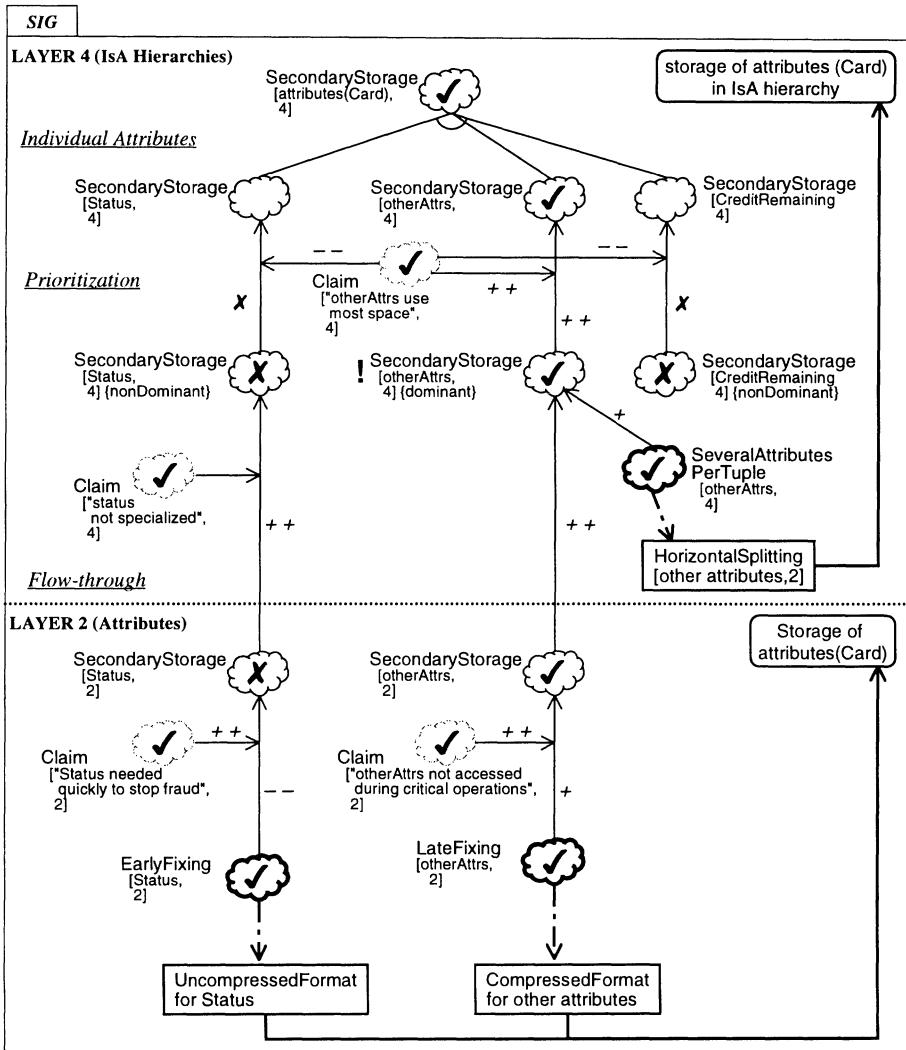


Figure 11.18. Evaluating the impact of decisions.

Figure 11.4 refines Performance[Transaction(Card)] into several top-level NFRs, one of which (not shown in that figure) is Performance[Sale], which is in turn refined into Time[Sale] and Space[Sale]. Now SecondaryStorage[Sale, 2] of Figure 11.19 contributes positively to Space[Sale].

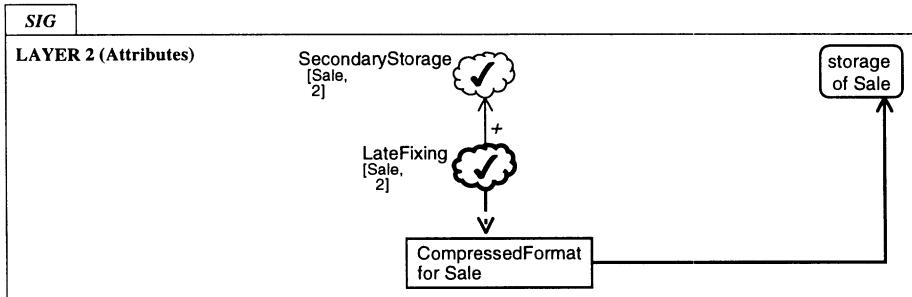


Figure 11.19. SIG for storage of sales information.

A compressed format is used to satisfice the softgoal. This will slow down the recording of sales, but such operations can be done *after* the merchant is given a response by the authorization transaction.

We have looked at several performance requirements. We have selected some aspects of the target system. When such selection decisions were evaluated, we found that some main softgoals have been satisficed. Importantly, priority (critical and dominant) softgoals were satisficed, meeting a principle for building performance into systems. We were able to deal with tradeoffs by focussing on satisficing the priorities, at the cost of not meeting the non-priorities.

Let us now turn to security and accuracy requirements.

11.4 DEALING WITH SECURITY AND ACCURACY REQUIREMENTS

The informal concerns for security are sometimes *ambiguous*, perhaps due to the nature of the document, inviting many possible interpretations. For example, an NFR or operationalizing softgoal needs clarification when it is expressed only in terms of its type name without clear explanation.

Security Softgoals

Consider a main softgoal, **Security[Authorize(Card, Money)]** from the bottom of Figure 11.4, shown at the top of Figure 11.20.

We recognize the importance of accuracy and confidentiality aspects of security. Accordingly, we used the SubType method to decompose a security softgoal, **Security[Authorize(Card, Money)]**, into softgoals for confidentiality, availability and integrity (Figure 11.20).

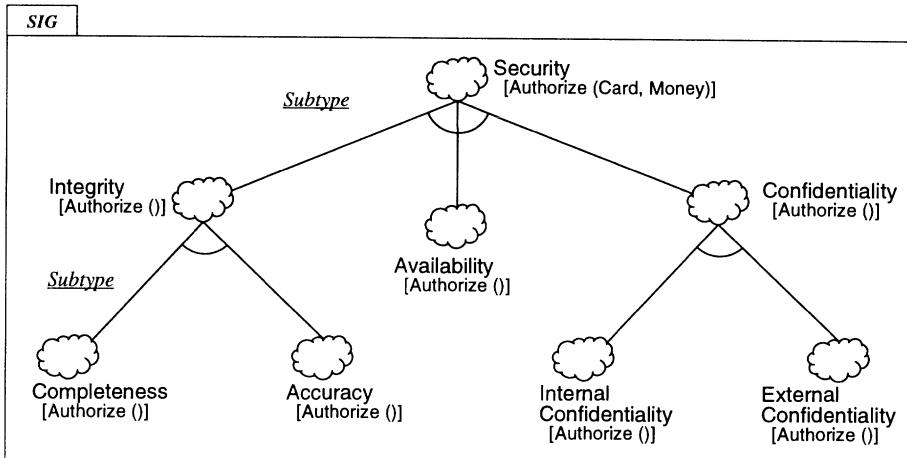


Figure 11.20. Main Security Requirements for the credit card system.

The integrity softgoal is further decomposed into Accuracy [Authorize(Card, Money)] and Completeness[Authorize(Card, Money)]. Recall the discussion in Section 6.7 of different notions of accuracy.

A security breach can take place either internally, by staff accessing the system, or externally, in terms of forgery of vouchers, remittance requests, etc. Thus the confidentiality softgoal is refined into softgoals for internal confidentiality and external confidentiality.

This is done using the Internal-External SubType method, to address two different *characteristics* of confidentiality.

Addressing Accuracy and Confidentiality Softgoals

Of the softgoals identified in Figure 11.20, we now focus on accuracy and confidentiality for authorization of sales. These softgoals are both shown at the top of Figure 11.21, even though they were not at the same level in Figure 11.20.

The developer refines Accuracy[Authorize(Card, Money)] into TimelyAccuracy[Authorize(Card, Money)], PropertyAccuracy[Authorize(Card, Money)] and ValueAccuracy[Authorize(Card, Money)].

The developer focusses on TimelyAccuracy[Authorize(Card, Money)], the faithful representation of information during the lifetime of the authorization transaction. Using a prioritization template, this softgoal is then treated as critical in Figure 11.22. We assume that market surveys show that timely accuracy is of strategic importance.

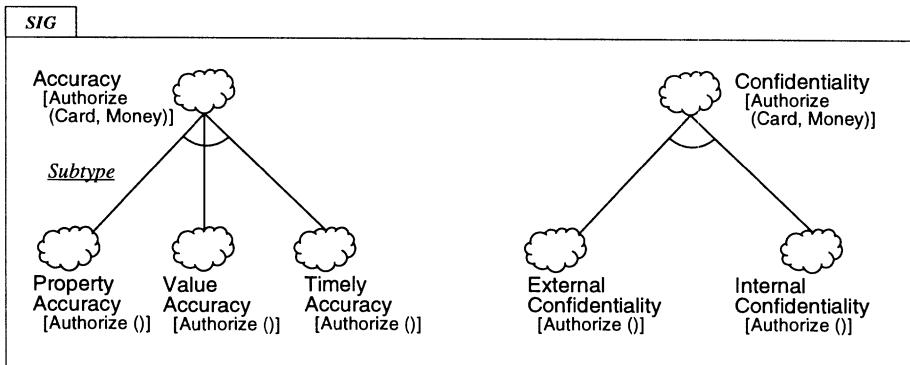


Figure 11.21. Accuracy and Confidentiality Requirements for the credit card system.

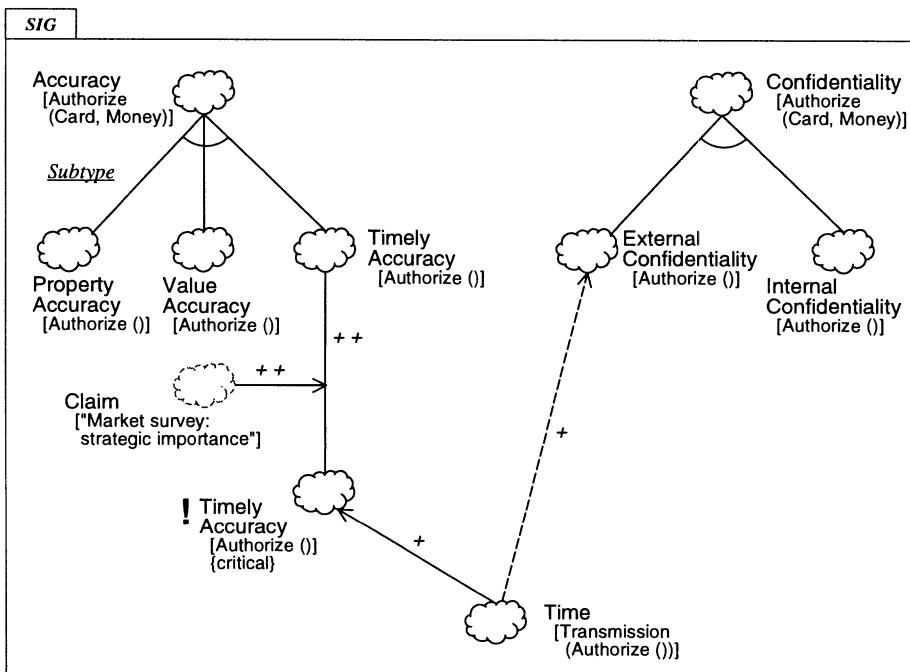


Figure 11.22. Interactions among softgoals.

Now one way to *HELP* meet $\neg \text{TimelyAccuracy}[\text{Authorize}(\text{Card}, \text{Money})]$ {critical} is to provide good transmission time for authorizations. Thus the developer refines the critical timely accuracy softgoal into $\text{Time}[\text{transmission}(\text{Authorize}(\text{Card}, \text{Money}))]$.

Providing good transmission time for transactions has other benefits such as helping external confidentiality. Accordingly, a correlation is detected, showing that $\text{Time}[\text{transmission}(\text{Authorize}(\text{Card}, \text{Money}))] \text{ HELPS } \text{ExternalConfidentiality}[\text{Authorize}(\text{Card}, \text{Money})]$. To avoid undesirable consequences, the correlation catalogue might state a condition that, for the correlation to be positive, the phone line from the input device to the system should be safeguarded against wire-tapping. This helps the developer avoid *omissions* of certain important concerns.

Interestingly, we are able to express different kinds of NFRs, here timely accuracy, time performance and confidentiality, in the same softgoal interdependency graph, using essentially the same notation. If desired, the time performance softgoal could be refined to include a layer parameter (This is not shown in the figure).

We've considered external confidentiality here. Soon we will consider internal confidentiality, in Figure 11.24.

The developer applies the *RapidPosting* operationalization method (Figure 11.23). This helps transmission time, hence timely accuracy, by reducing the time delay for the posting of the sale to the accounts.

Other operationalizations could also be used. For example, installation of a particular kind of point-of-sale transaction device could help rapid posting. However, this operationalization might also increase equipment cost.

The evaluation procedure is used to show the effect of selecting rapid posting (Figure 11.23). Rapid posting *HELPS* transmission time. Using the rules for evaluation, this results in a weak positive label for the satisficing of $\text{Time}[\text{transmission}(\text{Authorize}(\text{Card}, \text{Money}))]$, which the developer changes (here and elsewhere) to satisfied. We also have positive contributions to satisficing external confidentiality of authorization, and to the critical softgoal for timely accuracy of authorization.

Internal Confidentiality Softgoals

The developer focusses on internal confidentiality in Figure 11.24, a continuation of part of Figure 11.23.

The developer applies the *Subclass* method to $\text{InternalConfidentiality}[\text{Authorize}(\text{Card}, \text{Money})]$. This results in two *InternalConfidentiality* softgoals, one for authorization of gold cards, the other for authorization of regular cards.

The developer focusses on authorization for gold cards, and applies the *Subset* method to *partition* the sales amounts (*Money*) into large amounts and small amounts.

We assume that the bank has a policy that care should be exercised in dealing with authorizations involving gold cards and large money amounts (Figure 11.25). Although there are relatively few such transactions, they are

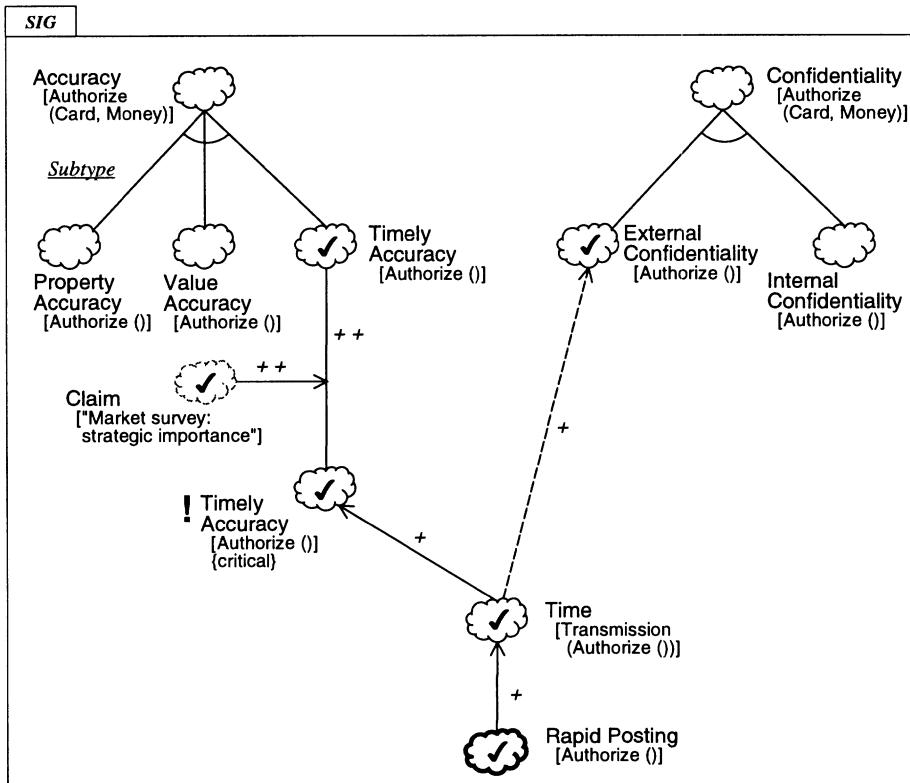


Figure 11.23. Evaluating the use of an operationalization.

treated as critical. This treatment is supported by a prioritization template. In addition, internal confidentiality is identified as being mandatory for authorization of large amounts on gold cards.

Evaluating the Chosen System.

By applying the Alarm operationalization method, the manager of the authorization department can be informed of suspicious authorizations involving large amounts and gold cards.

The alarm can be realized in two ways (not shown in Figure 11.25): PhysicalAlarm, using a loud sound, and SoftAlarm, using a communication network.

The impact of choosing an alarm for authorization of large amounts on gold cards is shown using the evaluation procedure (Figure 11.25). There is a

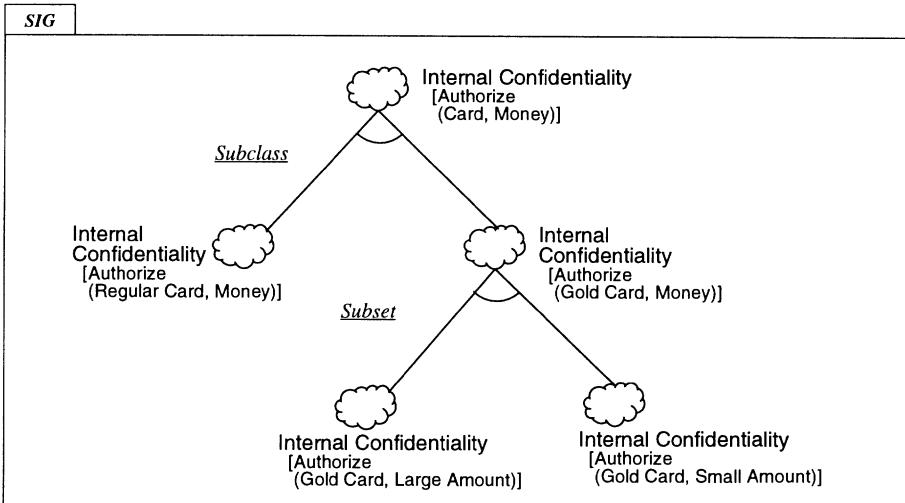


Figure 11.24. Refining an Internal Confidentiality softgoal.

positive impact on internal confidentiality for authorizing large amounts on gold cards. Assuming that `InternalConfidentiality[Authorize(GoldCard, SmallAmount)]` is not a priority, there is a positive contribution to internal confidentiality of all authorizations involving gold cards, `InternalConfidentiality[Authorize(GoldCard, Money)]`.

11.5 DISCUSSION

This chapter has addressed some non-functional requirements for a credit card authorization system. This study has used actual national credit card statistics, and then estimated the portion applicable to one bank. Assumed NFRs for performance, security and accuracy have been addressed. Interestingly, the NFR Framework allows such a variety of NFRs to be addressed together.

The study has involved a series of refinements, and the use of prioritization of softgoals to ensure that the important requirements are satisfied. Prioritization can help the developer deal with tradeoffs between priorities and non-priorities.

For performance, for example, we dealt with time-space tradeoffs. By focussing on priorities at the expense of non-priorities, we obtained results consistent with principles for achieving good response time for users [C. Smith90]. The performance part of the study also illustrated how issues can be handled at more than one layer, while addressing various language features, such as IsA hierarchies.

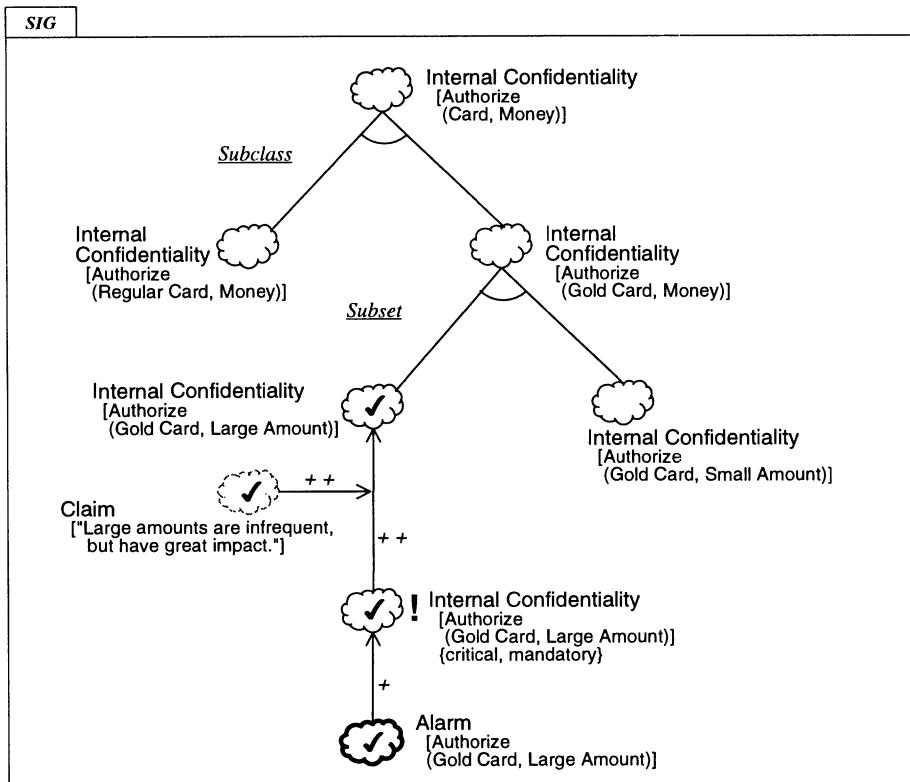


Figure 11.25. Evaluating the impact of alarms on Internal Confidentiality.

In considering security and accuracy requirements, the use of a series of refinements helped the developer focus on priorities. Correlations were helpful in determining the impact of one softgoal upon another. We found that choosing one operationalization can help satisfy more than one NFR.

11.6 LITERATURE NOTES

This chapter is based on studies of NFRs for credit card systems, reported elsewhere [Nixon93, 97a] [Chung93a,b, 95a].

12 AN ADMINISTRATIVE SYSTEM

12.1 INTRODUCTION

In this chapter, an administrative system is studied. Performance requirements are considered for a government system to help administer income tax appeals. This involves long-term, consultative processes. The study considers descriptions of the organization, its workload and procedures.

We studied [Nixon94a, 97a] an Income Tax Appeals system, which manages the handling of taxpayers' appeals of decisions about their tax returns. It provides support for decision-making over a long period of time, recording and tracking information and decisions. We considered a possible extension to this government information system.

Each case is typically complex, and the number of cases is typically much less than the number of short-term transactions in other systems, such as credit cards.

12.2 DOMAIN DESCRIPTION, FUNCTIONAL REQUIREMENTS AND ORGANIZATIONAL WORKLOAD

An Income Tax Appeals system manages the handling of appeals by taxpayers who disagree with decisions made by the Canadian Department of National Revenue. During the appeals process, a taxation officer should contact the taxpayer, and may also consult other staff members. This is a highly consultative and interactive process, subject to constraints, and may last months or even

years. The existing system which records and tracks information and decisions over a long period of time. We considered [Nixon94a] a possible extension to this government information system.

After reviewing a taxpayer's individual tax return, the Department of National Revenue issues a *notice of assessment*, which sets out the amount of tax payable. A taxpayer who disagrees with the assessment may file a formal *notice of objection*. This sets in motion a formal and somewhat complex appeals process, which can last several months or even years, which is intended to provide an impartial re-consideration of an assessment.

We consider a system which monitors the appeals process over long time periods, and a proposed extension which provides a follow-up service to track appeals and issues reminders to staff to keep on schedule.

Currently, the system maintains an inventory control database of information on each appeal case. This draws on manually-prepared documents, which record case disposition and the time spent by staff on each case. This information then is used to prepare reports and handle on-line enquiries.

However, workload control and the supervision of the progress of appeals cases are not automated. We therefore consider an enhanced system. As a partially automated follow-up service would be helpful to departmental operations,¹ we add design specifications and performance requirements which could reasonably be expected for such a system, including the issuing of reminders to staff to contact taxpayers before deadlines.

While a system specification of the existing appeals database system is not available to us, we have access to the taxation operations manual and forms [Revenue Canada80, 89, 92a] which partially describe the appeals system and some of the data in the system. We also use actual taxation organizational workload statistics [Revenue Canada91, 92b,c]. The following description of the appeals process is based upon material presented in, or derived from, departmental information and policies.

The Tax Appeals Process

Normally, a notice of objection must be filed by a taxpayer within a stated time limit (e.g., 90 days from the mailing of the notice of assessment).² Upon receipt by the Department of an objection, several initial steps are taken, including retrieval of files, suspension of account collection activity, and the assignment of the case to one of over 400 appeals officers. During review of the objection, an appeals officer may repeatedly engage in a variety of activities including examination of the taxpayer's account, discussion with the taxpayer, consultation with other staff members, calculations and negotiations. There are some temporal constraints arising from departmental policy. An appeals officer should normally discuss the matter with the taxpayer within 50 days of receipt of the

¹Letter from Fernand Livernoche, Acting Director, Policy and Programs Division, Department of National Revenue, Taxation, November 19, 1992.

²Under new rules, objections may also be filed within one year of the due date of the return.

objection, and complete the review within 90 days, resulting in a decision by the Minister of National Revenue within 120 days. There are several possible outcomes of an objection, which will affect its final processing. For example, if the original assessment is changed, adjustments are made and a notice of re-assessment is issued. If no change is made to the original assessment, the taxpayer is so informed.

Organizational Workload

Actual organizational workload statistics [Revenue Canada92b,c] are used in this study to help deal with performance requirements, and make implementation decisions. Each year, approximately 18 000 000 tax returns are filed [Revenue Canada91]. For the year ended March 31, 1992, some 57 072 appeals were completed, and a closing inventory of 38 532 uncompleted appeals remained at year-end. In each of the 4 quarters in the year ending March 31, 1992, between 64% and 84% of reassessments were issued within 60 days, while 18% to 55% were issued within 30 days. While the average time to dispose of most cases was 104 days, some cases were taking longer than the 120-day service standard. For the quarter ended March 31, 1992, 4% of the reassessments took more than 120 days to be completed. This pattern was fairly consistent for the 3 prior quarters. On March 31, 1992, 5 378 cases remained which were outstanding for over 120 days.

12.3 NON-FUNCTIONAL REQUIREMENTS

We consider some performance requirements for the proposed extension to the system. As the documentation to which we had access did not specifically state system performance requirements, these requirements are based on our intuition and understanding of the domain.

1. There should be good time performance for managing information about the long-term appeal processes in the system.
2. Reminders to staff (e.g., to contact a taxpayer by a certain date) should be issued quickly.
3. Information on appeals (which is used to generate reminders to staff) should be accessible quickly, and stored efficiently.³

To meet performance requirements, we need to consider the performance of implementation alternatives for such a system. For example, part of the appeals process involves repeated access on a timely basis to appeals files. As this can be implemented in many different ways, with widely varying performance characteristics, we need to be able to see the impact (whether positive or negative) of implementation decisions upon performance requirements.

³As we subsequently found out during domain interviews (Chapter 15), storage requirements are not actually a priority in the domain.

The performance of long-term processes (which last months or years, although they may have large idle periods) has been less studied than shorter-term processes. Note also that we are concerned here with system-oriented performance issues (such as reducing time to access an account), rather than organization-oriented performance issues (such as how the appeals branch could change its procedures to reduce the time to consider an appeal by, say, a month); for an examination of organizational goals, see Chapter 14.

12.4 RECORDING DOMAIN INFORMATION IN A DESIGN

We now consider how a developer could record the domain information in a conceptual design.

Characteristics of the application domain lend themselves to the use of TaxisDL (See Section 9.1 and [Borgida93] for descriptions) as a design language. Its data model features (including long-term processes and integrity constraints) can naturally represent the appeals process and proposed follow-up service, which is consultative, interactive and long-term, is subject to legislative and policy constraints, and has many classifications of cases and decisions.

From the domain information, we use the design language to record a schema, including the long-term appeals process. Figure 12.1 is a simplified representation of a portion of the long-term appeals process, represented as a TaxisDL *script*.

Scripts (See Chapter 9 and [Chung88]) are augmented Petri nets [Zisman78]. Control flows between *locations* (or “states,” represented as nodes), along directed *transitions* (arcs). The operations to be accomplished along a transition are specified by the *activities* of a transition.⁴ Transitions also have *givens* (preconditions) which are conditions that must hold before the execution of the transition can begin.

The script represents the appeals process from the viewpoint of an appeals officer, who deals with taxpayers. Starting at the top of Figure 12.1, it shows the initial steps taken when an appeal is received. Next, the officer is reminded (possibly repeatedly) to contact the taxpayer. The officer is also reminded to issue a recommendation for the outcome of the case. The officer may consult with the taxpayer and possibly with staff within the department. This cycle of reminders and consultations may be repeated. After the decision is made, the taxpayer is notified, records are updated, and other divisions of the department are informed.

12.5 OVERVIEW OF SIGs

We illustrate how a system developer can use the NFR Framework, as specialized in the Performance Requirements Framework, to deal with time and space

⁴In TaxisDL, activities (postconditions) of a script transition are called “goals.” These functional goals are distinct from softgoals of the NFR Framework.

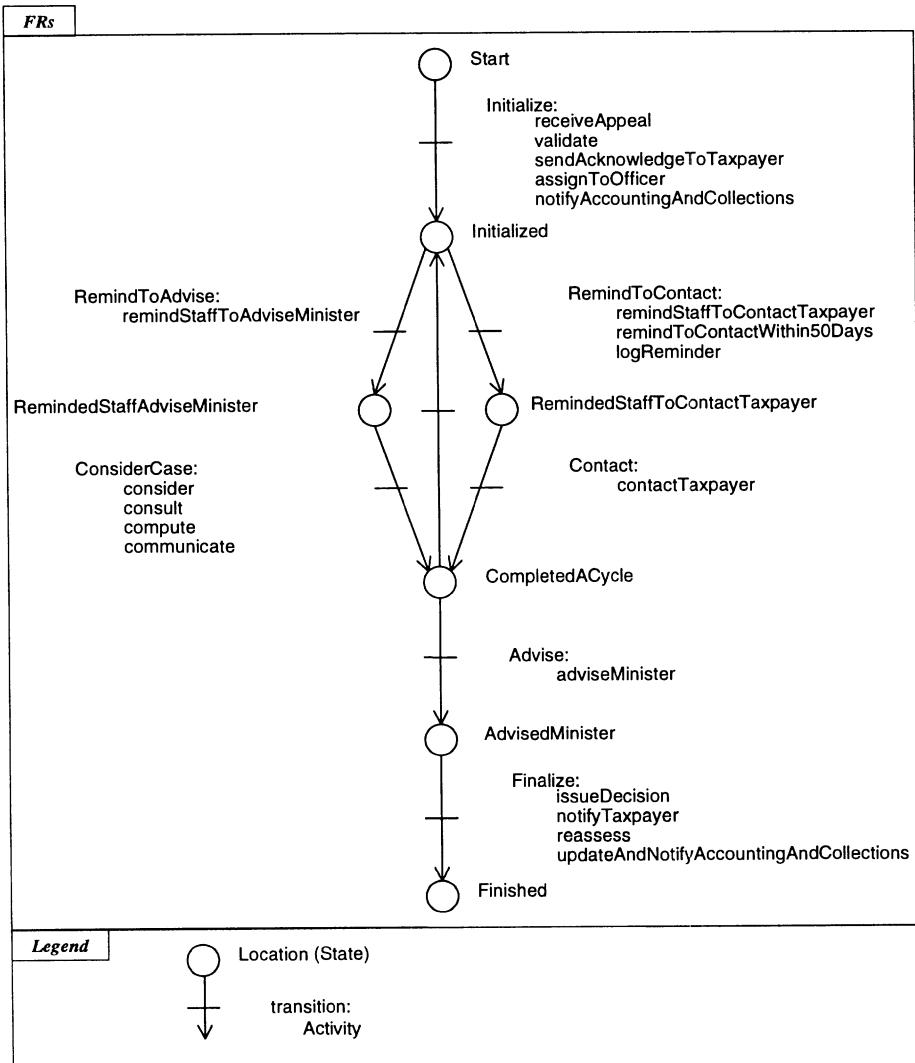


Figure 12.1. The tax appeals process represented as a Script.

requirements for the tax appeal system. We show how results are recorded in SIGs.

We start with a requirement of good **ManagementTime** performance (**MgmtTime**) for managing actions of the long-term tax appeals process. One

important aspect is good management time for issuing reminders to appeals staff to contact taxpayers. Recall from Chapter 9 that management time deals with the time used to manage long-term processes and integrity constraints.

To produce a customized solution, we use some performance refinement methods, which were introduced in Chapters 8 and 9. This is done in conjunction with knowledge of the actual application domain and its workload. We represent, consider and use knowledge of many implementation techniques for semantic data models [Hull87] [Nixon90, 93], such as TaxisDL, including temporal integrity constraints (See [Chung88]). Actual statistics and workload patterns (e.g., many cases are resolved faster than the service standard requires) are used to help deal with tradeoffs and select among implementation techniques.

Issues are considered at a few layers. In developing a SIG, softgoals and decisions at higher layers (here, dealing with long-term processes) have an impact upon lower layers.

Decisions for target systems are made, and then evaluated in terms of meeting the overall softgoals. This information is incrementally recorded in softgoal interdependency graphs

12.6 TIME SOFTGOALS FOR MANAGING LONG-TERM TAX APPEAL PROCESSES

Let's turn to the first performance requirement.

Stating an Initial Softgoal

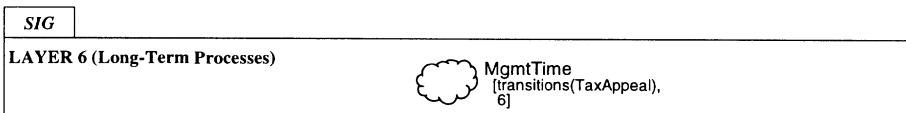


Figure 12.2. Initial softgoal of good time performance for managing transitions of appeal process.

The developer desires good time performance for managing the actions of the long-term tax appeal process. In a TaxisDL script, actions are performed along the *transitions* which are shown as arcs in Figure 12.1.

The developer states a softgoal of good time performance for managing the transitions of the tax appeal process. This time softgoal is stated at Layer 6, which deals with long-term processes, and is shown in a SIG in Figure 12.2.

The softgoal is written as `MgmtTime[transitions(TaxAppeal), 6]`. Here `MgmtTime` (part of the the Performance Type of Figure 9.6) is the *type* of the softgoal, `transitions(TaxAppeal)` is a *topic* of the softgoal, and 6 is the *layer topic*

of the softgoal. The interpretation of the softgoal is that there should be good time performance for managing the transitions of the tax appeal script.

Refining Softgoals

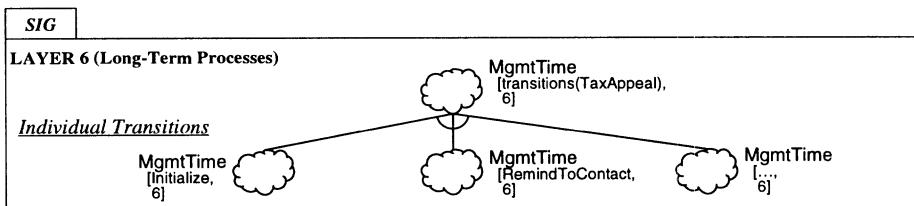


Figure 12.3. Decomposition into softgoals for individual transitions.

Now we can refine the top softgoal into other softgoals, using refinement methods.

By the **IndividualTransitions** decomposition method, a specialization of the **IndividualComponents** method, the management time softgoal is refined into softgoals for each individual transition of the script. The parent softgoal is refined into a set of management time softgoals, one for each transition of the Tax Appeal script (**Initialize**, **RemindToContact**, etc.). This AND decomposition is shown in the SIG of Figure 12.3. In other words, if good management time performance can be attained for each transition, there should be good performance for the group of transitions as a whole.

Further Refinements

Using the NFR Framework, it is up to the developer to chose what to refine, when to refine and how much to refine. We show how a developer can refine softgoals by considering the components and sub-components of a script.

To deal with the proposed reminder service, the developer decides to focus on the transition which reminds the staff to contact the taxpayer, **RemindToContact**. Thus we refine the softgoal **MgmtTime[RemindToContact, 6]**. (Refinements for the softgoals involving **Initialize** and other transitions are not shown here, but could be considered.)

Now TaxisDL script transitions have two components. *Givens* are true at the beginning of a transition, i.e., preconditions. *Activities* are true at the end of a transition, i.e., postconditions. Accordingly, the **TransitionComponents** method can be used to refine the management time softgoal for the **RemindToContact** transition into one management time softgoal for the transition's givens, and one for the transition's activities (middle of Figure 12.4).

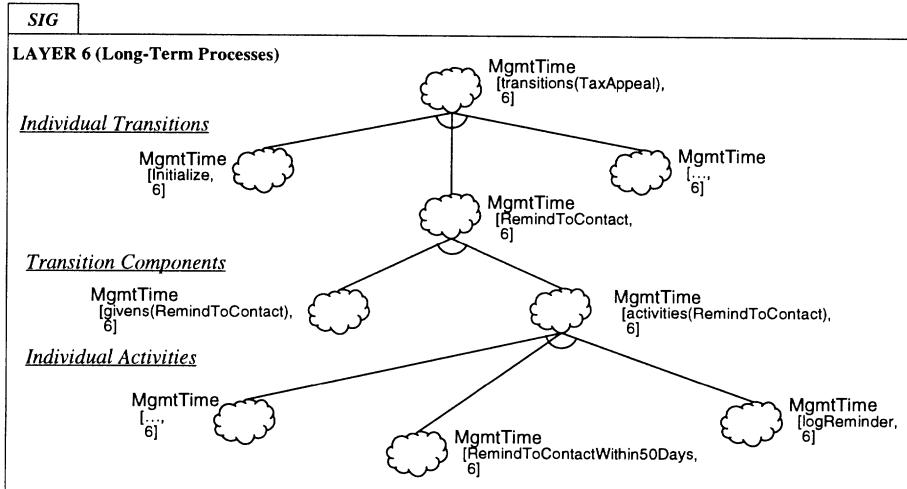


Figure 12.4. Further decompositions at Layer 6.

In a TaxisDL design specification, a transition's activities reflect what is to be achieved. Examples here include reminding the staff to contact the taxpayer, preparing a log of reminders issued, etc. Accordingly, the management time softgoal for a transition's set of activities is refined into several management time softgoals, one for each activity of the transition. The activities include reminding the agent to contact the taxpayer within 50 days, and making a log of the reminder. This refinement is done by using the `IndividualActivities` decomposition method (bottom of Figure 12.4).

Linking Softgoals and SIGs at Different Layers

Recall from Chapters 8 and 9 that performance softgoals can be associated with particular *layers*, which helps further organize SIGs into layers. Each layer deals with a set of issues (here, particular data model features of the source language). Methods can be used to move down to lower layers, dealing with simpler languages (which omit the features of higher layers), until we get to the target language.

Here, in considering the long-term tax process at Layer 6, the developer pays attention to the management time softgoal for the transition activity specification that staff contact the taxpayer within 50 days the first time. This softgoal is written `MgmtTime[RemindToContactWithin50Days, 6]`.

We consider how this might be implemented. One way would include issuing a reminder to the staff within 40 days of the appeal: if not already

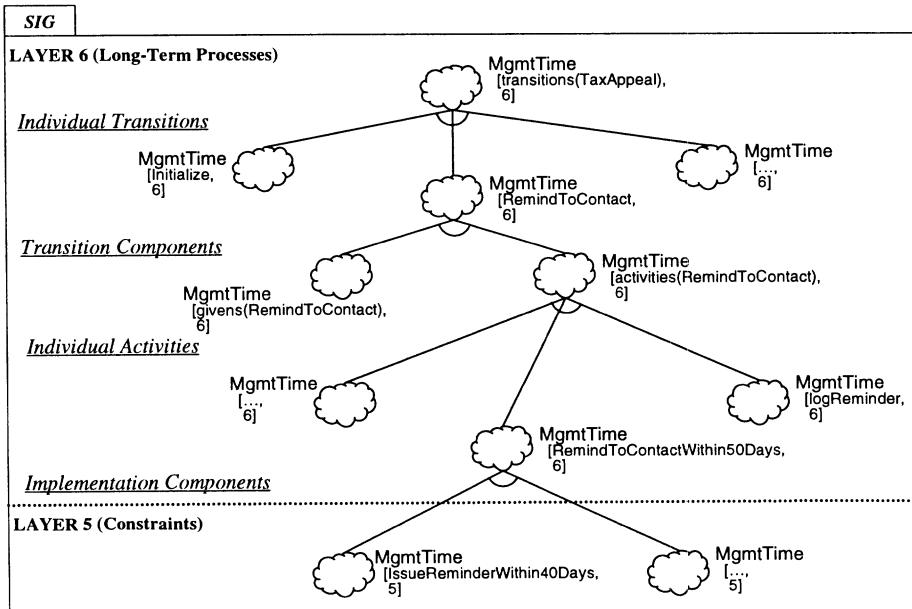


Figure 12.5. An inter-layer refinement.

contacted, the taxpayer could be contacted within the remaining 10 days, to meet the service standard. We can then refine the management time softgoal using the **ImplementationComponents** decomposition method, which decomposes a parent softgoal into softgoals for the components which implement the topic of the parent softgoal (bottom of Figure 12.5).

Note, however, that we have taken a problem involving one layer (here, long-term processes at Layer 6) and are now dealing with part of it in terms of a lower layer – here, integrity constraints at Layer 5. We now are dealing with a *temporal integrity constraint*, to issue reminders within a certain time.

This reduction to lower (simpler) layers is represented by an *inter-layer interdependency link*. This results in softgoals at the lower layer (e.g., MgmtTime[IssueReminderWithin40Days, 5]) which can now be refined, satisfied, etc., but now focussing on the issues, methods, etc., applicable to this lower layer. In effect we have started another SIG at a lower layer. The inter-layer interdependency links connect the SIGs at each layer to form a complete SIG involving all applicable layers. Decisions at lower SIGs have an impact upon higher layers, and their impact is propagated upwards via inter-layer contributions.

12.7 OPERATIONALIZATION METHODS FOR INTEGRITY CONSTRAINTS

We continue the development, focussing on the softgoal of good management time performance for issuing a reminder within 40 days. We are now at Layer 5, addressing integrity constraints.

Considering Operationalization Methods

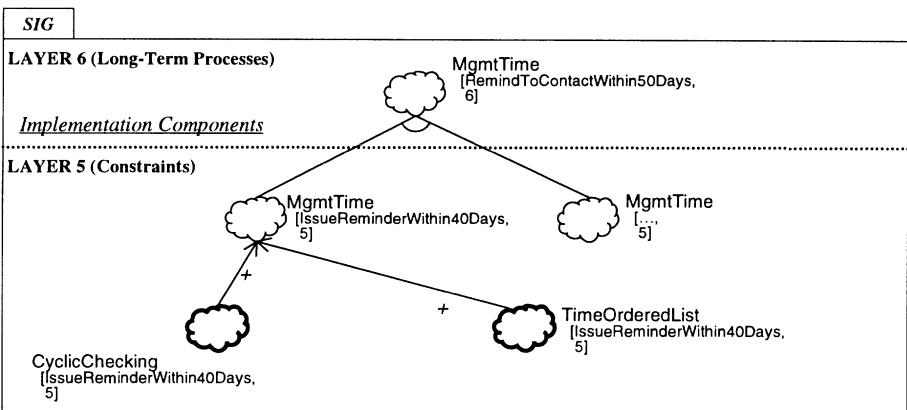


Figure 12.6. Considering some operationalizing softgoals.

In moving towards a target system, the developer considers operationalization methods (implementation techniques) which will satisfy the performance softgoals. In Figure 12.6, we consider operationalizing softgoals derived from implementation techniques for temporal constraints [Chung88]. One option is **CyclicChecking**, which cycles through and examines each transition of each taxpayer's script. If it is 40 days since the appeal was filed, a reminder is sent to staff. This gives reasonable performance with a small number of script instances. Otherwise, the time overhead of repeatedly checking each script is too great, since only a few scripts will meet the temporal condition.

The developer considers another operationalization method for temporal constraints, the **TimeOrderedList** [Chung88], which keeps script instances in a list, sorted by the time at which they need to be activated. If the order of actions is known in advance, the sorting provides efficient access time, since we only examine the front of the list. Appeals with the appropriate date are removed from the list, and reminders are issued.

Figures 12.6 through 12.8 omit much of Layer 6, to condense the figures.

Using Domain Information to Argue about Alternatives

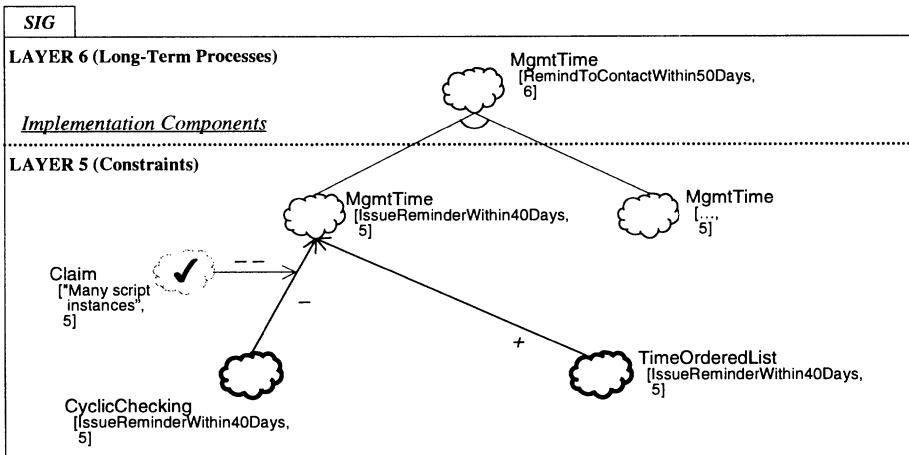


Figure 12.7. An Argument about an Operationalization.

The process of selecting implementation alternatives needs to reflect different needs, priorities and workloads of the particular organization for which the system is being produced. These factors can be used to support decisions made by the developer. For example, organizational workload (e.g., the number of appeal cases, or the speed of resolving cases) can be used as arguments for choosing one operationalization over another. An argumentation softgoal (Claim) records such design rationale.

When there are many script instances, the time overhead for cyclic checking is great, since every script will be repeatedly checked. However, the temporal condition (being 40 days from the filing of the appeal) will be met for only a small number of scripts. This is in fact the case for the tax appeal application, as the workload statistics [Revenue Canada92b,c] indicate that there were recently 38 532 appeals pending, which, while not huge, is not insignificant. Hence in Figure 12.7 the developer argues that the criterion for using cyclic checking is not met.

As a result, the positive contribution of **CyclicChecking** in Figure 12.6 becomes a negative contribution in Figure 12.7. In Figure 12.8, the developer rejects **CyclicChecking**. In addition, the developer *denies* the *interdependency link* (shown as an “x” along the link), so that **CyclicChecking** does not contribute to its parent. Thus **CyclicChecking** is removed from further consideration.

Refining Operationalizing Softgoals

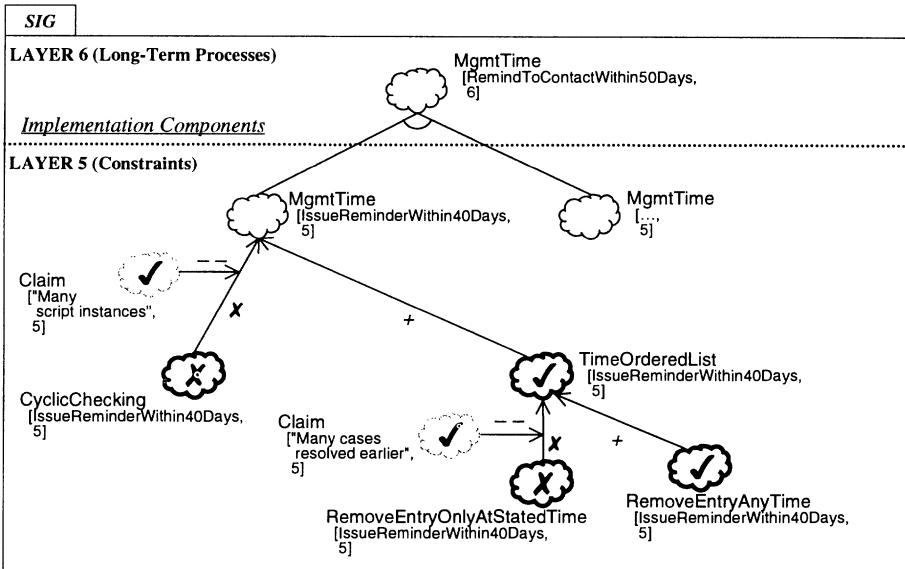


Figure 12.8. Refining an Operationalizing Softgoal.

Here we have chosen a list structure for the implementation. We may wish to consider more specialized structures. This can be done by *refining an operationalizing softgoal* into other operationalizing softgoals.

For example, a specialized list structure may be more efficient. A queue structure (*RemoveEntryOnlyAtStatedTime*, a specialization of *TimeOrderedList*, shown at the bottom of Figure 12.8), can be optimized to restrict removal operations to the front of the list. This would offer some management time and space advantages if reminders are issued for every appeal on the 40th day. However, the queue is ruled out because the workload patterns show that reminders are not always needed, as some appeals are handled faster than the service standard.

In fact, staff may contact taxpayers (and even conclude the appeal) before the deadline; in these cases, reminders are not needed. If this is a frequent situation, the developer may be able to exploit it to benefit performance. For guidance, the developer again turns to the workload statistics, and observes that some cases are resolved within 30 days, and a majority are resolved within 60 days. Hence, many cases are concluded before the 40-day service standard, and in some others the taxpayer has been contacted before then. An implication for implementation is that entries need to be able to be efficiently removed

in advance of the normal time. The developer therefore uses the statistics to argue against the queue structure, and instead chooses a more flexible list structure which easily allows early removal, `RemoveEntryAnyTime`.

Considering Performance Softgoals at Lower Layers

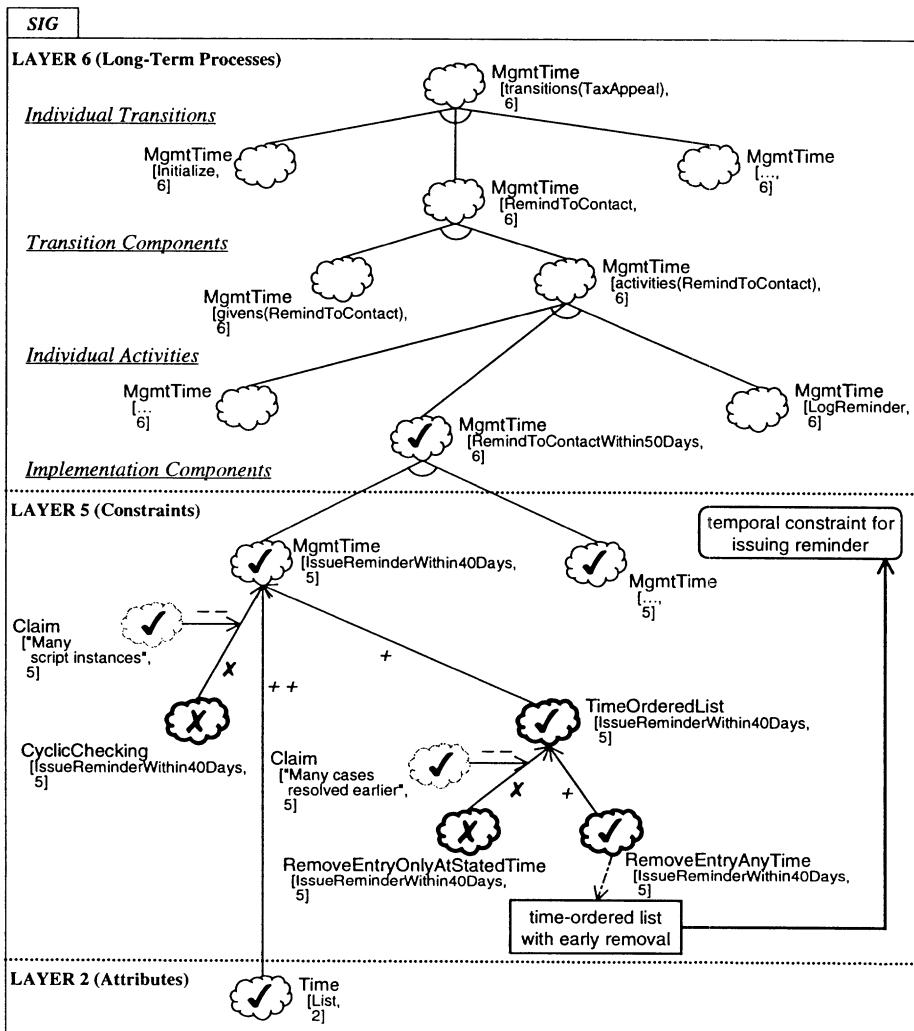


Figure 12.9. Evaluation of the SIG.

Having chosen a particular list structure at Layer 5 to meet the integrity constraint, the developer still needs to deal with details of storing the attributes of the list structure, which is considered at Layer 2. Note that the developer can choose to skip intermediate layers (here, Layers 4 and 3) when the softgoals under consideration need not be refined to address the issues of those layers (here, IsA hierarchies, and transactions).

To manage long-term processes and integrity constraints at higher layers, we have dealt with **MgmtTime** softgoals for reminders. At the lower layers of implementation, we will be dealing with shorter-term **Time** softgoals for maintaining a list structure. As a result, the developer refines **MgmtTime** [**IssueReminderWithin40Days**, 5] into **Time**[**List**, 2]. The resulting interlayer link is shown at the bottom of Figure 12.9.

This *inter-layer refinement* can be viewed as a variant of the **ManagementTime** to **Time** method (Section 9.7) which takes a parent **MgmtTime** softgoal at Layer 6 or 5, and refines it to a **Time** softgoal at a lower layer. However, here we have also changed the topic of the offspring softgoal.

Recall from Section 8.3 that an NFR softgoal which is an offspring at the top of a lower layer must have a parent which is an NFR softgoal, not an operationalizing softgoal, at a higher layer. This refinement is in keeping with that provision of the NFR Framework.

The refinement of the Layer 2 softgoal will be discussed in Section 12.8.

Evaluating Softgoal Accomplishment

Figure 12.9 also shows the evaluation of softgoal interdependency graphs. To determine the overall effect of local decisions upon top NFR softgoals, an evaluation procedure (labelling algorithm, Section 3.3) is used. Importantly, the procedure can dynamically ask developers for their expertise, e.g., when there are inconsistencies, or situations not handled by an existing catalogue.

Figure 12.9 is a combined SIG of the development described in Sections 12.6 and 12.7. Evaluation of multi-layer SIGs proceeds from the bottom of the figure, starting with the leaf softgoals, commencing at the lowest layer.

At each layer of a fully refined SIG, most leaves are either operationalizing softgoals or claims, which are labelled. In other words, the NFR softgoals have been refined to operationalizing softgoals, each of which has been chosen or rejected. Otherwise, the developer can assign labels to unlabelled leaves. Here, at the bottom of Figure 12.9, **Time**[**List**, 2] is not yet refined, but the developer assumes it will be satisfied, and labels the softgoal accordingly. In Section 12.8, it will be refined, and we will see that the result of evaluating the resulting SIG at Layer 2 does satisfy **Time**[**List**, 2].

Now **Time**[**List**, 2] is labelled satisfied, and *HELPS* its parent, **MgmtTime**[**IssueReminderWithin40Days**, 5]. Using the rule for this combination of label and link type, a weak positive contribution (“W⁺”) is made to the parent. We’ll return to the label of the parent softgoal shortly. In the meantime, note that the propagation rules work for an inter-layer interdependency link in the same way as for links within one layer.

Now we go up to Layer 5. The three leaf operationalizing softgoals have each been already accepted (labelled as satisfied) or rejected (denied). The two rejected ones, `CyclicChecking` and `RemoveEntryOnlyAtStatedTime`, have arguments (`Claims`) attached to their links to their respective parents. Their claims have been denied. In both cases, the claims are conditions, based on domain and workload knowledge, under which the use of the operationalizing softgoals would be appropriate. The result is to deny the offspring-parent *interdependency links* (shown by “ \times ” along the links). The effect of denying interdependency links (which are normally considered satisfied) is that no value is propagated to the parent, i.e., the offspring are removed from consideration.

Turning to the remaining leaf operationalizing softgoal, `RemoveEntryAnyTime`, it is selected (satisfied), and is connected to its parent by a *HELPS* link. This propagates W^+ to the parent, `TimeOrderedList`, which has no contributions from other offspring.

Now the developer can step in, using expertise to consider that `TimeOrderedList` is satisfied. As a result, its label of “ W^+ ” is changed to “ \checkmark ”.⁵

Now we go up to `MgmtTime[IssueReminderWithin40Days, 5]`. Recall that it receives a W^+ contribution from `Time[List, 2]` and no contribution from `CyclicChecking`. It also receives W^+ from the *HELPS* link from satisfied `TimeOrderedList`. The developer considers that the combination of the two W^+ contributions, one from Layer 2, the other from Layer 5, in fact satisfies `MgmtTime[IssueReminderWithin40Days, 5]`, which is labelled with “ \checkmark ”. Notice that results propagated from lower layers via inter-layer interdependency links are combined with results from other softgoals at the higher layers, in the normal way.

In the remainder of the SIG, there are several leaf NFR softgoals (e.g., `MgmtTime[..., 5]` and `MgmtTime[LogReminder, 6]`) which have not been refined to operationalizing softgoals. The developer could choose to refine softgoals more fully. Another alternative would be to assume that there are satisfied, and label them with “ \checkmark ”.

Now using the rule for *AND* contributions, since the offspring `MgmtTime[IssueReminderWithin40Days, 5]` and `MgmtTime[..., 5]` are satisfied, so is their parent at Layer 6, `MgmtTime[RemindToContactWithin50Days, 6]`.

At this point, we have satisfied the second performance requirement, good management time for issuing reminders to staff, by satisfying `MgmtTime[RemindToContactWithin50Days, 6]` and `MgmtTime[IssueReminderWithin40Days, 5]`. This helps satisfy some softgoals at Layer 6.

Thus there is some positive contribution towards satisfying `MgmtTime[transitions(TaxAppeal), 6]`. However, further refinement (not shown) would be needed to definitely satisfy that top-level softgoal, and provide good time performance for managing information about the long-term appeal processes in the system, which was the first performance requirement listed in Section 12.3.

⁵This kind of label change from “ W^+ ” is done frequently, especially for *HELPS* links, and is usually not shown in diagrams.

In Section 12.8 we will look at the third requirement, that information on appeals should be accessible quickly, and stored efficiently.

Relating Functional Requirements to the Target System

The right side of Figure 12.9 relates functional requirements (here, the temporal constraint for issuing a reminder) to a chosen implementation (here, the particular list structure) at Layer 5. We have shown links from source to target within one layer. They can also relate the overall design to the implementation target. At Layer 6, we have not refined NFR softgoals to satisficing softgoals; hence we do not link that layer to functional requirements.

The lower right of the figure relates operationalizing softgoals to the selected implementation.

12.8 DEALING WITH A TRADEOFF

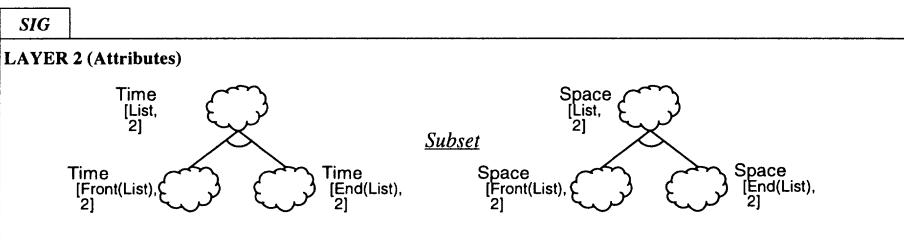


Figure 12.10. Time and Space softgoals and their refinements.

In addition to the good time performance for handling the list of appeals, **Time**[List, 2], the developer also requires good **Space** performance. This softgoal is written **Space**[List, 2] at the top of Figure 12.10. As we will see, we will deal with a time-space tradeoff.

To effectively maintain the list of reminders, items in the list are arranged in date order. Cases that need to be handled on the current date are placed at the front, while cases that do not need to be handled until much into the future are placed at the back of the list. In considering possible refinements to these softgoals, the developer considers different types of access to the time-ordered list.

Hence new appeals, which require future action, will generally be inserted towards the back of the list, while retrievals (for seeing if a reminder should be issued) will generally be from the front of the list, and deletions (when a reminder has been issued *or* the taxpayer has been contacted) will often be from the front of the list. This suggests that the front of the list may be considered separately from the rest of the list. The **Subset** method, can effect

such a partitioning. It refines the time softgoal for the list into softgoals for the front (where many accesses are made), $\text{Time}[\text{Front(List)}, 2]$, and the remainder (end) of the list, $\text{Time}[\text{End(List)}, 2]$. The space softgoal is refined in the same way, using domain and development knowledge.

Identifying Priorities

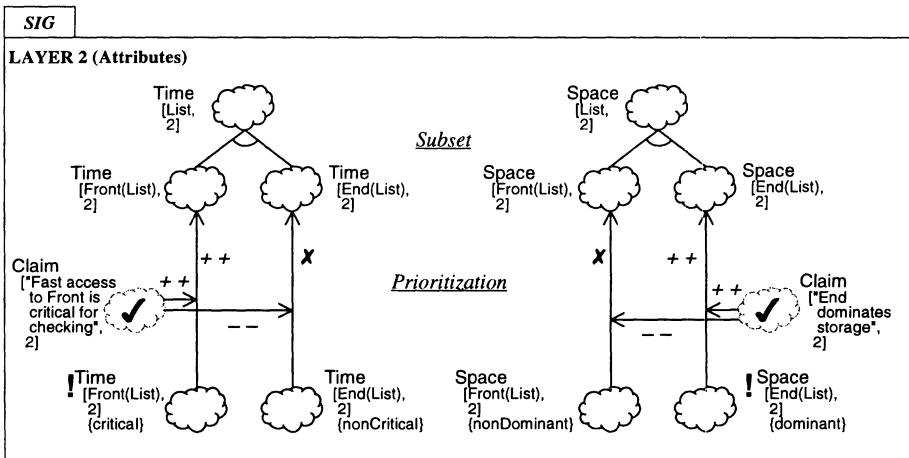


Figure 12.11. Identifying priorities using workload-based arguments.

The developer now focusses on *priorities* (critical and dominant parts of the workload) [C. Smith90]. The developer notes that fast access to the front of the list is *critical* for fast checking of the temporal integrity constraint, because the front is the location of the appeals which need reminders the soonest. On the other hand, there are only a few entries at the front, so the developer argues that the end of the list *dominates* storage.

These observations are stated as *arguments* (Claims, in Figure 12.11), and are based on knowledge of the domain and its organization workload. In some cases, we can use standard arguments; an example is the VitalFew argumentation template, which argues for a form of the “80–20 rule” [Juran79] [McCabe87].

This prioritization results in two softgoals being identified as *critical* or *dominant* (e.g., $!\text{Time}[\text{Front(List)}, 2]\{\text{critical}\}$). The other two softgoals are identified as *non-priorities*. The result of identifying the other softgoals as (*nonCritical* or *nonDominant*) is to remove them from consideration. This is done by denying their contributions to their parents.

Developing an Hybrid Method to Deal with Tradeoffs

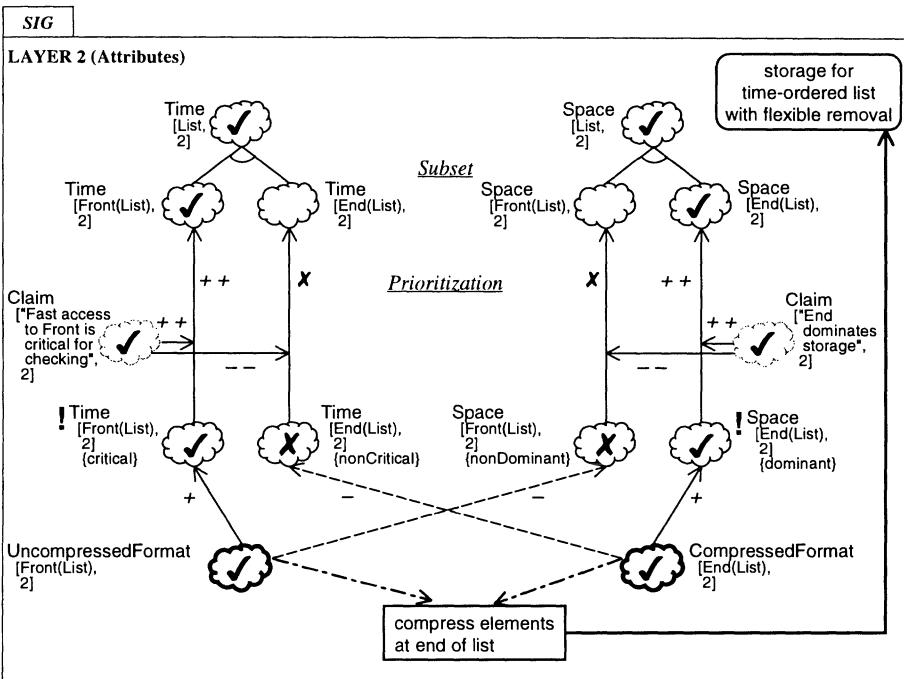


Figure 12.12. Developing an hybrid method to deal with tradeoffs.

The developer now deals with a time-space tradeoff. One operationalization method is to use **CompressedFormat** for attributes. This saves storage space, but accesses will require extra time, both for compression when storing, and uncompression when retrieving.

The developer applies this observation to the priority softgoals. The developer considers an hybrid implementation where the attributes at the front of the list (corresponding to appeals for which a reminder should be issued imminently) are uncompressed, while those at the back (corresponding to appeals for which a reminder need not be issued until some future date) are compressed. **UncompressedFormat** *HELPS* the critical **Time** softgoal, based on the method definition of Section 8.4. Likewise, **CompressedFormat** *HELPS* the dominant **Space** softgoal. These *HELPS* contributions are shown in Figure 12.12. Selected operationalizations are labelled as satisfied ("✓").

What about the non-priority softgoals? Correlations can relate softgoals which have not been explicitly linked. Here, **UncompressedFormat** *HURTS* the

non-dominant Space softgoal. Likewise, CompressedFormat HURTS the non-critical Time softgoal. These HURTS links are identified by comparison of the SIG with the correlation rules in Figure 8.15. Here, the correlations have negative impact, but, fortunately, only on the non-priority softgoals.

Thus the developer chooses to only compress attributes which are at the end of the list, corresponding to reminders which need not be issued until some future date. This results in fast access time to appeals at the front of the list, which is critical for issuing reminders needed soon. It also provides efficient space usage for the end of the list, which dominates storage. However, the storage softgoal for the front is not met. This is not so bad, since it does not dominate storage. Similarly, the non-critical time softgoal for access to the end of the list is not met.

The chosen operationalizations are related to the target system at the bottom of the figure. The target system is related to the functional requirements at the top right of the figure.

Evaluation of Softgoals

A frequent situation for AND decompositions is when a satisfied priority softgoal is combined with an unsatisfied non-priority one. The developer has the flexibility to adjust the label values of non-priority softgoals so that the overall softgoal is considered satisfied.

As usual, evaluation within this layer starts at the bottom (Figure 12.12). Here the developer labels the two selected operationalizing softgoals as satisfied. These propagate a weak positive (“W⁺”) contribution to the priority parents, and, via the correlation links, a weak negative (“W⁻”) contribution to the non-priority parents.

However, the developer steps in to label !Time[Front(List), 2]{critical} and !Space[End(List), 2]{dominant} as satisfied (“✓”), since the priority softgoals have been satisfied.

Now these satisfied priority softgoals MAKE their respective parents, Time[Front(List), 2] and Space[End(List), 2].

Now consider the non-priority softgoals Time[End(List), 2]{nonCritical} and Space[Front(List), 2]{nonDominant}. First the developer changes their weak negative (“W⁻”) labels to denied (“✗”). Since the *interdependencies* from the non-priority softgoals to their parents are denied (shown as “✗” along the interdependency links), the parents Time[End(List), 2] and Space[Front(List), 2] are removed from further consideration.

As a result, the top-level softgoals , Time[List, 2] and Space[List, 2], are satisfied. This is because each has only one remaining offspring participating in an AND interdependency, and the one offspring is satisfied.

This is an example of using *prioritization* templates of Figures 4.30 and 4.31 and Section 8.3. to give the developer flexibility to address the stated needs of the domain. Note that the critical and dominant softgoals are satisfied. Only the non-critical and non-dominant are not satisfied. This

is in accordance with principles for achieving good response time for users [C. Smith90].

As a result, the top time and space softgoals are satisfied. Recall from Section 12.7 that when doing the evaluation we assumed that **Time**[List, 2] had been satisfied. As this in fact is the case, the evaluation of that section would not be changed.

We have shown how the NFR Framework can handle time and space softgoals, and a tradeoff. We have considered a time-space tradeoff.

12.9 DISCUSSION

This study has illustrated the use of a number of aspects of the Performance Requirements Framework. This includes the basic representation scheme of the NFR Framework (softgoals, contributions, methods, etc.), as well as knowledge specific to performance and implementation of information systems. The approach has been illustrated for long-term, consultative processes, and integrity constraints. Layered structures were used to further organize the development process, by addressing some data model features first, and reducing the remaining problem to address simpler features.

We have used domain-specific workload information and have dealt with priorities and tradeoffs. Results in SIGs have been consistent with some principles for achieving good response time for users.

After the study was done, we interviewed an expert from the domain. From the feedback, we found that actual priorities and tradeoffs in the domain were different from our assumptions. We found that time was a priority for the domain. However, space was not; instead, accuracy was important. Hence the actual system had to deal with a time-accuracy tradeoff, rather than a time-space tradeoff. Feedback is discussed in Chapter 15.

12.10 LITERATURE NOTES

This chapter is based on a study of an income tax appeal system, reported elsewhere [Nixon94a, 97a] [Chung95a].

Another administrative system has been studied using the NFR Framework [Chung93a, 95a]. It addressed a proposed system to manage the flow of documents used in the decision-making process of the Cabinet of a government. Confidentiality is a critical concern in such systems. Both the Cabinet document and tax appeals processes are highly consultative and interactive. They are subject to constraints, and can last for several months.

13 APPLICATION TO SOFTWARE ARCHITECTURE

Non-functional requirements, such as modifiability, performance, reusability, comprehensibility and security, are often crucial for software systems. They should be addressed as early as possible in a software lifecycle, and properly reflected in a software architecture before a commitment is made to a specific implementation.

This chapter applies the NFR Framework to a particular phase in Software Engineering, namely *software architectural design*. We describe how to use the NFR Framework to systematically guide a software architect in selecting among architectural alternatives. This provides *goal-driven, process-oriented architectural design*.

We illustrate our approach to addressing NFRs in the software architectural design process by using a standard example from the software architecture field. We consider a variety of NFRs for a *Keyword in Context (KWIC)* system. NFRs such as modifiability and comprehensibility are addressed using the NFR Framework, but are not treated in the detail that has been done for accuracy, security and performance in Part II.

In our view, the work of [Chung95c,d], which is presented in this chapter, is one of the first that considers adaptation of the NFR Framework specifically in the context of software architecture.

13.1 INTRODUCTION

Software architecture is becoming an increasingly important topic in software engineering (e.g., [DagstuhlWorkshop95] [IWASSWorkshop95]). Some (e.g., [Boehm94] [Perry92] [Kazman94]) have argued convincingly for the importance of addressing non-functional concerns in software architectures. However, as pointed out by Garlan and Perry [Garlan94], architectural design has traditionally been largely informal and *ad hoc*. The manifested symptoms include difficulties in communication, analysis, and comparison of architectural designs and principles.

A more disciplined approach to architectural design is needed to improve our ability to understand the interacting high-level system constraints and the rationale behind architectural choices, to reuse architectural knowledge to make the system more changeable, and to analyze the design with respect to NFRs.

This chapter outlines an approach by which such knowledge can be organized. This chapter discusses how the NFR Framework can be applied to systematically guide selection among architectural design alternatives. The NFR Framework has been applied to several NFRs (particularly accuracy, performance and security), has been used to study several information systems, and is now applied to architecture.

After presenting functional and non-functional requirements for the KWIC system, and discussing software architecture and approaches, this chapter shows the use of the NFR Framework to catalogue knowledge of software architecture. Then we illustrate the use of the NFR Framework by a study of architectural design for a KWIC system.

Functional Requirements and Target Alternatives for the KWIC System

Here we are developing an architecture for a Keyword in Context system, which was formulated by Parnas [Parnas72]:

The KWIC (Key Word in Context) index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

The example is chosen since it is relatively well known and used in several studies (by Parnas [Parnas72], Garlan et al. [Garlan92], and Garlan and Shaw [Garlan93]) which provide a good illustration of tradeoffs among NFRs and design alternatives for the KWIC domain. However, it is used primarily as an instructional example in this chapter, so that we can illustrate the use of the NFR Framework for selecting among alternatives for architectural design.

We will consider four architectural design alternatives that Garlan and Shaw [Garlan93] discuss (the first two were considered by Parnas [Parnas72], and the third is a variant which was considered by Garlan et al. [Garlan92]):

1. Shared Data: In this approach, a main program (**Master Control**) iterates in sequence through the four basic modules: **input**, **shift**, **alphabetize**, and **output**. Data communication between the modules is carried out by means of shared storage, which is accessed with an unconstrained sequential read-write protocol.

2. Abstract Data Type: Instead of direct sharing of data, each module accesses data only by invoking procedures in the interface that each module provides.

3. Implicit Invocation: Like the Shared Data approach, modules share data, but through an interface. Unlike the Shared Data approach, module interaction is triggered by an event. For example, adding a new line to the line storage triggers the **Circular Shift** module to do the shifting (in a separate abstract shared data store), which in turn causes the **Alphabetizer** to alphabetize the lines.

4. Pipes and Filters: Using a pipeline, each of the four filters processes data and sends it to the next filter. With distributed control, each filter can run only when it has data transmitted on pipes.

Addressing Non-Functional Requirements during Architectural Design

We want to meet non-functional requirements for modifiability, performance, etc., for the KWIC system. But if we optimize performance too early, we may well hinder future modifiability. How do we keep track of NFRs and their interactions, while selecting among design alternatives to meet the requirements?

The NFR Framework offers explicit representation of non-functional requirements, systematic use of catalogued methods of architectural design knowledge, management of tradeoffs among architectural design alternatives through catalogued correlations, and evaluation of softgoal achievement for a particular choice of architectural design using the evaluation procedure.

The process of using the NFR Framework is intended not only as a means for supporting the design of software architecture but also for providing a graphical record for later review, justification, change and reuse. In this regard, the NFR Framework has also been applied to dealing with change [Chung95a, 96].

Software Architecture: Concepts and Approaches

We draw on concepts, such as *elements*, *components*, and *connectors*, that have been identified as essential to portray architectural infrastructure, as advocated by Perry and Wolf [Perry92], Garlan and Shaw [Garlan93], Abowd, Allen, and Garlan [Abowd93], Callahan [Callahan93], Mettala and Graham [Mettala92]. We also draw on earlier notions on information system architecture by Zachman [Zachman87]. In our view, an emphasis on NFRs is complementary to efforts directed towards identification and formalization of concepts for functional design.

Concerning the role of NFRs, design rationale, and softgoal assessment, the proposal by Perry and Wolf [Perry92] is of close relevance to our work. Perry and Wolf propose the use of architectural style for constraining the architecture and coordinating cooperating software architects. They also propose that rationale, together with elements and form, constitute the model of software architecture. In the NFR Framework, properties of the architectural form can be justified with respect to their positive and negative contributions to the stated NFRs, and relationships of the architectural form can be abstracted into contributions and softgoal labels, which can be interactively and semi-automatically determined.

Kazman et al. [Kazman94] propose a basis (called SAAM) for understanding and evaluating software architectures and give an illustration using modifiability. This proposal is similar to the NFR Framework, in spirit, as both take a qualitative approach, instead of a metrics-based approach. In a similar vein, but oriented towards software reuse, Ning et al. [Ning93] propose an approach (called ABC), in which they suggest the use of NFRs to evaluate the architectural design, chosen from a reuse repository of domain-specific software architectures, which closely meets very high-level requirements. Both SAAM and ABC are product-oriented, i.e., they use NFRs to understand or evaluate architectural *products*. We, however, take a process-oriented approach, providing support for systematically dealing with NFRs *during* the process of architectural design.

The NFR Framework [Chung93a] [Mylopoulos92a] aims to improve software quality [Chung94a,b] and has been studied [Chung95b] for a variety of information system types with a variety of NFRs, including accuracy, security and performance.

In our view, there are parallels to NFR-related work on information systems, and the work of [Chung95c,d], presented in this chapter, is one of the first that considers adaptation of the NFR Framework specifically in the context of software architecture.

13.2 CATALOGUING SOFTWARE ARCHITECTURE CONCEPTS USING THE NFR FRAMEWORK

An important step in using the NFR Framework is to obtain and organize knowledge of a specific NFR, e.g., performance, from both academic research and industrial experience, and then record and organize it in method catalogues. In our studies of NFRs for information systems [Chung95a,b, 96], some of which are described in Part III of this book, we found that the power of the NFR Framework can be increased if the models and tools are used not only to operate on the knowledge pertaining to the case at hand (e.g., redesigning a particular process), but also to bring to bear experiences accumulated in generic knowledge (e.g., methods for achieving fast turnaround, for assuring security, etc.), or case-based knowledge (e.g., in redesigning other operations). Once a more organized catalogue of architectural methods is developed, it could be used in studies of using the NFR Framework to deal with architectural design.

Along with feedback from industrial and academic experts, such studies would enhance the coverage of the NFR Framework and evaluate its usefulness for architectural design.

Treating NFRs as Softgoals

In our process-oriented approach, non-functional requirements, such as “modifiable system” and “good system performance,” are explicitly represented as softgoals to be addressed and satisfied during the process of architectural design. Each softgoal (e.g., $\text{!Modifiability}[\text{System}]\{\text{critical}\}$) is associated with a type (e.g., Modifiability), a topic list (e.g., System), and an optional priority (e.g., critical).

One fundamental premise of the NFR Framework is that NFR softgoals can interact with each other, in conflict or in synergy. This property is used to systematically guide selection among architectural design alternatives and to rationalize the overall architectural design process.

NFR Types

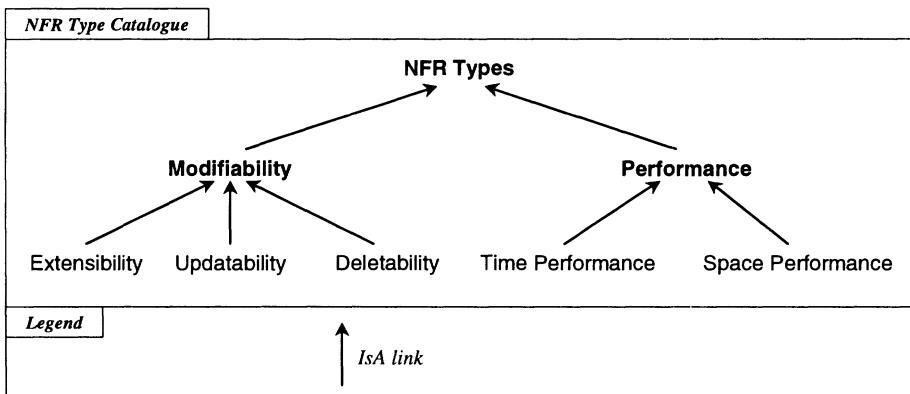


Figure 13.1. Catalogue of some NFR Types considered for software architecture.

Figure 13.1 provides a catalogue of some NFR Types which are considered for software architecture. This figure is an extension of the NFR Types presented in Figure 3.2.

NFRs such as modifiability and comprehensibility have not been treated using the NFR Framework in the detail that has been done for accuracy, security and performance in Part II.

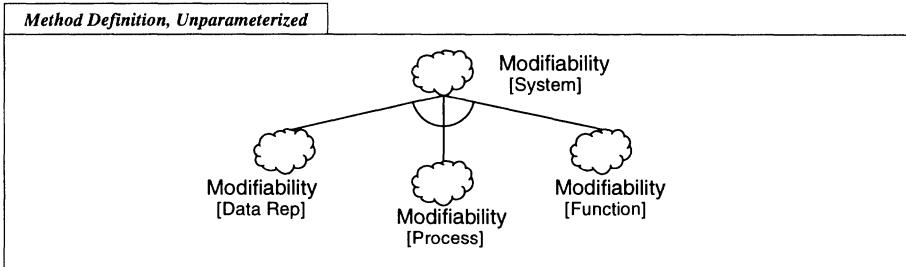


Figure 13.2. A method for refining a modifiability softgoal for the system.

The type catalogue organizes our general knowledge about modifiability. It shows sub-types of modifiability. From this information, we can generate type decomposition methods, such as:

In order to satisfy modifiability, one needs to satisfy extensibility, updatability, and deletability.

Methods

Architectural design knowledge and experience about specific NFRs can be organized into methods and made available to the software architect through systematic search.

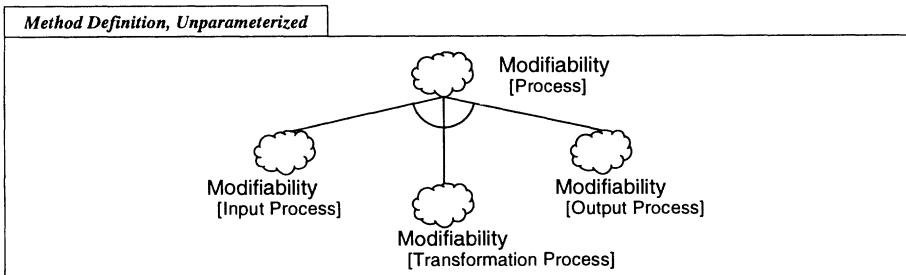


Figure 13.3. A method for refining a modifiability softgoal for the process.

Figure 13.2 shows a topic decomposition method. Modifiability considerations for a system are decomposed into concerns for data representation, processes and functions.

Correlation Catalogue					
Contri- bution of offspring Operation- alizing Softgoal	to parent <i>NFR Softgoal</i>				
	Modifiability [Process]	Modifiability [Data Rep]	Space [System]	Time [System]	Reusability [System]
Shared Data [Target System]	BREAKS	BREAKS	MAKES		HURTS
Abstract Data Type [Target System]	HURTS	HELPS		HURTS	HELPS
Implicit Invocation [Target System]	HELPS	HURTS	HURTS	BREAKS	?
Pipe & Filter [Target System]	HELPS	BREAKS	BREAKS WHEN cond1		HELPS

cond1: size of data in domain is huge

Figure 13.4. A generic Correlation Catalogue, based on [Garlan93].

Figure 13.3 shows a method which decomposes the topic on process, including algorithms as used in [Garlan93]. Decomposition methods for processes are also described in [Nixon93, 94a, 97a], drawing on implementations of processes [Chung84, 88].

These two method definitions are unparameterized. A fuller catalogue would include parameterized definitions too.

Operationalization methods, which organize knowledge about satisfying NFR softgoals, are embedded in architectural designs when selected. For example, an **ImplicitFunctionInvocationRegime** (based on [Garlan93], architecture 3) can be used to hide implementation details in order to make an architectural

design more *extensible*, thus contributing to one of the softgoals in the above decomposition.

Argumentation methods and templates are used to organize principles and guidelines for making design rationale for or against design decisions (Cf. [J. Lee91]).

Correlations

Knowledge and experience about tradeoffs among architectural design alternatives can be organized into correlation rules and made available to the software architect through systematic search. Once stated and organized, correlation rules can be browsed by the software architect in selecting among architectural alternatives. Correlations can then be specialized to record domain-specific softgoal conflict and synergy, omissions and redundancies.

Figure 13.4 shows a correlation catalogue based on the presentation by Garlan and Shaw [Garlan93], which compared the extent to which alternative target solutions (operationalizations) address design considerations. In our adaptation of the notion of “satisficing,” entries in Figure 13.4 reflect contributions by architectural design alternatives (operationalizing softgoals) for or against NFR softgoals. An empty entry indicates a lack of significant contribution. As it stands, the catalogue is applicable to a variety of KWIC systems. This generic catalogue can be extended to incorporate more tradeoff knowledge or tailored to the needs of the intended application domain. An entry with “?” means an UNKNOWN contribution. To deal with this, the software architect will have to consider the characteristics of the application domain.

13.3 ILLUSTRATION OF THE ARCHITECTURAL DESIGN PROCESS

Let us now illustrate the architectural design process.

Initial NFR Softgoals

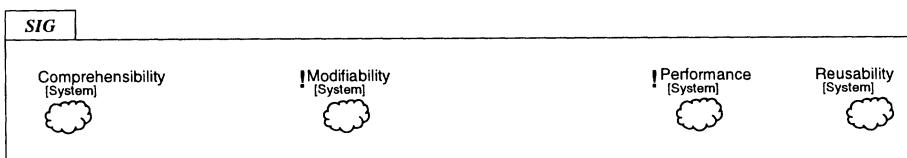


Figure 13.5. Initial NFR Softgoals for a KWIC system.

We consider an assumed initial set of main NFR softgoals which indicate that “the system should be modifiable, have good performance, and be reusable and

comprehensible, with modifiability and performance being priorities.” The software architect can represent these by $\text{!Modifiability}[\text{System}]$, $\text{!Performance}[\text{System}]$, $\text{Reusability}[\text{System}]$ and $\text{Comprehensibility}[\text{System}]$. They are shown in Figure 13.5.

To simplify presentation in this chapter, the kind of priority (e.g., “`{critical}`”) is often omitted. In addition, priorities are often directly attached to softgoals, without showing prioritization refinements such as:

$\text{!Modifiability}[\text{System}] \text{ MAKES } \text{Modifiability}[\text{System}]$

The softgoal topics are fairly broad here, representing the overall system. Softgoal refinements can make the softgoal topics more specific. Refinements can also introduce performance layers (discussed in Chapters 8 and 9) as an additional topic for softgoals. Let’s see how the initial NFR softgoals can be refined.

Refining Softgoals using Methods

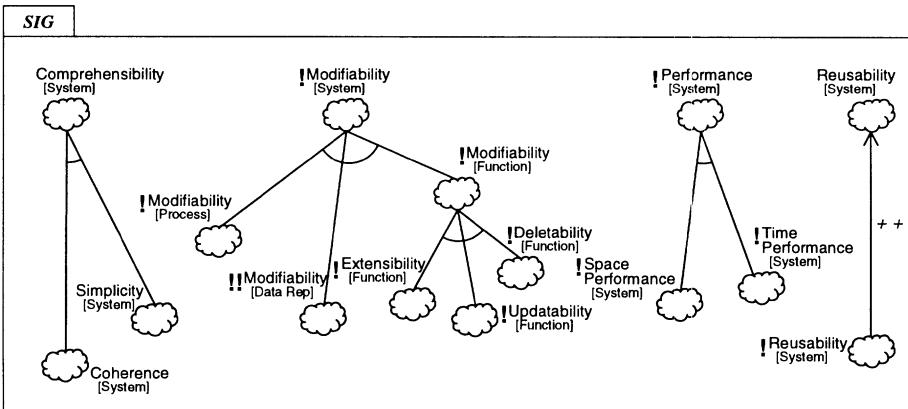


Figure 13.6. Refining softgoals using methods.

After stating NFRs as NFR softgoals, the software architect attempts to *refine* them (Figure 13.6). This can involve *decomposing* softgoals or *clarifying* (*disambiguating*) them.

Decomposition can, for example, involve refining a softgoal for a module into softgoals for components of the module.

Refinement can also clarify (disambiguate) softgoals, since they can mean many different things to different people and are too coarse to be put under analysis concerning their interactions. For example, security has different

shades of meaning in industrial and military contexts; and performance can be decomposed into time and space considerations.

The software architect can refine a softgoal either on its type or on its topic. The architect focuses on decomposing $\text{!Modifiability}[\text{System}]$ on its topic.

Softgoals can also be refined on their priority, based on the criticality of the softgoal, or the dominance of the softgoal's topic in the workload. Figure 13.6 shows the prioritization:

$\text{!Reusability}[\text{System}]\{\text{critical}\} \text{ MAKES } \text{Reusability}[\text{System}]$

As discussed above, other prioritizations are shown in the figure without using refinements. These include:

$\text{!TimePerformance}[\text{System}]\{\text{critical}\} \text{ MAKES } \text{TimePerformance}[\text{System}]$
 $\text{!!Modifiability}[\text{DataRep}]\{\text{veryCritical}\} \text{ MAKES } \text{Modifiability}[\text{DataRep}]$
 $\text{!SpacePerformance}[\text{System}]\{\text{dominant}\} \text{ MAKES } \text{SpacePerformance}[\text{System}]$

After browsing methods in consultation with domain experts, the architect might refine the softgoal into three other offspring softgoals: for modifiability of the process, modifiability of the data representation, and modifiability of the function. These softgoals are written $\text{!Modifiability}[\text{Process}]\{\text{critical}\}$, $\text{!!Modifiability}[\text{DataRep}]\{\text{veryCritical}\}$, and $\text{!Modifiability}[\text{Function}]\{\text{critical}\}$ (Figure 13.6). This topic decomposition method draws on the work by Garlan and Shaw [Garlan93], who consider changes in processing algorithm and changes in data representation, and by Garlan et al. [Garlan92], who further consider enhancement of system function.

The software architect further refines $\text{!Modifiability}[\text{Function}]$, this time on its type, into $\text{!Extensibility}[\text{Function}]$, $\text{!Updatability}[\text{Function}]$, and $\text{!Deletability}[\text{Function}]$. This type decomposition method draws on work by Kazman et al. [Kazman94], who consider extension of capabilities in terms of adding new functionality, enhancing existing functionality, and deleting unwanted capabilities.

Similarly, the software architect refines $\text{!Performance}[\text{System}]$ on its type into softgoals $\text{!SpacePerformance}[\text{System}]\{\text{dominant}\}$ and $\text{!TimePerformance}[\text{System}]\{\text{critical}\}$, using a SubType method which draws on work on performance requirements from Nixon [Nixon93, 94a, 97a] (See Chapters 8 and 9). Further refinements can address the system's response time to users, using Smith's principles for building performance into systems [C. Smith90].

Operationalizations, Prioritization and Design Rationale

In moving towards a target system, the software architect considers the four operationalizing softgoals (architectural design alternatives) from [Garlan93] that were described earlier. These are shown at the bottom of Figure 13.7, which also shows tradeoffs among the operationalizations.

The software architect has a number of softgoal conflicts and synergies to deal with.

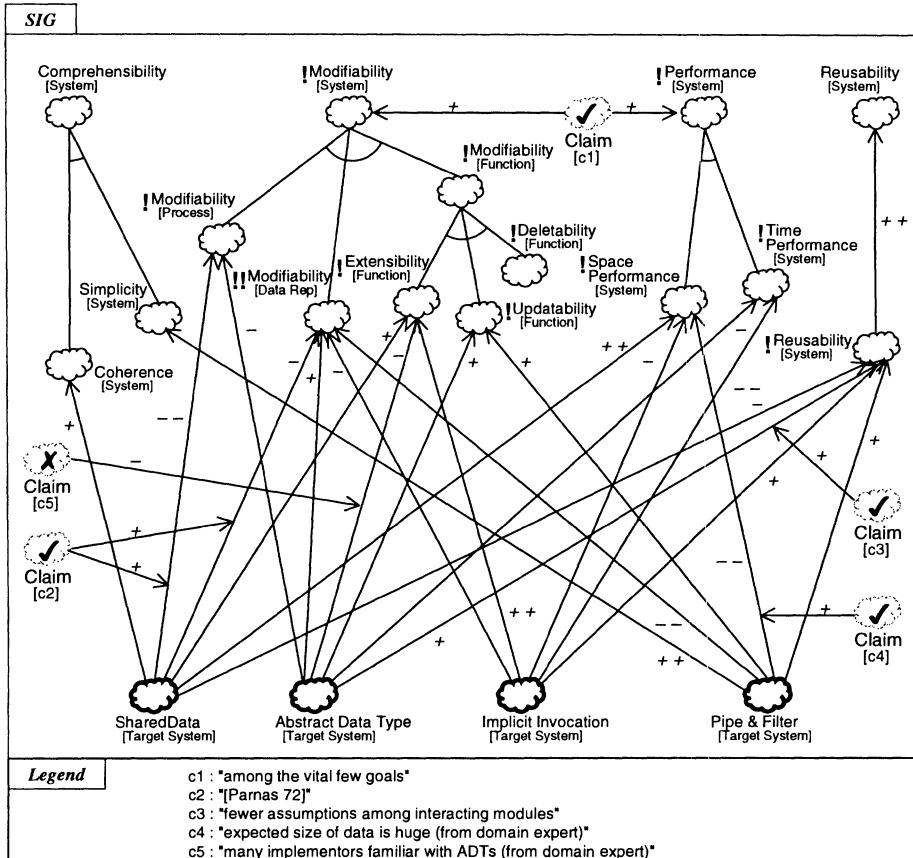


Figure 13.7. Tradeoffs among operationalizations.

To handle the situation, the architect prioritizes softgoals, as **nonCritical**, **Critical**, **veryCritical**, **nonDominant**, **Dominant** and **veryDominant**. This decision can be justified by way of design rationale. For example, treating modifiability, performance, and reusability as priority softgoals can be supported, via the **VitalFew** argumentation template, possibly with a user survey.

With the prioritization, the software architect can put emphasis on high-priority softgoals, and readily resolve softgoal conflict. For example, as **!!Modifiability[Data Rep]{veryCritical}** is considered very important, architectural design alternatives which strongly hurt the softgoal might be eliminated from further consideration, here **Shared Data** and **Pipe & Filter**.

Design rationale can come from several sources, including the literature. For example, an argument may simply be a citation, such as Claim["[Parnas72]"] which supports that the Shared Data scheme strongly hurts both Modifiability[Process] and Modifiability[Data Rep].

Resolving Tradeoffs using Domain Information

Architectural design alternatives make partial contributions for or against NFR softgoals which can be synergistic or in conflict with one another. Correlations can be used to generate new interdependencies between existing softgoals, as well as to suggest the generation of new softgoals, hence making tradeoffs explicit.

Correlation Catalogue					
Contri- bution of offspring <i>Operation- alizing Softgoal</i>	to parent <i>NFR Softgoal</i>				
	Modifiability [Process]	Modifiability [Data Rep]	Space [System]	Time [System]	Reusability [System]
Shared Data [Target System]	BREAKS	BREAKS	MAKES		HURTS
Abstract Data Type [Target System]	HURTS	HELPS		HURTS	HELPS
Implicit Invocation [Target System]		HURTS	HURTS	BREAKS	HELPS
Pipe & Filter [Target System]		BREAKS	BREAKS		HELPS

Figure 13.8. Domain-Specific Correlation Catalogue for KWIC Example.

The software architect takes the generic correlation catalogue (Figure 13.4) and *specializes* it into a domain-specific correlation catalogue (Figure 13.8). During specialization of correlation rules (Figure 13.8), uncertainties in the generic correlation catalogue (e.g., *UNKNOWN* entries ("?")) should be resolved. For example, the software architect could consult a domain expert who knows about the characteristics of the intended application domain to determine if the **Pipe and Filter** would significantly hinder the **SpacePerformance** softgoal. Once obtained, domain characteristics can be used as an argument in a SIG (e.g., *Claim*[“expected size of data is huge (from domain expert)”), as shown in Figure 13.7]. This also eliminates from Figure 13.8 the condition *cond1*, which was in Figure 13.4. In addition, contributions to **Modifiability[Process]** are changed here. Similarly, **Implicit Invocation** and **Pipe & Filter** do not *HELP* **Modifiability[Process]** here, as shown in the domain-specific correlation catalogue (Figure 13.8) and the SIG (Figure 13.7).

The software architect has specialized the correlation rules using domain information. This led to both positive and negative contributions, which the software architect examined. Some were rejected or tailored, and justifications were provided. Such catalogues can help identify tradeoffs, which may be somewhat difficult to see when the number of correlation contributions is high, as is the case in of Figure 13.7.

The resulting correlations are applied to the SIG in Figure 13.7. Note that correlation links are not shown with dashed lines in Figures 13.7 and 13.9. However, their impacts are shown.

Selection Among Alternatives

A particular architectural design can make a positive or negative contribution to an NFR softgoal, or make no contribution at all. For example, the use of an abstract data type may help updatability, but at the cost of poorer time performance (Figure 13.9). Hence, *selecting* an architectural design requires careful examination of the degree of satisficing softgoals, particularly priorities.

Figure 13.9 shows the selection of one operationalization, **AbstractDataType[TargetSystem]**, the use of an abstract data type in the target system. At the bottom of the figure, this selected operationalization is labelled as *satisficed*, while the rejected ones are *denied*.

This kind of approach is made possible by earlier softgoal reduction as an application of the “divide-and-conquer” paradigm. Refinement, for decomposition and disambiguation, has facilitated systematic organization of and search for NFR-related reusable knowledge, clearer understanding of tradeoffs, and conflict resolution with design rationale which reflects the needs and characteristics of the intended application domain.

Evaluation

Figure 13.9 shows the impact of the decisions upon the main softgoals. The use of abstract data types supports modifiability of data representation, which is

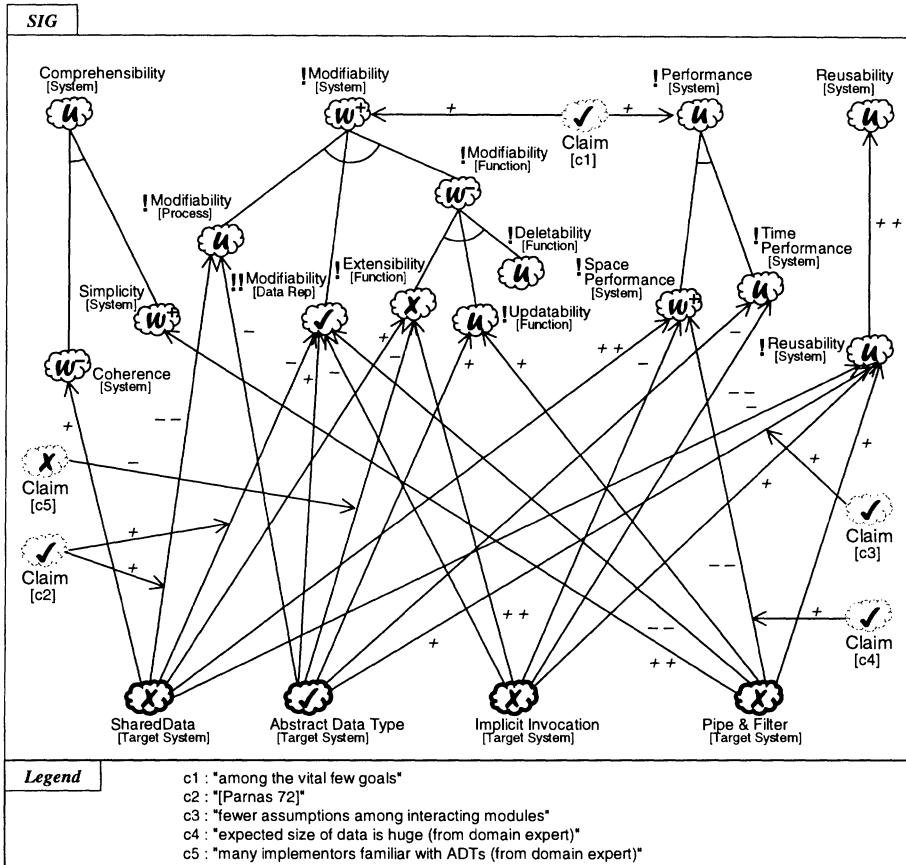


Figure 13.9. Evaluating the impact of the chosen alternative on NFRs.

very critical, but hinders process modifiability, which is critical. The decisions give weak positive contributions to the dominant space performance softgoal. However, further work will be needed to determine the impact on the critical time performance softgoal, which currently receives conflicting contributions. In addition, extensibility has been denied.

A slightly more fine-grained evaluation scheme is proposed in [Chung95a, 96].

13.4 DISCUSSION

Software architecture is becoming an increasingly important topic in software engineering. Software systems are increasingly distributed, open, and constantly changing. NFRs need to be addressed at the architectural level.

This chapter has presented an application of the NFR Framework, building quality into a software system during the architectural design phase. This helps to systematically guide selection among architectural design alternatives, thus providing an alternative to an *ad hoc* approach. We see three aspects of this.

First, our application of the NFR Framework is intended to improve the software architect's ability to understand the high-level system constraints and the rationale behind architectural choices, to reuse architectural knowledge concerning NFRs, to make the system more changeable, and to analyze the design with respect to NFR-related concerns. Our approach facilitates codification of knowledge about NFR-related architectural design and tradeoffs, and systematic management and use of such knowledge.

Second, this application emphasizes collecting, organizing and using information, such as knowledge about the particular domains and systems, their workloads and priorities, to meet the architectural quality needs of the particular systems being developed.

Finally, throughout the process of architectural design, it is the the software architect who is in control.

In the context of architectural design, this application is only preliminary, with its use illustrated only on an instructional example. Once a more codified catalogue of architectural methods is developed, it could be used in studies of using the framework to deal with architectural design.

A variety of architectural case studies are needed to gain experience and expert feedback on benefits and weaknesses and to see whether this approach can be effectively applied to a variety of real, large-scale industrial-strength domain-specific software architectures, application frameworks, and reference architectures.

We have only illustrated the application of the NFR Framework to selecting among architectural design alternatives at a very abstract level. An important aspect of future work is to deal with more complex architectural problems, and to show the scalability of the NFR Framework. This of course requires codification of current and future knowledge about architectural alternatives and design criteria. In addition, studies should be conducted using a rigorous methodology allowing the usefulness of the framework to be confirmed or denied.

The results, we trust, would be the provision of a more satisfactory approach to building NFRs into software architecture.

13.5 LITERATURE NOTES

This chapter is based on a paper by Lawrence Chung, Brian Nixon and Eric Yu, in the Proceedings of the First International Workshop on Architectures for Software Systems [Chung95c], and a revision of that paper which appeared in the CD-ROM Supplement to the Proceedings of CASCON'95 [Chung95d].

A recent paper dealing with software architecture [Chung99] draws on the source papers of this chapter [Chung95c,d], the NFR Framework, and the *i** Framework ([Yu94b], see Chapter 14).

Architecture and quality has been considered for data warehouses [Jarke98]. In addition to software quality, that paper considers data quality.

14 ENTERPRISE MODELLING AND BUSINESS PROCESS REDESIGN

14.1 INTRODUCTION

Applications of the NFR Framework are not limited to the development of software systems. In this chapter, we apply the NFR Framework to enterprise modelling and business process redesign.

This application makes use of several concepts from the NFR Framework. It uses the concept of *softgoal*, and reasoning techniques that are based on *satisficing*. The *softgoal interdependency graph* is used to capture and support the design process and reasoning. In addition, *methods* and *correlations* are *catalogued* and *applied* as a developer faces particular design decisions.

The approach is illustrated with an hypothetical example, taken from the literature.

Effective processes are crucial to the success of any organization. There has been ongoing effort to find better ways to improve work processes in organizations, e.g., to achieve faster cycle times, better quality products, high customer satisfaction, lower cost, and greater flexibility. The availability of advanced information technologies has provided a major impetus to many organizations to rethink their work processes. Instead of making incremental, quantitative improvements and optimizations, it has become common to seek dramatic improvements in performance by making major, structural changes, often involving the use of information technology. This has come to be known as *business process reengineering* [Hammer90, 93] [Davenport93].

To achieve these changes, Hammer advocates asking fundamental questions about how work is currently done, possibly challenging long-held assumptions, and understanding why things are done the way they are. A lack of care during design can cause a great deal of upheaval in an organization, without achieving the desired gains [Davenport94] [Hammer95].

One of the challenges is therefore to understand the implications of *target techniques (operationalizations)*. The use of appropriate kinds of models can assist the formulation of suitable questions and answers. This can help to systematically guide developers towards beneficial techniques.

The *i** Framework

The *i** Framework has been developed to support the modelling, analysis, and redesign of organizations and business processes [Yu94d]. The name *i**, pronounced “i-star,” stands for *distributed intentionality*. In the *i** Framework, organizations are viewed as consisting of social *actors* who depend on each other for *softgoals* to be satisfied, *goals* to be achieved, *tasks* to be performed, and *resources* to be furnished. The *i** Framework includes a *Strategic Dependency Model* for describing the network of relationships among actors, and a *Strategic Rationale Model* for describing and supporting the reasoning that each actor has about its relationships with other actors.

The *i** Framework makes use of the NFR Framework’s concepts, notably *softgoals*, which are used both in the description of business processes, and in assisting the redesign of processes.

Intentional elements consist of softgoals, goals, tasks and resources.

Enriching Process Descriptions with Strategic Relationships and Dependencies

Many kinds of models have been proposed and used for describing (or “mapping”) business processes [Curtis92] [Mylopoulos98]. For example, it is common to use systems analysis techniques such as structured analysis (e.g., SADT [Ross77], DFD [DeMarco78]) and Entity-Relationship Models [Chen76], which focus on the modelling of activities and entities. While these are important for systems development, they offer little help in the search for target techniques to address business problems. Most existing process models have been designed for describing *what* an organization is like, but they cannot express *why* the organization is the way it is. The motivations, intents and rationales behind the activities and entities of the organization are missing from these models.

In particular, these models fail to capture the *strategic relationships* among organizational stakeholders. The *i** Framework focusses on the modelling of strategic relationships. It considers such relationships to be “strategic” in the sense that each party is assumed to be concerned with opportunities and vulnerabilities, and seeking to protect or further its interests. Of course, in reality there are different possible strategic relationships. For example, one may seek to protect the interests of others, or not take advantage of others.

In modelling a process, a developer describes how organizational actors depend on each other for *softgoals* to be satisfied, *goals* to be achieved, *tasks* to be performed, and *resources* to be furnished. By using *intentional concepts* such as softgoals, abilities, beliefs, and commitments, the modelling framework allows a much more realistic characterization of a work process that is inhabited by real-life actors (as opposed to abstract, depersonalized activities).

However, in introducing the concept of relationships among actors, an important question arises as to the notions of “goal” that are needed to capture realistic kinds of relationships. For one notion of goal, the criteria for success and failure is sharply-defined *a priori*. This notion of goal is widely used in artificial intelligence in automated reasoning (e.g., [Nilsson71]). However, another frequently encountered kind of relationship is the *softgoal* of the NFR Framework, which has a qualitative aspect, and can be elaborated upon and clarified as it is refined. The i^* Framework incorporates both notions of *goal* and *softgoal*.

Supporting Process Design Reasoning by Adapting the NFR Framework

While the significance of models for describing processes is widely recognized, the task of coming up with possible techniques during redesign is seldom supported by models. The use of appropriate models, however, can be very helpful in the systematic exploration of the space of possible techniques, and in reasoning about their relative merits and pitfalls. In particular, a goal-oriented approach provides an excellent way of focussing the search for good designs. Requirements such as faster cycle time, higher customer satisfaction, greater flexibility, and higher employee morale, are often best treated as softgoals. In this way, tradeoffs are considered as possible techniques are explored. The NFR Framework therefore offers a good foundation for supporting process design reasoning.

Several adaptations of the NFR Framework are needed. In a process design context, multiple actors are involved in the eventual execution of the process. In addition, these actors, and possibly others, are also *stakeholders* during the design process. The design reasoning, therefore, has to reflect the goals and interests of these stakeholders.

Most process designs now occur within a context of existing processes. That is, a developer is more likely to be doing a *redesign* rather than designing from scratch. A developer needs to understand the existing process, discover what its problems are, then redesign it while at the same time exploiting new opportunities. Thus a developer might not start with top-level softgoals and proceed primarily in a top-down fashion. A developer is more likely to start with an existing process and organization. Developers may uncover increasingly fundamental goals as they pursue answers to ever deeper “why” questions. However, at any level of questioning, they may also begin to seek alternative means to achieving goals by considering *how else* those goals can be achieved.

Thus the redesign process may proceed in a up-and-down sequence along a means-ends hierarchy.

Processes must include actual operations, and not just descriptions of how well those operations are carried out. Thus the i^* Framework draws on the NFR Framework's treatment of both functional and non-functional requirements. Non-functional and functional aspects of process description and design reasoning are used in the i^* Framework.

Example: Handling Insurance Claims

We illustrate the approach using an insurance claims example. In the area of claims processing, insurance companies may no longer be content with information systems that simply automate well-established process steps. Instead, information systems may be viewed as part of a thoroughly redesigned claims handling process. Questions that might drive the redesign effort might include:

- Why does it take so long to have a claim settled after an automobile accident?
- Why does the company hire appraisers to assess damages?
- How else can claims be settled?
- What other concerns would arise if new ways of handling claims are adopted?

Now we consider the nature of strategic information about organizations, using the insurance claims example. Consider an insurance company which wants to minimize payments to claimants, and for this reason hires appraisers to keep repairs, and therefore payments, to the necessary minimum. At the same time, the insurance company wants to keep customers happy so that they will continue to renew their policies. Car owners want repair damages to be assessed fairly, and may ask body shops to give repair estimates that maximize the insurance payment. It can be seen that the various parties have different strategic interests. These interests help determine what information is collected and used by the claims representative (e.g., accident particulars, witness statements) and the appraiser (e.g., photographs of damage, multiple repair estimates).

14.2 THE STRATEGIC DEPENDENCY MODEL

A Strategic Dependency Model is a graph. Nodes in the graph can represent *actors*. Each link between two actors indicates that one actor *depends* on the other for something in order that the former may attain some softgoal, goal, resource or task. We call the depending actor the *depender*, and the actor who is depended upon the *dependee*. The object around which the dependency relationship centres is called the *dependum*. These are shown in the legend of Figure 14.1. Each *dependency link* is shown as “–D–.” The dependum can be a softgoal, goal, task or resource.

In Figure 14.1, the depender and dependee are actors. More generally in the Strategic Dependency Model, *dependencies* relate actors, softgoals, goals,

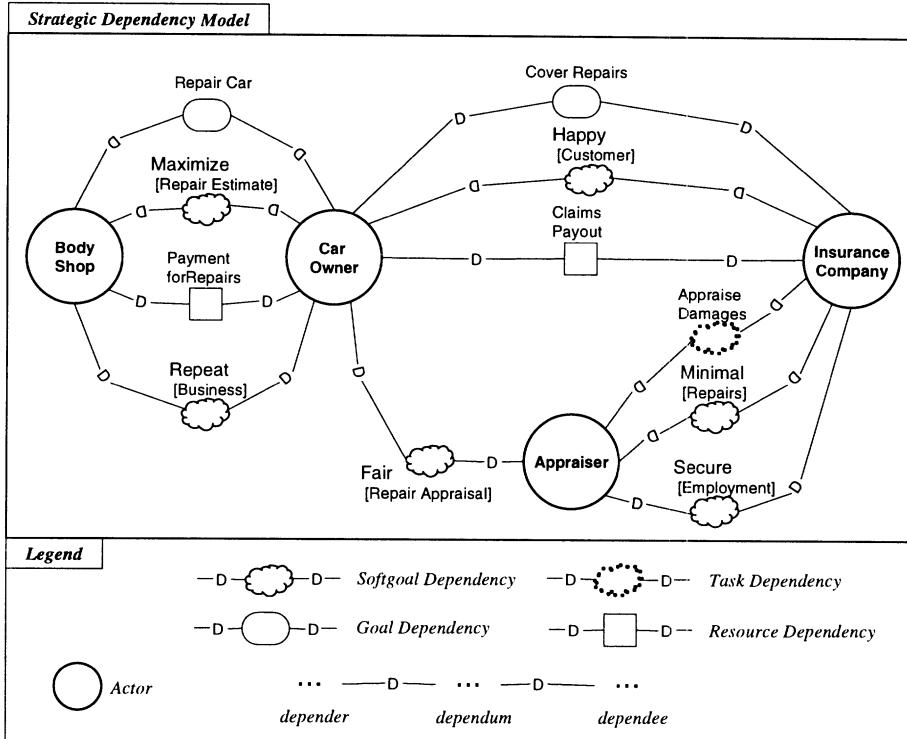


Figure 14.1. Strategic Dependency Model for existing auto insurance claims handling.

tasks and resources. That is, the dependee can be any of an actor, softgoal, goal, task or resource, and so can the dependee.

When depending on another actor for a dependum, an actor is *able* to achieve softgoals, goals, tasks or resources that it is otherwise unable to achieve, or is able to achieve them not as easily or not as well. At the same time, the dependee becomes *vulnerable*. If the dependee fails to deliver the dependum, the dependee would be adversely affected in its ability to achieve its *intentional elements* (i.e., softgoals, goals, tasks or resources).

For example, car owners can have their cars repaired by body shops, even if car owners do not have the ability to do the repairs themselves. However, they are vulnerable to the cars not being properly repaired.

The model provides four kinds of dependencies, shown in the legend of Figure 14.1. They are: *softgoal dependency*, *goal dependency*, *task dependency*, and *resource dependency*. They are distinguished by the kind of dependum that is allowed in the relationship between dependee and dependee.

Figure 14.1 shows a Strategic Dependency Model for an existing automobile insurance business claims handling configuration. The actors, such as the car owner and insurance company, are shown as circles, and are related by various kinds of strategic dependencies. Each “–D–” in the figure shows the direction of the dependency, from depender to dependee.

The insurance company wants to offer good service to the customer in order to keep the business (*Happy[Customer]*). This is a softgoal dependency. As usual, the *softgoal* has an *NFR type*, and a *topic*.

To maintain profitability, the company depends on appraisers to appraise damages so that only the minimal necessary repairs are approved, *Minimal[Repairs]*. This is a softgoal dependency, since profitability is a softgoal. In addition, the company depends on appraisers to appraise damages. This is a task dependency, since appraisal is a task.

The car owner depends on the insurance company to reimburse for the repairs from an accident (*ClaimsPayout*). This is a resource dependency. For this, car owner pays insurance premium in order to have coverage (*CoverRepairs*). This is a goal dependency.

The car owner depends on the claims appraiser for a fair appraisal. However, the appraiser might be viewed as acting in the interests of the insurance company because the appraiser depends on the insurer for continued employment. The car owner, in turn, might depend on the body shop to give an estimate that maximizes the car owner’s interests, since the body shop depends on the car owner for repeat business. An analysis of strategic dependencies at a more detailed level would reveal the role of information and information systems in these relationships [Yu93b].

Existing information systems that support the claims process have this kind of understanding embedded, but likely only as implicit assumptions. These assumptions are seldom made explicit during information system development because existing modelling techniques generally do not encourage or support the modelling of relationships that involve intentional concepts. Without this deeper understanding, it is difficult to change information systems to meet changing needs, as evidenced by the problem of “legacy systems.”

Change is rapid in business environments, and recent management concepts, such as business reengineering, are being used. As a result, existing relationships, business patterns and assumptions are re-examined and are often reconfigured, sometimes dramatically.

Hammer and Champy [Hammer93, pp. 136–143] describe an hypothetical scene in which a process redesign team explores new techniques to improve claims handling of an automobile insurance business. Since a small claim may cost almost as much to process as a large claim, one way to reduce administrative costs is to reduce insurance company involvement in dealing with small claims. “Let the insurance agent handle small claims,” it was suggested. The insurance agent will make all the inquiries and payments, while the insurance company will concentrate on large claims that have more significant impact on profitability. Agents get to cement their relationships with customers, while

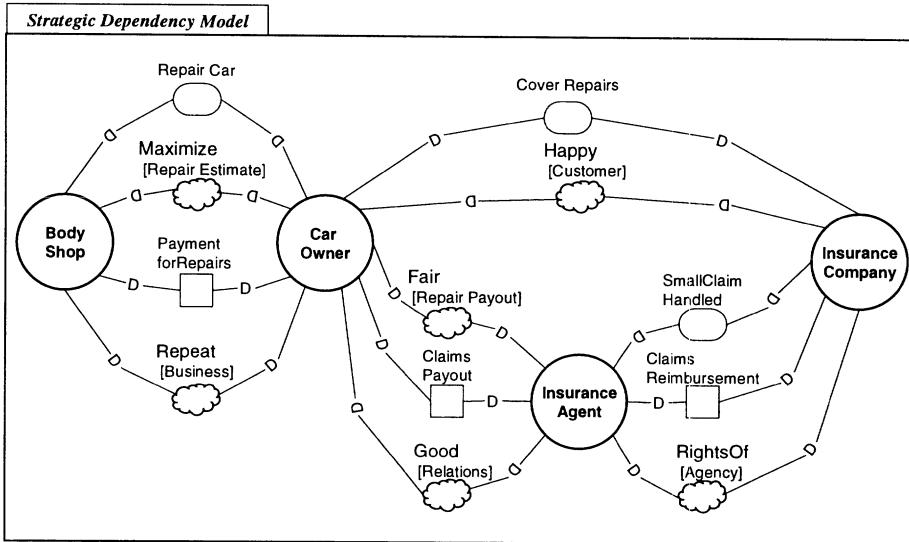


Figure 14.2. Strategic Dependency Model for letting the insurance agent handle claims.

customers are more likely to get fair hearings from their agents about a fair payment amount, it was argued. This would help keep the customer happy.

Figure 14.2 shows a Strategic Dependency Model for this new business process configuration. Needless to say, shifting the claims handling responsibilities to the insurance agent means that the information needs of the insurance agent are also significantly altered. Based on the new model of strategic dependencies, a developer could consider what information needs to be shared or sent among insurance agents and the insurance company, and how accurate and up-to-date they need to be.

A willingness to dispense with existing approaches to running an insurance business may lead to proposals which are dramatically different. “Let the body shop handle the claims,” someone else suggested. Currently, body shops are not likely to be on the side of the insurance company. For example, an insurance company might not want to pay according to a body shop’s repair estimates, since the body shop may charge too much, and be on the customer’s side. This is illustrated by the strategic dependencies in Figures 14.1 and 14.2.

However, for small claims, it may not be a bad idea to bypass the paperwork and help customers get their cars fixed as quickly as possible. This meets customers’ desire to have their cars fixed promptly, while significantly reducing administrative costs for the insurance company. However, this approach raises

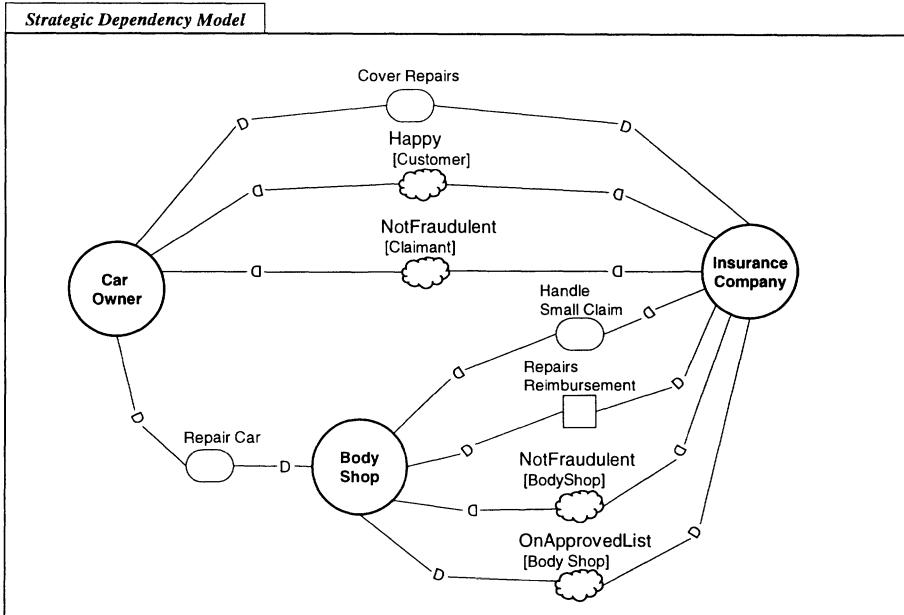


Figure 14.3. Strategic Dependency Model for letting the body shop handle claims.

concerns about possible fraud, which need to be addressed. Figure 14.3 shows the Strategic Dependency Model for this alternative.

The Strategic Dependency Model encourages a developer to obtain a deeper understanding of an organization and its business processes, beyond the usual understanding based on activities and entity flows. It helps a developer to identify what is at stake, for whom, and what impacts are likely if a dependency is not met. This is done by focussing on intentional dependencies among actors.

The Strategic Dependency Model can provide hints about why a process is structured in a certain way. However, it does not sufficiently support the process of suggesting, exploring, and evaluating alternative target techniques. That is the role of the Strategic Rationale Model.

14.3 THE STRATEGIC RATIONALE MODEL

A Strategic Rationale Model is a graph that describes the reasoning behind relationships. This includes relationships between actors, between tasks, and between actors and tasks. Thus it reveals the internal linkages that connect external strategic dependencies.

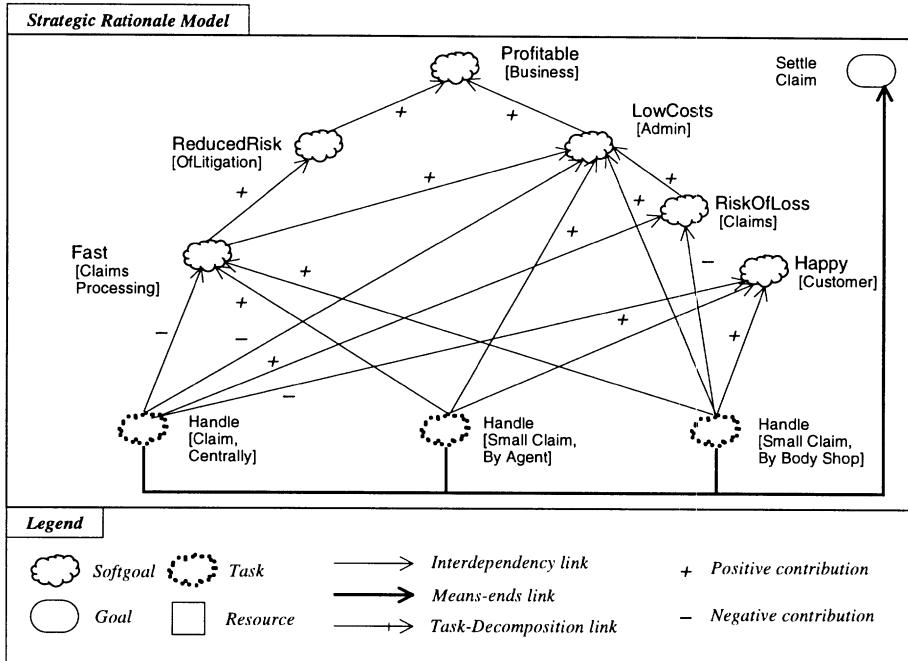


Figure 14.4. Strategic Rationale Model relating softgoals and tasks for insurance claims handling.

A Strategic Rationale Model contains nodes and links, which are shown in the Legend of Figure 14.4. There are four kinds of intentional elements (nodes): *softgoals*, *goals*, *tasks* and *resources*. There are three kinds of links: *interdependency links* from the NFR Framework, as well as *means-ends links* and *task decomposition links*.

A process is often depicted as a collection of activities with entity flows among them.¹ For example, a claims handling process would include such activities as verifying the insurance policy coverage, collecting accident information, determining who is at fault, appraising damages, and making an offer to settle.

Figure 14.4 shows the top softgoals and their refinements. This is quite similar to a *Softgoal Interdependency Graph (SIG)* of the NFR Framework, in which *softgoals* which are linked by *interdependencies*. Interdependency links capture *refinements* in the downward direction, and *contributions* upwards.

¹For a survey of process modelling, see [Curtis92].

Softgoals are refined into other softgoals. Contributions between softgoals are shown as “+” and “-.” Softgoals in the i^* Framework are drawn as NFR softgoals in the NFR Framework, since they work in basically the same way. Contributions in the i^* Framework are positive or negative; the NFR Framework has a larger number of contribution types.

In moving towards a target system, *tasks* (dotted-line dark clouds) in the i^* Framework appear where one has *operationalizing softgoals* (solid-line dark clouds) in the NFR Framework. Softgoals are refined into *tasks* in the i^* Framework, in the same way that softgoals are refined into *operationalizing softgoals* in the NFR Framework. Thus tasks contribute to softgoals in the i^* Framework, just as operationalizing softgoals contribute to NFR softgoals in the NFR Framework. As a reminder of this relationship, operationalizing softgoals of the NFR Framework and tasks of the i^* Framework are drawn similarly.

Figure 14.4 shows three target techniques (tasks) for operationalizing (realizing) the handling of claims. These are:

- *handling claims centrally*: This is the existing technique.
- *handling small claims by the insurance agent*: This is one proposed technique.
- *handling small claims by the body shop*: This is the other proposed technique.

Each technique makes an impact on the softgoals, such as **Happy[Customer]**, **Fast[ClaimsProcessing]**, and **Profitable[Business]**. For example, central claims handling can be slow, yet reduce the risk of loss, due to more standardized and thorough review of claims. Handling small claims by the insurance agent or body shop can be faster, and can help satisfy customer. Of course, handling claims by the body shop runs the risk of loss due to fraud. These positive and negative *contributions* to softgoals are shown in Figure 14.4. Softgoals are evaluated to be satisfied or denied, as described in Part II.

A *softgoal* does not necessarily have *a priori*, clear-cut criteria for satisfaction. Although some of these can be measured and quantified, a qualitative approach can be used at the stage of exploring the space of alternatives.

Softgoal refinements in this chapter often simultaneously change both the type and topic of the softgoal. This is permitted in the NFR Framework, but is not frequently used in the examples of this book.

As with the NFR Framework, the target system is shown at the bottom, and related to the *goal* (comparable to functional requirements in the NFR Framework) at the top right.

Figure 14.5 shows the Strategic Rationale Model for handling claims centrally. This model of the current configuration shows a decomposition of a task into offspring tasks. This is comparable to the decomposition of operationalizing softgoals into other operationalizing softgoals in the NFR Framework. In turn, an offspring task is decomposed into goals, which are decomposed into goals. All this is done within the *actor boundary* for claims handling.

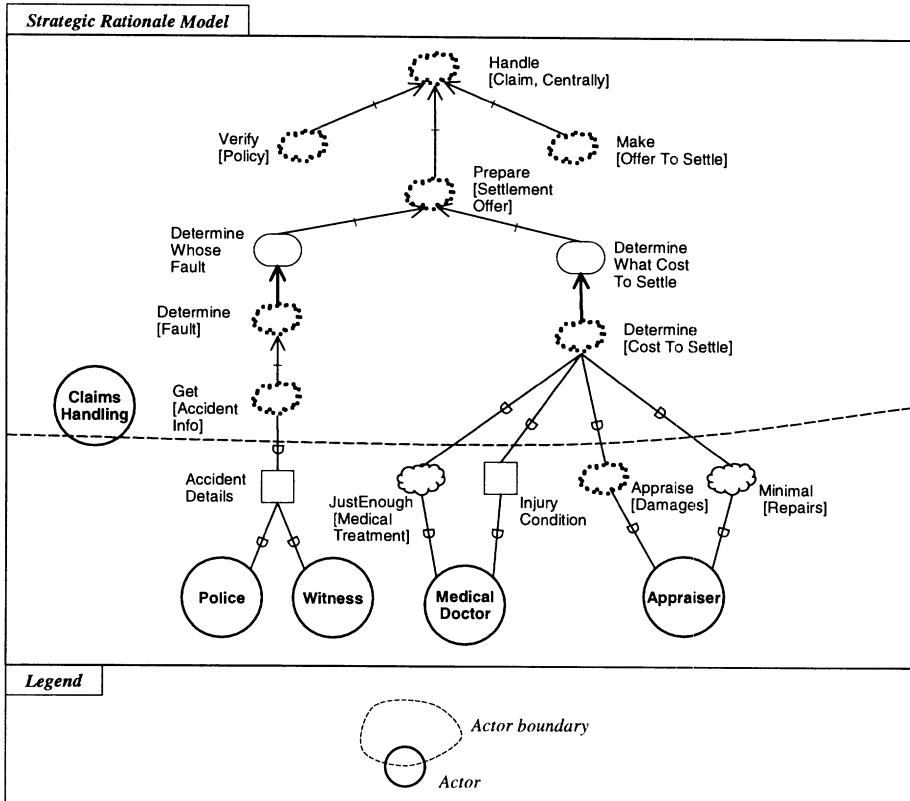


Figure 14.5. Strategic Rationale Model for handling claims centrally.

A variety of dependencies are then stated which relate claims handling to other actors, such as a witness and an appraiser. Note that the *dependencies* in Figure 14.5 (See Legend of Figure 14.1) relate tasks as well as actors. In addition, dependencies in the Strategic Rationale Model also relate softgoals and resources.

In the Strategic Rationale Model, the graph contains softgoals with interdependencies, as well as means-ends relationships and task decompositions (Figures 14.4 through 14.8). When a process element is expressed as a goal, this means that there might be different possible ways of accomplishing it. A task specifies one particular way of doing things (of accomplishing a goal), in terms of a decomposition into subtasks, softgoals, subgoals and resources. In seeking ways to redesign a business process, goals offer places to look for improvement. An ambitious redesign effort needs to discover and rethink high-level goals – by

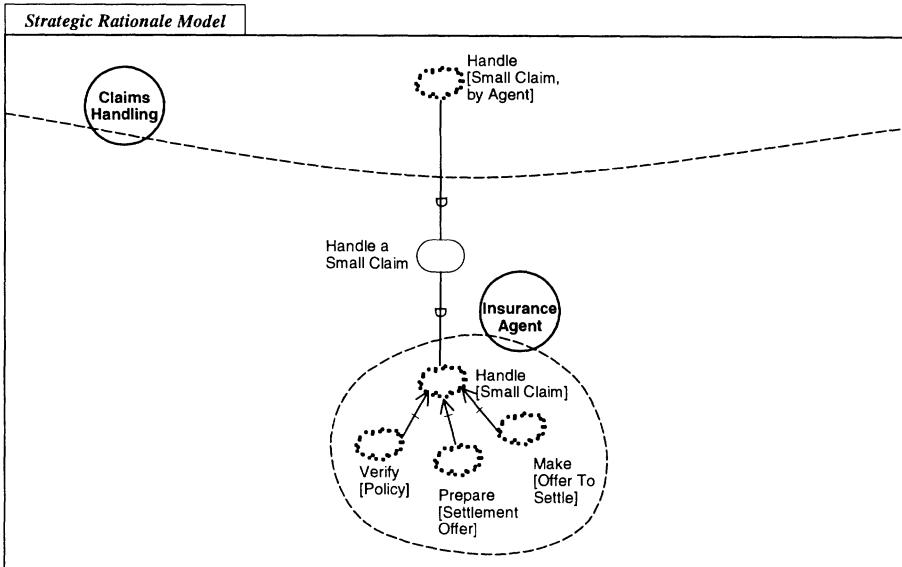


Figure 14.6. Strategic Rationale Model for handling small claims by the insurance agent.

asking “why” questions — rather than being content with solutions to low-level goals. Higher goals are discovered by asking “why” questions. Once sufficiently high-level goals have been identified, alternatives may be sought by asking “how else” the goals can be accomplished.

In the auto insurance example [Hammer93], the reengineering team wanted to consider possible techniques, by identifying a high-level *goal*: to settle claims. Not being restricted to current business practices for accomplishing this goal, the team proposed new techniques that involve new strategic business relationships with insurance agents and body shops.

Figure 14.6 shows the Strategic Rationale Model for one new technique, handling small claims by the insurance agent.

A dependency relates claims handling and the insurance agent. Note their respective claims boundaries in the figure.

A task for the agent is then decomposed into several tasks. Note that task decompositions always occur *within* the boundary of an actor.

On the other hand, strategic dependency links are always *between* different actors, and cross their actor boundaries.

Figure 14.7 shows the Strategic Rationale Model for another new technique, handling small claims by the body shop.

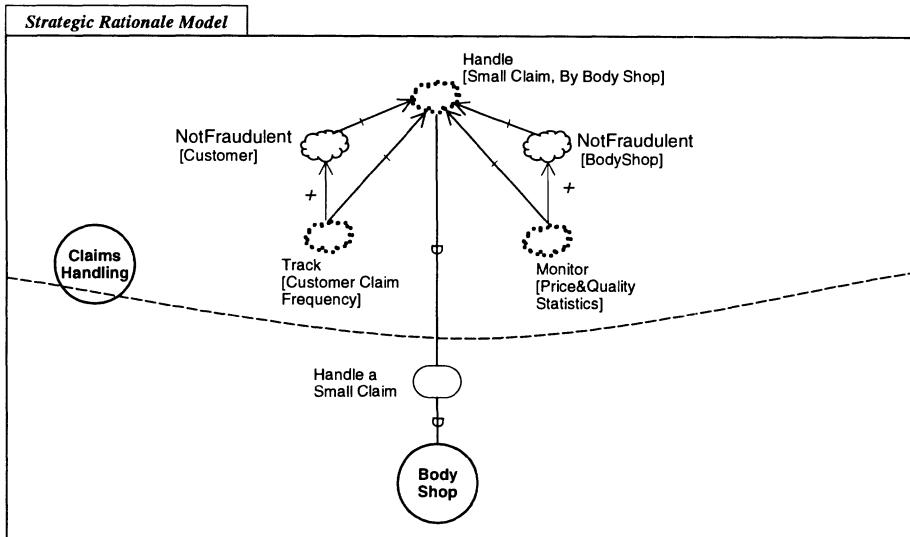


Figure 14.7. Strategic Rationale Model for handling small claims by the body shop.

Observe that tasks can be refined into softgoals within the i^* Framework. In the NFR Framework, however, operationalizing softgoals cannot be refined into NFR softgoals. It is interesting to observe that both the i^* Framework and the Performance Requirements Framework (Chapters 8 and 9) allow multiple layers of refining NFR softgoals into operationalizations, which can then in effect be refined into NFR softgoals at lower layers. This is done in different ways. The i^* Framework allows tasks (which are akin to operationalizing softgoals) to be refined directly into softgoals. The Performance Requirements Framework uses a layering mechanism: a particular NFR softgoal can be refined into operationalizations at the same (or lower) layers; at the same time, the particular NFR softgoal can also be refined into other NFR softgoals at lower layers. This keeps within the restriction of the NFR Framework which disallows operationalizing softgoals to be refined into NFR softgoals.

Figure 14.8 shows the overall Strategic Rationale Model. It displays the existing approach and proposed alternatives for handling claims. It combines earlier Figures 14.4 through 14.7. The top part of the model should be quite understandable to those familiar with the NFR Framework.

The bottom half of the figure analyzes the strategic rationales for different approaches to handling claims. From left to right, it shows the existing technique, then the proposed techniques for handling small claims by the insur-

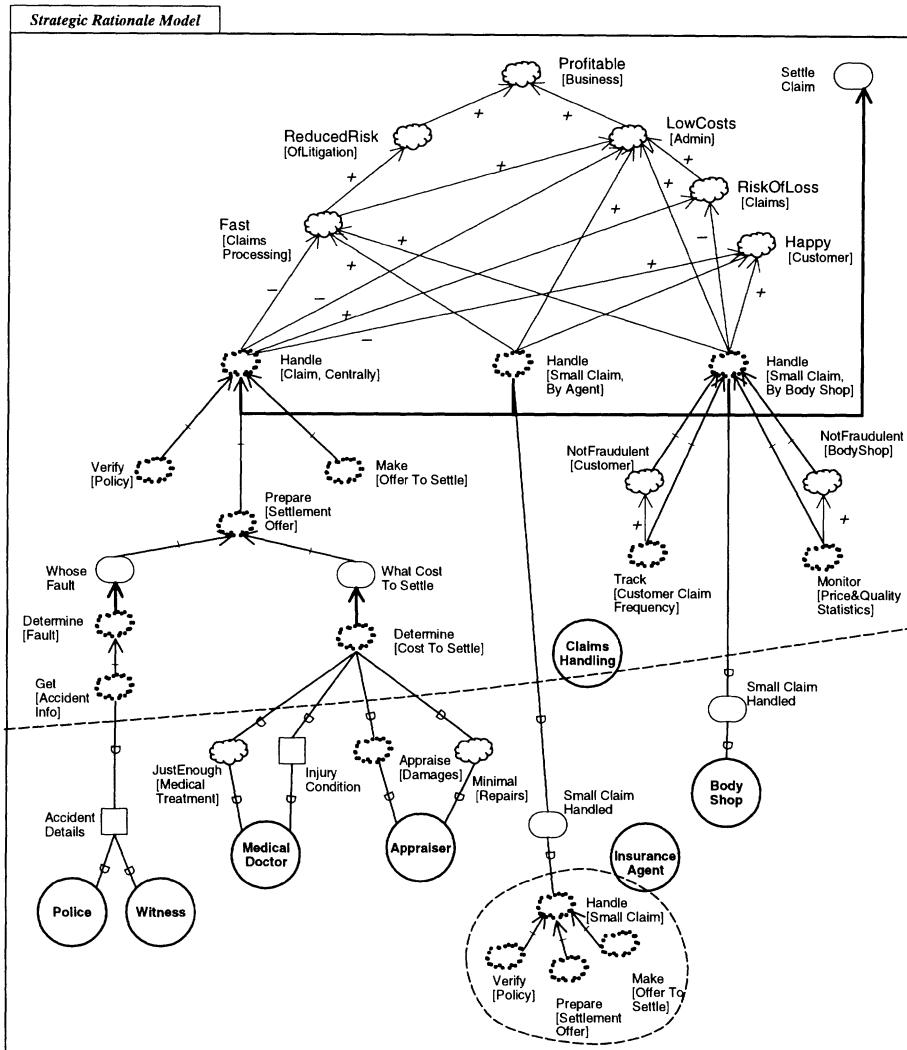


Figure 14.8. Overall Strategic Rationale Model for Claims Handling.

ance agent, and by the body shop. This analysis has an impact on the Softgoal Interdependency Graph at the top of the figure.

The boundaries of the different actors are shown in the figure. Interestingly, different techniques have analysis clustered on different sides of actor boundaries. For example, for claims handled by the body shop, the analysis

shows what the *insurer* will do to track possible fraud cases. On the other hand, for claims handled by the insurance agent, who is presumably trustworthy, the analysis shows what the *agent* do.

By explicitly representing means-ends relationships, the Strategic Rationale Model provides a systematic way for exploring the space of possible new process designs. Just as the NFR Framework offers catalogues to organize knowledge of methods and correlations, in the *i** Framework, generic knowledge in the form of methods and rules can be used to suggest new techniques and to identify related goals [Yu94d].

14.4 DISCUSSION

The concepts and techniques developed in the NFR Framework have applications not limited to the design of software systems. This chapter has illustrated an application of the NFR Framework to the area of enterprise modelling and business process redesign. This uses concepts of the NFR Framework, including softgoals, interdependencies, contributions and catalogues of knowledge.

As information systems are increasingly called upon to help alter the strategic relationships among business work units and external players (such as customers, suppliers, and business partners), models that are capable of describing strategic relationships and to help reason about them are needed. We have proposed one approach which emphasizes that organizations are made up of strategic, intentional actors. The Strategic Dependency Model allows the modelling of how strategic actors relate to each other intentionally. In comparison, the Strategic Rationale Model allows modelling of the means-ends reasoning done by actors concerning different ways of relating to each other for accomplishing work. Moreover, the adoption of a conceptual modelling approach facilitates the connection between organizational requirements and information systems development, using a consistent knowledge-based approach to software engineering (e.g. [Jarke92a, 93b]).

Another application of the *i** Framework is in the domain of software processes. Software process improvement is very important to ensure quality software. The *i** Framework is applicable to software processes as well as business processes [Yu94a].

A distinctive feature of the *i** Framework is that it incorporates concepts and techniques from the NFR Framework in several ways. These include *softgoals*, which are *refined*, and make *contributions* along *interdependency links*. Refinements are often drawn from *catalogues of knowledge*. In addition, the *source* and *target* system descriptions are interrelated.

The *i** Framework uses softgoals of the NFR Framework to complement the more conventional notion of goal, as in artificial intelligence, to enrich modelling and reasoning about relationships.

14.5 LITERATURE NOTES

This chapter is based on [Yu94c], and [Mylopoulos97] which was written by John Mylopoulos, Alex Borgida and Eric Yu.

The i^* Framework is presented in detail elsewhere [Yu93a,b] [Yu94a,b,c,d]. The models are formally represented in the conceptual modelling language Telos [Mylopoulos91] and their semantics are based on intentional concepts such as goal, belief, ability, and commitment (e.g., [Cohen90]).

Goal-oriented reasoning is in a number of requirements engineering frameworks [Feather93] [Fickas85, 91, 92] [Dubois94] [Dardenne91, 93].

Earlier presentations of the NFR Framework used the term “goal” for what is now termed “softgoal.” The i^* Framework used the term “softgoal” for that concept from the NFR Framework, and the name is now used in the NFR Framework.

15 ASSESSMENT OF STUDIES

The NFR Framework has been applied to a number of case studies. We now present some feedback on the NFR Framework and some studies.

Feedback is presented from the viewpoint of domain experts familiar with the kinds of systems and organizations studied. This evaluation was obtained from interviewees who were not involved in the development of the NFR Framework. They reviewed some of the studies of using the Framework to deal with accuracy, performance and security requirements. To conclude the chapter, we discuss methodological considerations for studies.

Through this feedback, we have gained further insights, which have helped to improve the Framework. For example, feedback indicated the need to improve notation and terminology, and such changes have been reflected in the preparation of this book.

15.1 FEEDBACK FROM DOMAIN EXPERTS

The feedback addresses a number of studies. These include those studied in this book, which were summarized in Figure 10.1: credit card systems (Chapter 11) [Nixon93, 97a] [Chung93a,b, 95b] and income tax appeals (Chapter 12) [Nixon94a, 97a] [Chung95b]. The feedback also addresses some systems studied elsewhere, summarized in Figure 10.2: Cabinet documents management [Chung93a, 95b] and health insurance systems [Chung93a, 95b].

How Interviews were Conducted

To obtain feedback, both positive and negative, interviews were conducted in person or by telephone. Before the interviews, some written materials were circulated to the interviewees. The materials included an earlier draft of a paper summarizing a number of studies of using the NFR Framework [Chung95b], along associated questionnaires on NFRs developed for [Chung95b]. More detailed versions of the studies (including [Nixon93, 94a] [Chung93a,b]) were also circulated to appropriate interviewees.¹

For the income tax appeals system we interviewed Mr. Bill Polimenakos, who is Project Leader of the InfoAccess Group in the Information Management Systems Branch of Revenue Canada. For the Cabinet documents system we interviewed Mr. Lou Melli, who is an independent systems consultant, not a part of the government organization studied, who takes a consultative and interactive approach to dealing with systems and their requirements. For the health insurance study we interviewed Mr. Dominic Covvey, who is a medical computing consultant and was a consultant to a commission of inquiry on the confidentiality of medical records [Ontario80]. In addition, for the Credit Card system studies (Chapter 11), which were based on a composite of information from different credit card associations, we had an initial discussion (without going through our questionnaire in detail) with Mr. Brian Brett, Manager of Card Services Systems, Retail Banking for the Royal Bank of Canada.

The interviews generally followed parts of the questionnaires. Due to limited interview time, however, not all questions were asked. Due to time constraints on our part, there was not always sufficient time for these interviewees to fully review our material in advance.

We present a composite of the responses. This is based on all interviews conducted.

We classify the issues that were raised. There were major comments, as well as responses on the NFR Framework, the application domains, and on methodology.

Major Comments

First, we present the major comments, both positive and negative.

The interviewees said that the *NFR Framework* and its components would be helpful for developers, and can be helpful in the broad domains studied. The cataloguing of development techniques and NFR-specific knowledge would be helpful, and in some cases is comparable to, or more advanced than current practice in the domains considered. For medical systems, such catalogues are not available to guide people. The softgoal interdependency graphs and their components were considered helpful. However, all but one of the

¹ Descriptions (including diagrams and notations) of the studies given to interviewees were subsequently revised when presented in this book. Two of the interviewees have been associated with the University of Toronto.

interviewees did not initially understand the details of the softgoal interdependency graphs. In all cases, this was remedied by an explanation of the NFR Framework during the interview. Once understood, softgoal interdependency graphs were appreciated by all. In preparing this book, we have endeavoured to make softgoal interdependency graphs more understandable, and easier to read, by improving naming and presentation.

In conducting the studies, we initially aimed to use the NFR Framework to deal with real quality-related problems of large real systems. Concerning *applicability of the studies to the specific domains studied*, the interviewees felt that our studies were generally based on their domains, and illustrated our approach. However, for both government administrative systems (tax appeals and Cabinet documents), our softgoal prioritization, tradeoffs and resulting SIGs were different than what they would have produced. Given our assumptions, however, our results were understandable to them. We thus realized that our lack of consultation with domain people during the study left gaps in our domain knowledge. The medical interviewee could not evaluate our coverage since its level of presentation was too shallow compared to the level of detail in the particular system studied (which was old, ill-structured and difficult even for experts to understand).

There was a warm appreciation of the potential of the NFR Framework's *applicability to the broader domains studied*. One interviewee felt that our techniques could be used in the medical field, such as in developing procurement criteria when enhancing products, both existing ones and those under development.

The issue of the *scalability* of the NFR Framework was raised. The softgoal interdependency graphs were considered nice, but would the approach work in a large system (e.g., credit cards)? This needs to be tested in practice. At some level, large softgoal interdependency graphs may become cumbersome.

An open question concerning *training costs and payoff* was raised by several interviewees. Will the time and costs needed for training staff in the NFR Framework and for developing softgoal interdependency graphs be surpassed by payoffs during development?

Framework Responses

A number of responses addressed the NFR Framework.

There was an appreciation of the *NFR Framework's emphases*. One interviewee considered its focus on evaluation important, while another felt the evaluation approach needed explanation. The NFR type catalogue captured the kinds of concepts needed, but more detail can be added to it. It was felt that there is a need to clarify and meet requirements, and the Framework's ability to refine and focus on priority softgoals was understood. In one study, design rationale was found a little hard to understand.

The ability to be more *formal* when using the NFR Framework was considered positive. However, the Framework would have to be usable to domain and systems staff who currently work much more informally.

One interviewee questioned the NFR Framework's premise that the quality of a product depends on the *quality of the process* that produces it. For example, there can be quality processes which do not lead to quality products.

It was felt that it is good to deal with *tradeoffs (correlations)* early. Currently, there is not a balanced treatment of tradeoffs for some medical systems. They are secure, but rigid; to increase flexibility, security features may be disabled. On the other hand, for some government systems, a consultant can record conflict and synergy by using interviews and developing prototypes, which are then discussed with the client.

The broad quality issues (performance, accuracy and security) we addressed are of interest to the organizations. One open question is how one *determines the specific main (top-level) requirements*. Furthermore, can softgoals be dealt with in small clusters, and the results then combined?

Application-Domain Responses

Some important comments assessed our treatment of knowledge of the particular domains, and the proportionality of our treatment.

For both government administrative systems, we learned in the interviews that we did not give enough proportion to some important NFRs or address the most important tradeoffs or workload components. We attribute this to our lack of contact with domain people during the studies, once we had obtained the documents. As a result, we lacked some knowledge about the domain, its priorities and terminology.

Notably, in both administrative studies, we understated the importance of accuracy, which must be met, even in the presence of a tradeoff.

In the Taxation appeals study, we treated time and space as important. In fact, response time was important, but space was not (due to the relatively small number of appeals). Also, our treatment did not decompose the workload into interactive individual requests and periodic batch reporting. As a result, we did not address the problem of maintaining both accuracy and speed when batch reports are produced. This in fact was a central concern in the actual system, and needed to be considered (along with some specific implementation techniques, e.g., by keeping the system "read-only" when producing periodic reports) to properly deal with important requirements and tradeoffs.

The Cabinet documents study did not always correspond to the domain because we lacked some knowledge about the domain, its priorities and terminology. We treated security as a main softgoal, but in the domain, accuracy actually predominated other NFRs. Another reason why security was less crucial than considered in the study, was that clients in the particular domain seemed primarily interested in bringing the system into existence. In addition, the system would not store the actual Cabinet documents, just "header information" (such as topic and date). In addition, we detected an interaction, but it was among requirements which were minor in the domain.

We also found that some of our arguments about the domains were not quite right, again due to our lack of knowledge about the domains and their terminology.

Methodology Responses

Finally, some comments addressed methodological issues.

Some responses addressed *lifecycle issues*. Some of the larger systems studied have quite structured methodologies, which deal with all phases of the lifecycle (including post-development). How could the NFR Framework (which to date has had some emphasis on the earlier phases) be suitably integrated with other methodologies (which include production, testing, complaint handling, and feedback)? And how can the Framework help large existing systems, not built from scratch?

Both government agencies use interview-based approaches for *requirements acquisition*, to ascertain the client's needs. One interviewee was able to relate the NFR Framework's decomposition approach to an adaptive interview-based approach used by consultants in determining requirements of a small government system.

One interviewee found it nice that an softgoal interdependency graph can have a lot of *documentation* concisely in one page. There is the initial cost of extra time to understand softgoal interdependency graphs the first time, but the advantage is the expected future time savings, due to savings in the amount of documentation. (Note however that the overall size of documentation is not a problem in that organization.)

Training and Payback. Some interviewees pointed out that processes and methodologies need *people* to be used in practice. A variety of people (developers, administrators, etc.) would have to be *trained* in the use the NFR Framework. The *payback* (both from improvement in software quality and in methodology) needs to be determined. When does the approach start and stop being cost-effective?

15.2 DISCUSSION: LESSONS LEARNED FOR CONDUCTING STUDIES

We feel that comments from domain experts are valuable "reality checks" on our work. In reviewing the interviews, we feel that future studies could benefit from a more interactive and consultative approach, which would involve repeated contact early in the study to ensure that we have appropriately captured the priorities for requirements. This might allow studies to be closer to actual problems in the organizations studied, and allow more specific evaluation of the results of studies.

Our informal interviewing provided some initial feedback for the NFR Framework and studies. However, they were only done for some studies, and varied in the degree of detail covered (See the bottoms of Figures 10.1 and 10.2).

In our approach to the studies, things developed a step at a time. After studying a generic example (the research expense administration example), we then tried studying a variety of real systems, and then obtained expert feedback. However, we did not select an experimental methodology in advance.

In the future, we feel we could benefit from more thorough approaches. To obtain results with experimental rigour, we should consider choosing an experimental methodology, preferably from the start of the experiment. For example, we could consider a number of design issues [Yin89] for case studies, such as defining the objectives of a study, evidence to be considered, techniques, etc. Another concern for rigour in the area of management information systems is that case studies can address phenomena not directly observable, such as human factors, organizational power and ergonomics. Hence, such studies seem unable to meet the classical requirements of the “scientific method” (in the natural sciences). Yet an approach has been given [A. Lee89] for them to meet the requirements of rigorous research. This rigour is achieved by testing, among other things, if a given theory is falsifiable and confirmable, and if it allows for replicability and generalizability. In addition, when questionnaires are used, a more thorough process for designing, implementing and analyzing them can be used [Glushkovsky95].

Our studies were based on the documents, which varied in their nature and granularity, making it harder to make comparisons. In addition, a great deal of an organization’s “culture” may not be explicitly stated in the documents, yet is vital to understand the context of the system being developed. We have not directly dealt with this issue, and using more of an interview-based approach may help.

As future work, we would like to conduct larger studies, covering a larger variety of systems. This could involve studies across a spectrum of developers and also addressing additional factors, such as interplay between the NFR Framework and its social setting, user acceptance or resistance, and seamless integration into existing development tools. Another avenue would be to use the Framework from the start of a system development project.

It would be interesting to observe two teams working on the same task, one using the NFR Framework, the other not, and to compare the results. In another study, one team could use the Framework to develop an initial system, followed by another team which would deal with change using the initial results. This would help determine how easy it is to transfer SIGs between teams.

We would like to obtain more feedback on the NFR Framework and studies from academic researchers and industrial practitioners. It would also be helpful to address questions already raised during interviews. One open issue is the scalability of the Framework with respect to large systems. There is also the need to develop and evaluate methods of training users of the Framework and tools.

15.3 LITERATURE NOTES

Feedback on the studies is presented in [Chung95b]. Questionnaires on NFRs and the studies were developed for [Chung95b], and were used to conduct interviews with domain experts.

Thanks to Hui Liu (University of Jyväskylä) for discussions, presented in [Nixon97a], about methodologies for conducting studies.

POSTSCRIPT

To conclude the book, benefits of the NFR Framework, its current status, directions for improvement, and future work are described.

SYNOPSIS

This book has presented the NFR Framework, a systematic framework for dealing with non-functional requirements for software systems. By their nature, NFRs are difficult to handle, often are stated briefly and ambiguously, interact with each other, and have a global impact on a system. The NFR Framework helps developers to systematically address NFRs.

In dealing with NFRs, we feel there is a need to offer a structured, systematic approach, to help deal with the large space of development alternatives. As the body of knowledge can grow, we see the need for an extensible approach. Furthermore, developers' expertise is crucial for addressing NFRs. Hence usage of the NFR Framework is interactive, giving control of the development process to the *developers*, who use their expertise to *build quality into systems*.

The NFR Framework takes as its focal point the notion of *softgoal*, which is more flexible than goals in logical formalisms. Rather than initially seeking exact solutions, the Framework uses the notion of *satisficing* to obtain reasonable solutions. The Framework is also *qualitative* and *process-oriented* in its representation and analysis of NFRs.

The NFR Framework offers facilities for stating softgoals and then *refining* them into more specific softgoals. Softgoal refinement is facilitated by using *catalogues of methods*.

Contributions are used to represent the varying degrees of impact that one softgoal has upon another. They offer a more flexible approach than logical frameworks such as AND/OR trees.

Interactions and *tradeoffs* among softgoals can be detected via *catalogues of correlations*. In moving towards target systems, softgoals are *operationalized*, drawing on techniques from industrial and academic experience. *Design rationale* is recorded, so that it is available for current and future use.

All this is recorded in a graphical record, the *softgoal interdependency graph*. The graph can be evaluated to determine whether the decisions made will satisfice (i.e., help to meet) the softgoals.

We feel that the softgoal interdependency graph helps to capture the complex and dynamic development process. The development process could be quite complex, as it could involve a large search space and occurrences of conflicts, and quite dynamic, as a successful method application can become unsatisfactory later when a conflict is encountered or a better solution is found. A softgoal interdependency graph acts not only as a means for dealing with NFRs, but also as a record of development history which can be used for later review, justification and change.

The NFR Framework has been specialized to deal in detail with particular NFRs, including *accuracy*, *security* and *performance*. These are some of the NFRs which are crucial to the successful development and use of software systems. Expertise and terminology for these NFRs have been captured and catalogued using the NFR Framework. This has involved, for example, the use of industrial and military concepts of security, and principles for building good performance into systems. The concept of *information flow* has been fruitfully applied to dealing with accuracy and security requirements. For performance requirements, the Framework has been further specialized to deal with information systems. The characteristics, methods and specification languages of information systems are addressed and data models features are used to further organize the consideration of issues. Quite interestingly, the Framework was able to be specialized to deal with different NFRs, even though the different fields have diverse terminologies and techniques.

We have conducted *empirical studies* to apply the NFR Framework to deal with a variety of NFRs (accuracy, performance and security) for a variety of information systems. The studies reported in this book address a variety of domains, by considering information systems for credit cards and government administration (tax appeals). Other studies (health insurance, Cabinet documents management, and bank loans) have been reported elsewhere [Chung93a, 95a,b, 96]. The systems vary in size, workload and the nature of their operations.

The studies have illustrated the different aspects of the NFR Framework, in showing how to build quality into a variety of systems. They show the use of a number of refinement methods and operationalizations. They also illustrate the capturing and organizing of a variety of knowledge — about NFRs, particular domains and systems, and of particular development approaches and languages.

The studies have shown how domain information, organizational priorities and system characteristics can be used to develop solutions which are *customized* for the particular domain and system. They have also illustrated dealing with interactions between different types of NFRs. They have shown how to resolve tradeoffs and conflicts, e.g., by using domain information. In the case of performance, tradeoff resolution was consistent with principles [C. Smith86, 90] for building performance into systems.

The NFR Framework also helps a developer deal with *defects* in requirements, including *ambiguities* in descriptions, *omissions* of concerns, and *conflicts*. This should help a developer attain the benefits of detecting defects as early as possible [Boehm87], while avoiding the low rate of detection for current techniques reported in [Schneider92].

To evaluate the NFR Framework and some of the studies, we obtained some feedback from people familiar with some of the domains studied. There was some initial support for the Framework. Facilities for cataloguing of knowledge and handling tradeoffs were appreciated. Our studies were based on their domains and illustrated our approach. However our detailed conclusions in the studies would not necessarily have been reached by people in the organizations studied. This seems mainly due to our use of source documents without ongoing contact with the organizations during the studies.

We have also presented *applications* to various areas, including *software architectural design*, and *business process redesign*. These applications use the core concepts of the NFR Framework, while drawing in concepts specific to these application areas.

In dealing with software architecture, one may wish to deal with various non-functional requirements (e.g., performance, modifiability, security and reusability) and their interactions, before making a commitment to a detailed design. An initial application of the NFR Framework to software architecture illustrated the use of different parts of the NFR Framework in addressing a standard example problem in software architecture. It also illustrated the use of the Framework outside the area of information system development.

The NFR Framework has also been applied to early phases of requirements analysis for modelling of organizations, and for dealing with business process redesign. Using concepts from the NFR Framework including softgoals and satisficing, organizational objectives and strategies are linked with the work processes and technologies that support them. The approach was illustrated with a standard example.

STATUS OF THE NFR FRAMEWORK

Additional work on the NFR Framework is reported in detail elsewhere. These include *tool support*, *dealing with change*, applying the NFR Framework to *decision support systems*, and a study of *software process improvement*.

Tools. Tool support for the NFR Framework is possible, as demonstrated by a family of NFR Tools [Chung93a, 94c, Nixon97a]. These prototypes help a developer use the NFR Framework by organizing and cataloguing a variety of kinds of knowledge. By using a knowledge base management language (Telos [Mylopoulos90]) and associated tools (e.g., the ConceptBase knowledge base management system [Jarke92b] as well as other Telos-based tools [Stanley95] [Kramer95]), graphical and textual definition browsers are made available. The Tool then helps the developer use the Framework to build softgoal interdepen-

dency graphs, select methods from catalogues, detect correlations from catalogues, and evaluate the satisficing of softgoals.

The tool supports the main components of the NFR Framework, for some of the NFRs considered in this book. A “core” tool which has the mechanisms of the NFR Framework (Part I of the book) can be extended to handle additional NFRs. This can be done by adding modules for each particular NFR, without changing the core mechanisms. Further work on tools is ongoing.

The NFR Tool grew out of the earlier Mapping Assistant [Chung91b] of the DAIDA environment [Jarke92a, 93b] which provides support for all phases of information system engineering. The Mapping Assistant aids the developer by relating source descriptions and requirements to target alternatives. These can be used to constrain the “mapping” of functional requirements objects into design objects, and the mapping of design objects into implementation objects.

Dealing with Change. A study of a bank loan system [Chung95a, 96], addressed a combination of several NFRs (accuracy, performance, informativeness, etc.). We showed how the NFR Framework can be adapted to deal with changes in requirements, workload, priorities, development techniques, and so on. By using existing softgoal interdependency graphs as a starting point, we were able to deal with changes quickly and effectively. The ability of the Framework to handle the aspect of change provides an additional payoff for the costs of using this knowledge-based approach.

This suggests some extensions to the NFR Framework, particularly in representing changes in strengths of softgoal achievement. The existing values (which currently include satisfied, denied, etc.) could be augmented with additional “shadings” such as several degrees of satisfied, and several degrees of denied. This would help us represent, for example, situations where a requirement is initially satisfied, and subsequently also satisfied, but to a greater degree. This could be viewed as adding some features of quantitative frameworks, while, at a high level of abstraction, retaining the features of a qualitative framework.

Decision Support Systems. An initial application of the NFR Framework has been made to decision support systems [Jurisica98], another class of systems with a variety of NFRs. Case-based reasoning systems are one such type of system which use past experiences stored as cases (problem description, solution and feedback information) in a case base, in order to solve a current problem by searching for similar cases. An initial application of the NFR Framework to case-based reasoning catalogued techniques for implementing a case-based reasoner, and also used existing NFR catalogues, particularly for performance. It then studied a case-based reasoner for a medical information system application, where performance, accuracy, and confidentiality were significant issues. It considered tradeoffs, e.g., between performance and the accuracy of reasoning, which selects a possible treatment and predicts the outcome. Interestingly, the study also illustrated dealing with various changes. It would be interesting

to try out the approach on a larger variety of case-based reasoning algorithms, e.g., as presented in [Jurisica97], for a variety of domains with varying requirements.

Similarly, selection among libraries of existing alternatives (e.g., software packages) may be of interest [Nixon94b]. One can imagine using the NFR Framework to consider selection among alternatives, which may be encapsulated, e.g., in a database programming language. Along these lines, it would be helpful to identify and define *templates* (portions of SIGs) for commonly occurring patterns of refinement. To select among templates, one approach would be to look for similarities, using a case-based reasoner.

Study of Software Process Improvement. Preliminary evidence that the NFR Framework can improve the software development process is presented in [Cysneiros99], which describes an adaptation of the NFR Framework, and an associated case study. The Entity-Relationship (E-R) model is adapted to include NFRs, and the NFR Framework's SIGs are adapted to distinguish attributes of data classes from other attributes. A number of conditions are presented to ensure that an extended E-R model and an extended SIG are maintained consistently. In a case study of a laboratory information system, three teams worked on different laboratory areas, working independently except for a joint validation at the end of the project. One team used the combined ER-NFR approach to analyse NFRs, updating the SIG and the E-R model when omissions (such as unrepresented entities and attributes) and conflicts were detected. The other two teams did not use the NFR Framework, and each of the three teams developed software for a different laboratory area. The combined ER-NFR approach helped find changes in the data models causes by changes in NFRs. The payoff of using the NFR Framework was fewer changes after software delivery, i.e., lower maintenance costs. The overhead of using the combined ER-NFR approach was estimated to be less than 10% of development time.

PROSPECTS FOR THE NFR FRAMEWORK

We also see a number of future directions for the NFR Framework.

An obvious next step would be to extend the NFR Framework to provide a large number of specialized frameworks, each one dealing in detail with a different NFR. This would in effect produce an “encyclopaedia of NFRs,” with detailed entries for many NFRs. Here we have presented accuracy, security and performance in detail. We would like to look at other NFRs, such as reliability and fault tolerance. We would continue using knowledge bases to collect generic NFR types, methods and correlations.

The NFR Framework could be extended to deal with the combination of *quantitative* and *qualitative* reasoning. This could draw on quantitative work on metrics. For performance requirements, for example, this would include the integration of quantitative performance prediction [Nixon91] with qualitative selection among target alternatives.

Further studies are ongoing. Case studies using the Framework are being conducted at a telecommunications company, where the Framework is being considered for assisting architectural design. Another recent case study of NFRs deals with a university's student records system, including requirements for performance and Year 2000 compliance.

We would also like to see the NFR Framework applied during a greater variety of phases of the software lifecycle, not just the phases illustrated in this book (primarily requirements, conceptual design, and organizational modelling). For example, this could include testing, inspection and maintenance phases. While the use of the Framework during these additional phases has not been demonstrated in this book, we do feel such extensions can be done in a similar manner, based on the work that has been shown in this book.

The NFR Framework has focussed on analysis of NFRs during software development, drawing on work on decision support and design rationale [J. Lee91] for a broad range of design activities, not limited to software development. The components and aspects of the NFR Framework (such as facilities for refining softgoals, dealing with tradeoffs and ambiguities, and cataloguing concepts and development techniques) may be able to be applied to more general design tasks, beyond software design. For example, the use of three different softgoals (representing NFRs, design rationale, and (target) operationalizations) may be helpful in a variety of design contexts.

Concerning education and technology transfer, the NFR Framework has been used in several graduate-level courses in software engineering, requirements engineering, software architecture, and information systems design. The Framework was able to be used to elucidate how NFRs can be used to directly and systematically address quality concerns. Students working in industry could readily see how the Framework can help deal with the real-life problems they face at work.

Finally, we see two basic theses for which the work reported in this book gives some evidence:

1. We see the NFR Framework as being independent of the particular process that one uses to do requirements or software development. It can be attached to waterfall, spiral or other processes. In other words, the NFR Framework is not autonomous. It could fit into a variety of frameworks for functional requirements, e.g., data-flow diagrams [DeMarco78] or KAOS [Dardenne93]. It is a complementary model, for capturing and analyzing NFRs.

2. We think that the NFR Framework is applicable not only to NFRs, but to all kinds of requirements, including functional requirements. This is because all requirements can be viewed as softgoals which compete with each other and may have to be relaxed in order to accommodate other requirements. Thus there can be a degree of hardness or softness associated with each softgoal. Softgoals may be soft, or hard, or have a varying degree of softness or hardness. This may even apply to hard constraints, e.g., performance of 500 transactions per second. The developer might respond by offering 350 transactions per second, plus good security, thus relaxing the stated requirements to handle

a collection of requirements. This allows a generalization of a view where functional requirements are exclusively hard, and NFRs are exclusively soft. This is a benefit of the NFR Framework, allowing reasoning about consistency in a way that is more flexible than logical frameworks (See, e.g., [Nuseibeh93]).

LITERATURE NOTES

Parts of this Postscript are taken from [Nixon97a]. Our own evaluation of the NFR Framework, and directions for the future, are presented in [Chung93a, 95b, Nixon97a].

Thanks to Michael Brodie and Florian Matthes for observations concerning selection among software libraries, Igor Jurisica for discussions on case-based reasoning, and Joachim Schmidt for suggestions on relating the NFR Framework to work on database programming languages.

BIBLIOGRAPHY

- [AI84] *Artificial Intelligence*, An International Journal, Special Volume on Qualitative Reasoning about Physical Systems, Vol. 24, No. 1–3, Dec., 1984.
- [ANSI] Copies of ANSI standards can be obtained from: American National Standards Institute: U. S. Standards on Systems Quality and Related Topics, New York, NY.
- [Abowd93] G. Abowd, R. Allen and G. Garlan, “Using Style to Understand Descriptions of Software Architectures,” *Software Engineering Notes*, 18(5): 9–20, 1993. *Proc. of SIGSOFT ‘93: Symposium on the Foundations of Software Engineering*.
- [Abrial88] J-R. Abrial, C. Morgan, M. Spivey, and T. Vickers, *The Logic of ‘B’*, 26 Rue des Plantes, Paris, 75014, Sept. 1988.
- [Adam92] John A. Adam, “Threats and Countermeasures,” *IEEE Spectrum*, vol. 29, no. 8, Aug. 1992, pp. 21–28.
- [Albano85] Antonio Albano, Luca Cardelli and Renzo Orsini, “Galileo: A Strongly Typed, Interactive Conceptual Language.” *ACM TODS*, Vol. 10, No. 2, Aug. 1985.
- [Albano89] Antonio Albano, Conceptual Languages: A Comparison of ADAPLEX, Galileo and Taxis. In Joachim W. Schmidt and Costantino Thanos (Editors), *Foundations of Knowledge Base Management*. Berlin: Springer-Verlag, 1989, pp. 395–408.
- [Amoroso91] E. Amoroso, T. Nguyen, J. Weiss, J. Watson, P. Lapiska, and T. Starr, “Towards an Approach to Measuring Software Trust,” *Proc. IEEE Symposium on Security and Privacy*, May 1991, pp. 198–218.
- [Anderson89] John S. Anderson and Stephen Fickas, “A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem.” In *Proceedings, Fifth International Workshop on Software Specification and Design*, Pittsburgh, PA, U.S.A., May 19–20, 1989. Washington, IEEE Computer Society Press, 1989, pp. 177–184.
- [Andrews87] Timothy Andrews and Craig Harris, “Combining Language and Database Advances in an Object-Oriented Development Environment.” In Norman Meyerowitz (Editor), *OOPSLA ’87 Conference Proceedings*, Orlando, FL, 4–8 October 1987, *SIGPLAN Notices*, Vol. 22, No. 12, December 1987, pp. 430–440

- [Anton96] A. I. Anton, "Goal-based Requirements Analysis." *Proc., 2nd IEEE International Conference on Requirements Engineering*, April 1996.
- [Aristotle, 350 B.C.] Aristotle, *Topics*. 350 B.C.
- [Atkinson87] Malcolm P. Atkinson and O. Peter Buneman, "Types and Persistence in Database Programming Languages." *Computing Surveys*, Vol. 19, No. 2, June 1987, pp. 105–190.
- [BIM89] *BIM Prolog 2.4 Manual*. BIM sa/nv, Belgium, 1989.
- [Balzer85] Robert Balzer, "A 15 Year Perspective on Automatic Programming." *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985, pp. 1257–1268.
- [Barclays92] Barclays Bank PLC, *The Barclays Code of Business Banking*. London, England, effective 31st Jan. 1992.
- [Barclays93a] Barclays Bank PLC, *The Barclays Code of Business Banking*. London, England, May 1993.
- [Barclays93b] Barclays Bank PLC, *Annual Review & Summary Financial Statement*, London, England, 1993.
- [Barstow87] David Barstow, Artificial Intelligence and Software Engineering, *Proceedings, Ninth International Conference on Software Engineering*, Monterey, CA, USA, March 30 – April 2, 1987, pp. 200–211.
- [Basili88] V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. on Software Eng.*, vol. 14, no. 6, June 1988, pp. 758–773.
- [Basili91] V. R. Basili and J. D. Musa, The Future Engineering of Software: A Management Perspective, *IEEE Computer*, Vol. 24, No. 9, Sept. 1991, pp. 90–96.
- [Batory88] D. S. Batory, *Concepts for a Database System Compiler*. Technical Report TR-88-01, Department of Computer Sciences, The University of Texas at Austin, January 1988.
- [Bauer76] F. L. Bauer, "Programming as an Evolutionary Process," *Proc. 2nd International Conf. on Software Engineering*, Oct. 1976, pp. 223–234.
- [Benzaken90] Véronique Benzaken, An Evaluation Model for Clustering Strategies in the O₂ Object-Oriented Database System. In S. Abiteboul and P. C. Kanellakis (Editors), *ICDT '90*, Proceedings, Third International Conference on Database Theory, Paris, 12–14 December 1990. Berlin: Springer-Verlag, 1990, pp. 126–140.
- [Benzel89] T. C. V. Benzel, "Developing Trusted Systems Using DOD-STD-2167A," *5th Annual Computer Security Applications Conf.*, Tucson, Arizona, Dec. 4–8, 1989, pp. 166–176.
- [Bishop97] Matt Bishop, Steven Cheung and Christopher Wee, "The Threat from the Net." *IEEE Spectrum*, August 1997, pp. 56–63.
- [Boehm76] B. W. Boehm, J. R. Brown and M. Lipow, "Quantitative Evaluation of Software Quality. In *Proceedings, 2nd International Conference on Software Engineering*, San Francisco, CA, Oct. 1976. Long Branch, CA: IEEE Computer Society, 1976, pp. 592–605.
- [Boehm78] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod and M. J. Merritt, *Characteristics of Software Quality*. Amsterdam: North-Holland, 1978.

- [Boehm86] B. W. Boehm, "A Spiral Model of Software Development and Enhancement", Vol. 11, No. 4, *ACM Software Eng. Notes*, Aug. 1986.
- [Boehm87] B. Boehm, "Industrial Software Metrics Top Ten List", *IEEE Software*, Sept. 1987.
- [Boehm94] B. Boehm, "Software Architectures: Critical Success Factors and Cost Drivers," Proc., *16th Int. Conf. on Software Engineering*, May 1994, p. 365.
- [Boehm96] Barry Boehm and Hoh In, Identifying Quality-Requirement Conflicts. *IEEE Software*, March 1996.
- [Booch94] G. Booch, *Object-Oriented Analysis and Design, with Applications*. Benjamin-Cummings, 1994.
- [Booch97] G. Booch, I. Jacobson and J. Rumbaugh *Unified Modeling Language User Guide, Unified Modeling Language Reference Manual*, Addison-Wesley, 1997.
- [Borgida85a] Alexander Borgida, "Features of Languages for the Development of Information Systems at the Conceptual Level." *IEEE Software*, Vol. 2, No. 1, January 1985, pp. 63–73.
- [Borgida85b] Alexander Borgida, "Language Features for Flexible Handling of Exceptions in Information Systems." *ACM TODS*, Vol. 10, No. 4, Dec. 1985, pp. 565–603.
- [Borgida85c] Alexander Borgida, Sol Greenspan and John Mylopoulos, "Knowledge Representation as the Basis for Requirements Specifications." *IEEE Computer*, Vol. 18, No. 4, April 1985, pp. 82–91.
- [Borgida89] Alex Borgida, Matthias Jarke, John Mylopoulos, Joachim W. Schmidt and Yannis Vassiliou, "The Software Development Environment as a Knowledge Base Management System." In Joachim W. Schmidt and Costantino Thanos (Editors), *Foundations of Knowledge Base Management*. Berlin: Springer-Verlag, 1989, pp. 411–442.
- [Borgida90a] Alexander Borgida, John Mylopoulos, Joachim W. Schmidt, Ingrid Wetzel, "Support for Data-Intensive Applications: Conceptual Design and Software Development." In Richard Hull, Ron Morrison and David Stemple (Editors), *Proceedings of the Second International Workshop on Database Programming Languages*, 4–8 June 1989, Salishan Lodge, Gleneden Beach, Oregon. San Mateo, CA: Morgan Kaufmann, 1990, pp. 258–280.
- [Borgida90b] Alexander Borgida, "Knowledge Representation, Semantic Modeling: Similarities and Differences." In *Proceedings, International Conference on Entity-Relationship Approach*, Lausanne, Switzerland, October 1990.
- [Borgida93] Alexander Borgida, John Mylopoulos and Joachim W. Schmidt, "The TaxisDL Software Description Language." In M. Jarke (Ed.), *Database Application Engineering with DAIDA*. Berlin: Springer-Verlag, 1993, pp. 65–84.
- [Bowen85] T. P. Bowen, G. B. Wigle and J. T. Tsai, Specification of Software Quality Attributes,
- [Brataas92] G. Brataas, A. L. Opdahl, V. Vetland and A. Sølvberg, *Information Systems: Final Evaluation of the IMSE*. Tech. Rep., IMSE Project Deliverable D6.6-2, SINTEF (Univ. of Trondheim), Norway, Feb. 27, 1992.

- [**Brataas96**] Gunnar Brataas, *Performance Engineering Method for Workflow Systems: An Integrated View of Human and Computerised Work Processes*. Dr.ing. thesis, Information Systems Group, Faculty of Physics, Informatics and Mathematics, Norwegian University of Science and Technology, 1996.
- [**Briand95**] L. Briand, W. L. Melo, C. Seaman and V. Basili, "Characterizing and Assessing a Large-Scale Software Maintenance Organization," *Proc., 17th Int. Conf. on Software Eng.*, Seattle, Washington, April 24–28, 1995.
- [**Brodie86**] Michael Brodie and John Mylopoulos (editors), *On Knowledge Base Management Systems*. New York: Springer-Verlag, 1986.
- [**Brooks87**] F. P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, Apr. 1987, Vol. 20, No. 4, pp. 10–19.
- [**Brooks95**] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley Longman, 1995.
- [**Bubenko81**] Janis A. Bubenko Jr. *On Concepts and Strategies for Requirements and Information Analysis*, SYSLAB Report no. 4, Dept. of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden, 1981.
- [**Bubenko91**] Janis A. Bubenko jr., *Towards a Corporate Knowledge Repository*. Report No. 91-023-DSV, SYSLAB, Dept. of Computer & Systems Sciences, Royal Institute of Technology and Stockholm University, Kista, Sweden, 1991.
- [**Bui87**] T. Bui and T. R. Sivasankaran, "Cost-Effective Modeling for a Decision Support System in Computer Security," *Computers & Security*, vol. 6, no. 2, Apr. 1987, pp. 139–151.
- [**CTCPEC92**] Canadian System Security Centre, *The Canadian Trusted Computer Product Evaluation Criteria*, Version 2.0. Ottawa, Apr. 1992.
- [**Callahan93**] J. R. Callahan, *Software Packaging*, Technical Report CS-TR-3093, University of Maryland, 1993.
- [**Canadian Bankers91**] Canadian Bankers' Association, *MasterCard and Visa Statistics*, Toronto, Ont., Dec. 1991.
- [**Cao94**] Cao, Jianqing; Qin, Mingwan; and He, Ming: A Development Process for Engineering Project Management Information Systems Based on Semantic Data Models. *Proceedings, CAIA-94, The Tenth IEEE Conference on Artificial Intelligence for Applications*, San Antonio, Texas, March 1–4, 1994.
- [**Casas86**] Ignacio R. Casas, *PROPHET: A Layered Analytical Model for Performance Prediction of Database Systems*. Ph.D. Thesis, Dept. of Computer Science, University of Toronto, March 1986. Technical Report CSRI-180, Computer Systems Research Institute, University of Toronto, April 1986.
- [**Ceri90**] Stefano Ceri and Jennifer Widom, Deriving Production Rules for Constraint Management. In Dennis McLeod, Ron Sacks-Davis and Hans Schek (Editors), *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 13–16 August 1990, pp. 566–577.
- [**Chan82**] Arvola Chan, Sy Danberg, Stephen Fox, Wen-Te K. Lin, Anil Nori and Daniel Ries, Storage and Access Structures to Support a Semantic Data Model. *Proceedings, Eighth International Conference on Very Large Data Bases*, Mexico City, Sept. 8–10, 1982, pp. 122–130.

- [Chaudhri92] Vinay K. Chaudhri, John Mylopoulos and Vassos Hadzilacos, Concurrency Control for Knowledge Bases. *Proceedings, Third International Conference on Knowledge Representation and Reasoning*, Boston, Oct. 1992.
- [Chen76] Peter Pin-Shan Chen, The Entity-Relationship Model — Toward a Unified View of Data. *ACM TODS*, Vol. 1, No. 1, March 1976, pp. 9–36.
- [Chung84] Kyungwha Lawrence Chung, *An Extended Taxis Compiler*. M.Sc. Thesis, Dept. of Computer Science, University of Toronto, January, 1984. Also CSRG Technical Note 37, 1984.
- [Chung88] K. Lawrence Chung, Daniel Rios-Zertuche, Brian A. Nixon and John Mylopoulos, “Process Management and Assertion Enforcement for a Semantic Data Model.” In J. W. Schmidt, S. Ceri and M. Missikof (Editors), *Advances in Database Technology — EDBT '88*, International Conference on Extending Database Technology, Venice, Italy, March 1988, Proceedings. Lecture Notes in Computer Science, No. 303. Berlin: Springer-Verlag, 1988, pp. 469–487.
- [Chung91a] Lawrence Chung, “Representation and Utilization of Non-Functional Requirements for Information System Design.” In R. Anderson, J. A. Bubenko, Jr., A. Sølvberg (Editors), *Advanced Information Systems Engineering*, Proceedings, Third International Conference CAiSE '91, Trondheim, Norway, May 13–15, 1991. Berlin: Springer-Verlag, 1991, pp. 5–30.
- [Chung91b] K. Lawrence Chung, Panagiotis Katalagarianos, Manolis Marakakis, Michalis Mertikas, John Mylopoulos and Yannis Vassiliou, “From Information System Requirements to Designs: A Mapping Framework.” *Information Systems*, Vol. 16, No. 4, 1991, pp. 429–461.
- [Chung93a] Kyungwha Lawrence Chung, *Representing and Using Non-Functional Requirements: A Process-Oriented Approach*. Ph.D. Thesis, Dept. of Computer Science, University of Toronto, June 1993. Also Technical Report DKBS-TR-93-1.
- [Chung93b] Lawrence Chung, “Dealing With Security Requirements During the Development of Information Systems.” In Colette Rolland, François Bodart and Corine Cauvet (Editors), *Advanced Information Systems Engineering*, Proceedings, Fifth International Conference CAiSE '93, Paris, France, June 8–11, 1993. Berlin: Springer-Verlag, 1993, pp. 234–251.
- [Chung94a] Lawrence Chung, Brian A. Nixon and Eric Yu, “Using Quality Requirements to Drive Software Development.” *Workshop on Research Issues in the Intersection Between Software Engineering and Artificial Intelligence*, Sorrento, Italy, May 16–17, 1994.
- [Chung94b] Lawrence Chung, Brian A. Nixon and Eric Yu, “Using Quality Requirements to Systematically Develop Quality Software.” *Proceedings, Fourth International Conference on Software Quality*, McLean, Virginia, October 3–5, 1994. (Based on [Chung94a].)
- [Chung94c] Lawrence Chung and Brian A. Nixon, Tool Support for Systematic Treatment of Non-Functional Requirements. Manuscript, December 1994.
- [Chung95a] Lawrence Chung, Brian A. Nixon and Eric Yu, “Using Non-Functional Requirements to Systematically Support Change.” *RE '95, The Second IEEE International Symposium on Requirements Engineering*, York, England, 27–29 March 1995, Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 132–139.

- [Chung95b] Lawrence Chung and Brian A. Nixon, "Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach." *Proceedings, 17th International Conference on Software Engineering*, Seattle, WA, U.S.A., April 24–28, 1995. pp. 25–37.
- [Chung95c] Lawrence Chung, Brian A. Nixon and Eric Yu, "Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design." In D. Garlan (ed.), *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, Washington, April 1995.
- [Chung95d] Lawrence Chung, Brian A. Nixon and Eric Yu, "An Approach to Building Quality into Software Architecture." *CD-ROM Supplement to the Proceedings of CASCON'95*, Toronto, Nov. 7–9, 1995 (A revision of [Chung95c]).
- [Chung96] Lawrence Chung, Brian A. Nixon and Eric Yu, "Dealing with Change: An Approach Using Non-Functional Requirements." *Requirements Engineering*, Vol. 1, No. 4, 1996, pp. 238–260. (Printed 1997. A revision and extension of [Chung95a].)
- [Chung98] Lawrence Chung and Eric Yu, "Achieving System-Wide Architectural Qualities," *OMG-DARPA-MCC Workshop on Compositional Software Architectures*. Monterey, CA, January 6–8, 1998.
- [Chung99] L. Chung, D. Gross and E. Yu, "Architectural Design to Meet Stakeholder Requirements," *Proceedings, First Working IFIP Conference on Software Architecture (WICSA1)*. San Antonio, Texas, Feb. 22–24, 1999. (Based on [Chung95c,d] [Chung93a] [Yu94b]).
- [Clark87] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," *Proc. IEEE Symp. on Security and Privacy*, 1987, pp. 184–194.
- [Coad90] P. Coad and E. Yourdon, *Object-Oriented Analysis*. Yourdon Press/Prentice-Hall, 1990.
- [Cohen90] P. R. Cohen and H. J. Levesque, "Intention is Choice with Commitment," *Artif. Intell.*, 42 (3), 1990.
- [Coleman91] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes & P. Jeremaes, *Object-Oriented Development: The FUSION Method*, Prentice-Hall, 1991.
- [Coleman94] D. Coleman, D. Ash, B. Lowther and P. Oman, "Using Metrics to Evaluate Software System Maintainability." *IEEE Computer*, vol. 27, no. 8, Aug. 1994, pp. 44–49.
- [Conklin88a] J. Conklin and M. L. Begeman, "gIBIS: A Hypertext Tool for Explanatory Policy Discussions," *Proc. Computer Supported Cooperative Work*, 1988.
- [Conklin88b] J. Conklin and M. L. Begeman, "gIBIS: A Hypertext Tool for Explanatory Policy Discussions," *ACM Transactions on Office Information Systems*, Vol. 6, No. 4, 1988, pp. 303–331.
- [Constable86] R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, I. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and J. Smith, *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

- [Constantopoulos95] Panos Constantopoulos, Matthias Jarke, John Mylopoulos, Yannis Vaassiliou, "The Software Information Base: A Server for Reuse." *VLDB Journal*, January 1995.
- [Crocker84] Olga Crocker, Cyril Charney and Johnny Sik Leung Chiu, *Quality Circles: A Guide to Participation and Productivity*, Methuen Publications, Ont., 1984.
- [Crocker89] S. D. Crocker, "Software Methodology for Development of a Trusted BMS: Identification of Critical Problems." *5th Annual Computer Security Applications Conf.*, Tucson, Arizona, Dec. 4-8, 1989, pp. 148-165.
- [Crosby79] Philip B. Crosby, *Quality is Free*. New York: McGraw-Hill, 1979.
- [Cruz90] Isabel Cruz, Declarative Query Languages for Object-Oriented Databases. Manuscript, Dept. of Computer Science, University of Toronto, March 1990.
- [Curtis92] W. Curtis, M. I. Kellner and J. Over, "Process Modelling," *Communications of the ACM*, Vol. 35, No. 9, 1992, pp. 75-90.
- [Cysneiros99] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite, "Integrating Non-Functional Requirements into Data Modeling." To appear in *Proceedings, International Symposium on Requirements Engineering*, 1999.
- [DOD-STD-2167A] Department of Defense, Military Standard: Defense System Software Development, Feb. 29, 1988.
- [Dardenne91] A. Dardenne, A. van Lamsweerde and S. Fickas, "Goal-directed Requirements Acquisition," *Science of Computer Programming*, Special issue on *6th Int. Workshop on Software Specification and Design*, Como, Italy, 1991.
- [Dardenne93] Anne Dardenne, Axel van Lamsweerde and Stephen Fickas, Goal-directed Requirements Acquisition. *Science of Computer Programming*, Vol. 20, 1993, pp. 3-50.
- [DagstuhlWorkshop95] Dagstuhl Seminar on Software Architectures. Schloss Dagstuhl, Saarland, Germany, 20-24 February 1995.
- [Davenport93] T. H. Davenport, *Process Innovation: Reengineering Work Through Information Technology*, Boston: Harvard Business School Press, 1993.
- [Davenport94] Thomas H. Davenport, "Saving IT's Soul: Human-Centered Information Management." *Harvard Business Review*, March-April, 1994, pp. 119-131.
- [Davis93] A. Davis, *Software Requirements: Objects, Functions and States*. Prentice Hall, 1993.
- [Dechter90] R. Dechter, From Local to Global Consistency. *Proceedings, 8th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, Ottawa, Ontario, May 1990, pp. 231-237.
- [de Kleer86] J. de Kleer, "Problem Solving with the ATMS," *Artificial Intelligence* Vol. 28, pp. 127-162, 1986.
- [DeMarco78] T. DeMarco, *Structured Analysis and System Specification*. New York: Yourdon Press, 1978.
- [De Michelis98] G. De Michelis, E. Dubois, M. Jarke, F. Matthes, J. Mylopoulos, M. Papozoglou, J. W. Schmidt, C. Woo, E. Yu, "A Three-Faceted View of Information Systems: The Challenge of Change." *Communications of the ACM*, Vol. 41, No. 12, December 1998, pp. 64-70.

- [Deming86] W. E. Deming, *Out of the Crisis*, Center for Advanced Engineering Study, MIT, 1986.
- [Denning79] D. E. Denning and P. J. Denning, "Data Security," *ACM Computing Surveys*, vol. 11, no. 3, Sept. 1979, pp. 227–249.
- [Denning84] D. E. Denning, "Cryptographic Checksums for Multilevel Database Security," *Proc. IEEE Symposium on Security and Privacy*, 1984, pp. 52–61.
- [DiMarco90] Chrysanne DiMarco, *Computational Stylistics for Natural Language Translation*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, 1990.
- [DiMarco93] Chrysanne DiMarco and Graeme Hirst, "A Computational Theory of Goal-Directed Style in Syntax", *Computational Linguistics*, 1993.
- [Di Vito90] B. Di Vito, C. Garvey, D. Kwong, A. Murray, J. Solomon and A. Wu, "The Deductive Theory Manager: A Knowledge Based System for Formal Verification," *Proc. IEEE Symposium on Security and Privacy*, 1990, pp. 306–318.
- [Dorfman90] M. Dorfman and R. Thayer (eds.) *Standards, Guidelines and Examples on System and Software Requirements Engineering*. IEEE Computer Society Press, 1990.
- [Downing92] K. Downing and S. Fickas, "A Qualitative Modeling Tool for Specification Criticism." In Pericles Loucopoulos and Roberto Zicari (editors), *Conceptual Modeling, Databases and CASE: An Integrated View of Information Systems Development*. New York: John Wiley, 1992, pp. 507–517.
- [Doyle79] J. Doyle, "A Truth Maintenance System." *Artificial Intelligence*, Vol. 12, pp. 231–272, 1979.
- [Dubois86] E. Dubois, J. Hagelstein, E. Lahou, F. Ponsaert and A. Rifaut, "A Knowledge Representation Language for Requirements Engineering," *Proc. IEEE*, Vol. 74, No. 10, Oct. 1986, pp. 1431–1444.
- [Dubois89] E. Dubois, "A Logic of Action for Supporting Goal-Oriented Elaborations of Requirements." *Proc., 5th International Workshop on Software Specification and Design*, Pittsburgh, PA, 1989, pp. 160–168.
- [Dubois94] Eric Dubois, Phillippe Du Bois and Frédérick Dubru, "Animating Formal Requirements Specifications of Cooperative Information Systems." In Michael Brodie, Matthias Jarke, Michael Papazoglou, *Proceedings of the Second International Conference on Cooperative Information Systems*, Toronto, May 17–20, 1994, pp. 101–112.
- [Dubois96] Eric Dubois and Suchan Wu, "A Framework for Dealing with and Specifying Security Requirements in Information Systems." In *Proceedings, IFIP SEC'96*, 1996.
- [Feather93] M. S. Feather, "Requirements Reconnoitering at the Juncture of Domain and Instance." *Proc., IEEE Int. Symp. on Requirements Eng.*, San Diego, CA, January 4–6, 1993, pp. 73–76.
- [Fenton97] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd ed., International Thomson Computer Press, 1997.
- [Fernandez89] E. B. Fernandez, E. Gudes and H. Song, "A Security Model for Object-Oriented Databases," *Proc. IEEE Symposium on Security and Privacy*, 1989, pp. 110–115.

- [Fickas85] Stephen F. Fickas, Automating the Transformational Development of Software. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985, pp. 1268–1277.
- [Fickas87] S. F. Fickas, *Automating the Software Specification Process*, Tech. Rep. 87-05, Comp. Sci. Dept. Univ. of Oregon, Dec. 1987.
- [Fickas88] Stephen Fickas and P. Nagarajan, “Being Suspicious: Critiquing Problem Specifications,” *Proc. AAAI-88*, Saint Paul, Minnesota, Aug. 21–26, 1988. pp. 19–24.
- [Fickas91] Stephen Fickas, Rob Helm and Martin Feather, When Things Go Wrong: Predicting Failure in Multi-Agent Systems. In Robert Balzer and John Mylopoulos (Workshop Co-Chairs), International Workshop on the Development of Intelligent Information Systems, Niagara-on-the-Lake, Ontario, April 21–23, 1991, pp. 47–53.
- [Fickas92] Stephen Fickas and B. Robert Helm, Knowledge Representation and Reasoning in the Design of Composite Systems. *IEEE Transactions on Software Engineering*, Vol. 18, No. 6, June 1992, pp. 470–482.
- [Fishman87] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan, “Iris: An Object-Oriented Database Management System.” *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, 1987.
- [Flynn92] D. Flynn, *Information System Requirements: Determination and Analysis*. McGraw-Hill, 1992.
- [Garlan92] D. Garlan, G. E. Kaiser, and D. Notkin, “Using Tool Abstraction to Compose Systems,” *IEEE Computer*, Vo. 25, June 1992. pp. 30–38.
- [Garlan93] D. Garlan and M. Shaw, “An Introduction to Software Architecture,” *Advances in Software Engineering and Knowledge Engineering: Vol. I*, World Scientific Publishing Co., 1993.
- [Garlan94] D. Garlan and D. Perry, “Software Architecture: Practice, Potential, and Pitfalls,” *Proc. 16th Int. Conf. on Software Engineering*, 1994, pp. 363–364.
- [Garvin87] David. A. Garvin, “Competing on the Eight Dimensions of Quality,” *Harvard Business Review*, Nov. – Dec. 1987, pp. 101–109.
- [Gause89] D. Gause and G. Weinberg, *Exploring Requirements*. Dorset House, 1989
- [Glushkovsky95] Eli A. Glushkovsky, Radu A. Florescu, Anat Hershkovits and Daniel Sipper, “Avoid a Flop: Use QFD With Questionnaires.” *Quality Progress*, June 1995, pp. 57–62.
- [Goldman83] Neil M. Goldman, “Three Dimensions of Design Development.” *Proceedings of the National Conference on Artificial Intelligence*, Washington, D.C., August 22–26, 1983, pp. 130–133.
- [Gotel94] O. C. Z. Gotel and A. C. W. Finkelstein, “An Analysis of the Requirements Traceability Problem.” *Proc., Int. Conf. on Requirements Eng.*, Colorado Springs, 1994.
- [Green76] C. Green, “The Design of the PSI Program Synthesis System,” *Proc. 2nd International Conf. on Software Engineering*, San Francisco, California, Oct. 1976, pp. 4–18.

- [Green86] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a Knowledge-Based Software Assistant," In C. Rich and R. C. Waters (eds.): *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann Publishers, Inc., 1986, pp. 377–428.
- [Green94] Stewart Green, *Goal-Driven Approaches to Requirements Engineering*. Technical Report No. 94/9, Department of Computing, University of the West of England, Bristol, England, 1994.
- [Greenspan82] Sol J. Greenspan, John Mylopoulos and Alex Borgida, Capturing more world knowledge in the requirements specification. *Proceedings, Sixth International Conference on Software Engineering*, Tokyo, Japan, September 1982, pp. 225–234.
- [Greenspan84] S. Greenspan, *Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition*. Ph.D. Thesis, Dept. of Computer Science, Univ. of Toronto, 1984.
- [Greenspan94] Sol Greenspan, John Mylopoulos, Alex Borgida, On Formal Requirements Modeling Languages: RML Revisited. *Proceedings, International Conference on Software Engineering*, Sorrento, Italy, May 16–21, 1994, pp. 135–147.
- [Haberman86] N. Haberman, and D. Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering* 12 (12), Dec. 1986.
- [Hahn91] U. Hahn, M. Jarke and T. Rose, "Teamwork Support in a Knowledge-Based Information Systems Environment," *IEEE Trans. Software Eng.*, 17(5), May 1991, pp. 467–482,
- [Hailstone91] "Quality Management and Software Engineering," In Darrel Ince (Ed.) *Software Quality and Reliability: Tools and Methods*, Chapman & Hall, T. J. Press (Padstow) Ltd., Padstow, Cornwall, UK, pp. 24–32.
- [Hammer90] M. Hammer, Reengineering Work: Don't Automate, Obliterate, *Harvard Business Review*, July-August 1990, pp. 104–112.
- [Hammer93] M. Hammer and J. Champy, *Reengineering the Corporation: A Manifesto for Business Revolution*, HarperBusiness, 1993.
- [Hammer95] Michael Hammer and Steven A. Stanton, *The Reengineering Revolution: A Handbook*. HarperBusiness, 1995.
- [Hartson76] H. Rex Hartson and David K. Hsiao, "Full Protection Specifications in the Semantic Model for Database Protection Languages." *Proceedings, ACM Annual Conference*, Houston, TX, Oct. 1976, pp. 90–95.
- [Hauser88] J. R. Hauser and D. Clausing, "The House of Quality," *Harvard Business Review*, May–June 1988, pp. 63–73.
- [Hayes79] Patrick H. Hayes, "The naive physics manifesto." In Donald Michie (Editor), *Expert Systems in the Microelectronic Age*. Edinburgh University Press, 1979, pp. 242–270.
- [Hayes87] I. Hayes (ed.), *Specification Case Studies*, Prentice Hall Int'l., Englewood Cliffs NJ, 1987.
- [Heninger80] K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. on Software Eng.*, vol. SE-6, no. 1, Jan. 1980, pp. 2–13, Reprinted in R. H. Thayer and M. Dorfman

- (eds.), *Tutorial: System and Software Requirements Engineering*, IEEE Computer Society Press, 1990.
- [Hoare78] C. A. R. Hoare, Communicating Sequential Processes. *CACM*, Vol. 21, No. 8, Aug. 1978, pp. 666–677.
- [Hooton88] Alan R. Hooton, Ulises Agüero and Subrata Dasgupta, An Exercise in Plausibility-Driven Design, *IEEE Computer*, pp. 21–31, July 1988.
- [Hull87] R. Hull and R. King, “Semantic Database Modeling: Survey, Applications, and Research Issues.” *Computing Surveys*, 19(3), Sept. 1987, pp. 201–260.
- [Humphrey89] Watts Humphrey, *Managing the Software Process*. Addison Wesley, 1989.
- [Hyslop91] W. F. Hyslop, *Performance Prediction of Relational Database Management Systems*. Ph.D. Thesis, Dept. of Comp. Sci., Univ. of Toronto, 1991.
- [IEEE] IEEE standards can be obtained from: IEEE Computer Society: Publications, software engineering standards, conferences, Washington, DC.
- [IEEESoftware87] *IEEE Software*, Sept. 1987.
- [ISO] Copies of ISO standards can be obtained from: American National Standards Institute: U. S. Standards on Systems Quality and Related Topics, New York, NY.
- [ITSEC89] German Information Security Agency, *Criteria for the Evaluation of Trustworthiness of Information Technology (IT) Systems*, 1st Version, Bundesanzeiger, Köln, Germany, 1989.
- [ITSEC91] Office for Official Publications of the European Communities, *Information Technology Security Evaluation Criteria, Provisional Harmonised Criteria*, Version 1.2, June 1991, Luxembourg.
- [IWASSWorkshop95] *Proceedings, ICSE-17 Workshop on Architectures for Software Systems*, Seattle, WA, USA, 24–25 April 1995.
- [Jackson83] Michael Jackson, *System Development*, Prentice-Hall, 1983.
- [Jacobson92] I. Jacobson, M. Christerson, P. Johnsson & G. Övergaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Prentice-Hall, 1992.
- [Jagadish92] H. V. Jagadish and Xiaolei Qian, Integrity Maintenance in an Object-Oriented Database. In Li-Yan Yuan (Editor), *Proceedings of the 18th Conference on Very Large Data Bases*, Vancouver, B.C., August 23–27, 1992, pp. 469–480.
- [Jain91] Raj Jain, *The Art of Computer System Performance Analysis*. New York: Wiley, 1991.
- [Jarke89] Matthias Jarke, Manfred Jeusfeld, and Thomas Rose, A Software Process Data Model for Knowledge Engineering in Information Systems. *Information Systems*, Vol. 14, No. 3, Fall 1989.
- [Jarke92a] Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou, “DAIDA: An Environment for Evolving Information Systems,” *ACM Transactions on Information Systems*, Vol. 10, No. 1, Jan. 1992, pp. 1–50.
- [Jarke92b] Matthias Jarke (Editor), *ConceptBase V3.1 User Manual*. University of Passau, 1992.
- [Jarke93a] M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe and Y. Vassiliou, Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis.

- Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA, January 4–6, 1993. Los Alamitos, CA: IEEE Computer Society Press, pp. 19–31.
- [Jarke93b] M. Jarke (Ed.), *Database Application Engineering with DAIDA*. Berlin: Springer-Verlag, 1993.
- [Jarke98] Matthias Jarke, “Architecture and Quality in Data Warehouses.” EDBT 1998.
- [Jeusfeld90] Manfred A. Jeusfeld, Michael Mertikas, Ingrid Wetzel, Matthias Jarke, Joachim W. Schmidt, Database Application Development as an Object Modeling Activity. In Dennis McLeod, Ron Sacks-Davis and Hans Schek (Editors), *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, 13–16 August 1990, pp. 442–454.
- [Johnson91] W. L. Johnson, M. S. Feather, D. R. Harris and K. M. Benner, “Representation and Presentation of Requirements Knowledge.” Manuscript, USC/Information Sciences Institute, Oct. 1991.
- [Juran64] J. M. Juran, *Managerial Breakthrough*, New York: McGraw-Hill Book Co., 1964.
- [Juran79] J. M. Juran, Frank M. Gryna Jr. and R. S. Bingham Jr. (Eds.), *Quality Control Handbook*, 3rd Ed., New York: McGraw-Hill Book, 1979.
- [Jurisica97] Igor Jurisica, *Representation and Management Issues for Case-Based Reasoning Systems*. Ph.D. thesis, Dept. of Computer Science, University of Toronto, 1997.
- [Jurisica98] Igor Jurisica and Brian A. Nixon, “Building Quality into Case-Based Reasoning Systems.” In Barbara Pernici and Costantino Thanos (Editors), *Advanced Information Systems Engineering*, 10th International Conference, CAiSE’98, Pisa, Italy, June 1998, Proceedings. Berlin: Springer, 1998, pp. 363–380.
Proceedings, CAiSE’98, The 10th Conference on Advanced Information Systems Engineering. Pisa, Italy, June 8–12, 1998.
- [Kaindl97] H. Kaindl, “A Practical Approach to Combining Requirements Definition and Object-Oriented Analysis.” *Annals of Software Engineering*, Vol. 3, 1997, pp. 319–343.
- [Kant79] Elaine Kant, “A Knowledge-Based Approach to Using Efficient Estimation in Program Synthesis.” *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Tokyo, August 20–23, 1979, pp. 457–462.
- [Kant81] Elaine Kant and David R. Barstow, “The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis.” *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 5, Sept. 1981, pp. 458–471.
- [Kant83] Elaine Kant, On the Efficient Synthesis of Efficient Programs. *Artificial Intelligence*, Vol. 20, No. 3, May 1983, pp. 253–305.
- [Kaiser87] G. Kaiser and P. Feiler, “An Architecture for an Intelligent Assistant in Software Development,” *Proc. International Conf. on Software Engineering*, Monterey, 1987.
- [Karger88] P. A. Karger, “Implementing Commercial Data Integrity with Secure Capabilities,” *Proc. IEEE Symposium on Security and Privacy*, 1988, pp. 130–139.

- [Kazman94] R. Kazman, L. Bass, G. Abowd and M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures," *Proc. Int. Conf. on Software Engineering*, May 1994, pp. 81–90.
- [Keller90] Steven E. Keller, Laurence G. Kahn and Roger B. Panara, "Specifying Software Quality Requirements with Metrics," In *Tutorial: System and Software Requirements Engineering*, Richard H. Thayer and Merlin Dorfman (Eds.), IEEE Computer Society Press, 1990, pp. 145–163.
- [Kim89] Won Kim, Kyung-Chang Kim and Alfred Dale, "Indexing Techniques for Object-Oriented Databases." In Won Kim and Frederick H. Lochovsky (Editors), *Object-Oriented Concepts, Databases, and Applications*. New York: ACM Press, and Reading, MA: Addison-Wesley, 1989. pp. 371–394.
- [Kogure83] M. Kogure and Y. Akao, Quality Function Deployment and CWQC in Japan. *Quality Progress*, Oct. 1983, pp. 25–29.
- [Koubarakis89] Manolis Koubarakis, John Mylopoulos, Martin Stanley and Alex Borgida, *Telos: Features and Formalization*. Technical Report KRR-TR-89-4, Dept. of Computer Science, University of Toronto, 1989.
- [Kramer95] Bryan M. Kramer, RepBrowser tool and documentation. Dept. of Computer Science, University of Toronto, 1995.
- [Laudon86] K. C. Laudon, "Data Quality and Due Process in Large Interorganizational Record Systems," *CACM*, 29(1), Jan. 1986, pp. 4–11.
- [Lauzon91] David Lauzon, Taxis-to-Telos Translator. Software developed at University of Toronto, June 1991.
- [Lawton98] G. Lawton, "Biometrics: A New Era in Security." *IEEE Computer*, Vol. 31, No. 8, Aug. 1998.
- [Lazowska84] Edward D. Lazowska, John Zahorjan, G. Scott Graham and Kenneth C. Sevcik, *Quantitative System Performance*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [A. Lee89] A. S. Lee, "A Scientific Methodology for MIS Case Studies," *MIS Quarterly*, March, 1991, pp. 30–50.
- [E. Lee92] E. S. Lee, P. I. P. Boulton, B. W. Thomson, and R. E. Soper, *Composable Trusted Systems*, Tech. Report CSRI-272, Computer Systems Research Institute, Univ. of Toronto, May 31, 1992.
- [J. Lee90] Jintae Lee, SIBYL: A Qualitative Decision Management System. In P. H. Winston and S. A. Shellard (Editors), *Artificial Intelligence at MIT: Expanding Frontiers*, Vol. 1, Cambridge, MA: The MIT Press, 1990, pp. 105–133.
- [J. Lee91] J. Lee, "Extending the Potts and Bruns Model for Recording Design Rationale," *Proc., 13th Int. Conf. on Software Eng.*, Austin, Texas, May 13–17, 1991, pp. 114–125.
- [Leveson86] N. G. Leveson, "Software Safety: Why, What, and How." *ACM Computing Surveys*, Vol. 18, No. 2, June 1986, pp. 125–163.
- [Liew90] C. W. Liew, Feedback Directed Modification of Designs, *Proc. 6th IEEE Conference on AI Applications*, 1990.
- [Ledru95] Y. Ledru, "Specification and Animation of a Bank Transfer." In *Proceedings of KBSE '95, 10th Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, 1995.

- [Loucopoulos95] P. Loucopoulos and V. Karakostas, *System Requirements Engineering*. McGraw Hill, 1995.
- [Lyu96] M. R. Lyu (ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.
- [Lunt89] T. F. Lunt and J. K. Millen, *Secure Knowledge-Based Systems*, Tech. Rep. SRI-CSL-90-04, SRI Int., Aug. 1989.
- [Marmor-Squires89] A. Marmor-Squires, B. Danner, J. McHugh, L. Nagy, D. Sterne, M. Branstad, and P. Rougeau, "A Risk Driven Process Model for the Development of Trusted Systems," *5th Annual Computer Security Applications Conf.*, Tucson, Arizona, Dec. 4-8, 1989, pp. 184-192.
- [Martin73] J. Martin, *Security, Accuracy, and Privacy in Computer Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.
- [Martin95] J. Martin and J. Odell, *Object-Oriented Methods: A Foundation*. Prentice-Hall, 1995.
- [MasterCard91] *MasterCard International Annual Report, 1990*. 1991.
- [Matthes91] F. Matthes and J.W. Schmidt, "Bulk Types: Built-In or Add-On?" In Paris Kanellakis and Joachim W. Schmidt (Editors), *Database Programming Languages: Bulk Types & Persistent Data — The Third International Workshop*. August 27-30, 1991, Nafplion, Greece. San Mateo, CA: Morgan Kaufmann, pp. 33-54.
- [Matthes92a] F. Matthes, A. Rudloff, J.W. Schmidt and K. Subieta, *The Database Programming Language DBPL: User and System Manual*. FIDE Technical Report FIDE/92/47, Fachbereich Informatik, Universität Hamburg, July 1992.
- [Matthes92b] F. Matthes and J.W. Schmidt, "Definition of the Tycoon Language — A Preliminary Report." DBIS Tycoon Report 062-92, Fachbereich Informatik, Universität Hamburg, October 1992.
- [Matthes93] F. Matthes and J.W. Schmidt, "DBPL: The System and its Environment." In M. Jarke (Ed.), *Database Application Engineering with DAIDA*. Berlin: Springer-Verlag, 1993, pp. 319-348.
- [Matloff86] N. S. Matloff, "Another Look at the Use of Noise Addition for Database Security," *Proc. IEEE Symposium on Security and Privacy*, 1986, pp. 173-180.
- [McCabe87] T. J. McCabe and G. G. Schulmeyer, "The Pareto Principle Applied to Software Quality Assurance," In G. G. Schulmeyer and J. I. McManus (Eds.) *Handbook of Software Quality Assurance*, New York: Van Nostrand Reinhold, 1987, pp. 178-210.
- [Mettala92] E. Mettala and M. H. Graham (eds.), *The Domain-Specific Software Architecture Program*, Technical Report CMU/SEI-92-SR-9, Carnegie Mellon University, June 1992.
- [Macaulay96] L. Macaulay, *Requirements Engineering*. Springer Verlag, 1996.
- [Mertikas89] Michalis Mertikas, *From Requirements Specifications to Design of Information Systems*. Master's Thesis, Dept. of Computer Science, University of Crete, 1989.
- [Molnár93] Bálint Molnár, *A Methodology for Designing Responsive Information Systems*. Doctor of University Dissertation, Dept. of Mathematics and Computer Sciences, Technical University of Budapest. 1993.

- [Moss90] J. Eliot B. Moss, "Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach." In Richard Hull, Ron Morrison and David Stemple (Editors), *Proceedings of the Second International Workshop on Database Programming Languages*, 4–8 June 1989, Salishan Lodge, Gleneden Beach, Oregon. San Mateo, CA: Morgan Kaufmann, 1990, pp. 358–374.
- [Moffett88] J. D. Moffett and M. S. Sloman, "The Source of Authority for Commercial Access Control," *IEEE Computer*, vol. 21, no. 2, Feb. 1988, pp. 59–69.
- [Mostow85] Jack Mostow, "Towards Better Models of the Design Process," *AI Magazine*, Vol. 6, No. 1, pp. 44–57, Spring 1985.
- [Motro89] A. Motro, "Integrity = Validity + Completeness", *ACM Trans. Database Sys.*, vol. 14, no. 4, 1989, pp. 480–502.
- [Musa87] J. Musa, A. Iannino and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, New York: McGraw-Hill, 1987.
- [Mylopoulos80] John Mylopoulos, Philip A. Bernstein and Harry K. T. Wong, A Language Facility for Designing Database-Intensive Applications. *ACM TODS*, Vol. 5, No. 2, June 1980, pp. 185–207.
- [Mylopoulos86] John Mylopoulos, Alex Borgida, Sol Greenspan, Carlo Meghini and Brian Nixon, Knowledge Representation in the Software Development Process: A Case Study. In H. Winter (Ed.), *Artificial Intelligence and Man-Machine Systems*. Lecture Notes in Control and Information Sciences, No. 80. Berlin: Springer-Verlag, 1986, pp. 23–44. Reprinted in Technical Note CSRI-43, Computer Systems Research Institute, University of Toronto, January 1987.
- [Mylopoulos90] John Mylopoulos, Alex Borgida, Matthias Jarke and Manolis Koubarakis, "Telos: Representing Knowledge about Information Systems." *ACM Transactions on Information Systems*, Vol. 8, No. 4, Oct. 1990, pp. 325–362.
- [Mylopoulos92a] John Mylopoulos, Lawrence Chung and Brian Nixon, "Representing and Using Non-Functional Requirements: A Process-Oriented Approach." *IEEE Transactions on Software Engineering*, Special Issue on Knowledge Representation and Reasoning in Software Development, Vol. 18, No. 6, June 1992, pp. 483–497.
- [Mylopoulos92b] John Mylopoulos, Vinay K. Chaudhri, Dimitris Plexousakis and Thodoros Topaloglou, "A Performance-Oriented Approach to Knowledge Base Management." *Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, MD, November 8–11, 1992.
- [Mylopoulos92c] John Mylopoulos, "The PSN Tribe." *Computers & Mathematics with Applications*, Vol. 23, Nos. 2–5, Jan.–March 1992, pp. 223–241.
- [Mylopoulos95] J. Mylopoulos, E. Yu, Y. Lesperance, B. Nixon, L. Chung, H. Levesque and R. Reiter, "Business Processes: Tools for Analysis and Re-design." Software demonstration at *CASCON '95 Conference*, Toronto, Nov. 1995.
- [Mylopoulos96] John Mylopoulos, Vinay K. Chaudhri, Dimitris Plexousakis, Adel Shrufi and Thodoros Topaloglou, "Building Knowledge Base Management Systems: A Progress Report." In *VLDB Journal*, Vol. 5, No. 4, 1996, pp. 238–263.
- [Mylopoulos97] J. Mylopoulos, A. Borgida and E. Yu, "Representing Software Engineering Knowledge." *Automated Software Engineering*, Vol. 4, No. 3, July 1997, pp. 291–317. Kluwer Academic Publishers.

- [**Mylopoulos98**] John Mylopoulos, "Information Modelling in the Time of the Revolution." *Information Systems*, Vol. 23, Nos. 3–4, 1998, pp. 127–155.
- [**Mylopoulos99**] J. Mylopoulos, L. Chung and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis." *Communications of the ACM*, 1999.
- [**NIST90**] "Draft #2 July 23, 1990, Guidelines and Recommendations on Integrity," prepared for the 3rd *Integrity Workshop*, National Institute of Standards and Technology, Sept. 26, 1990.
- [**Neumann86**] P. G. Neumann, "On Hierarchical Designs of Computer Systems for Critical Applications," *IEEE Trans. Software Eng.*, SE-12, no. 9, Sept. 1986, pp. 905–920.
- [**Neumann91**] P. G. Neumann (compiled), "Illustrative Risks to the Public in the Use of Computer Systems and Related Technology," *Software Engineering Notes*, vol. 16, no. 1, Jan. 1991, pp. 2–9.
- [**Nielsen93**] J. Nielsen and R. L. Mack (eds.), *Usability Inspection Methods*. John Wiley & Sons, Inc. 1993
- [**Nilsson71**] N. Nilsson, *Problem-Solving Methods in Artificial Intelligence*. New York, McGraw-Hill, 1971.
- [**Ning93**] Jim Q. Ning, Kanth Miriyala and W. (Voytek) Kozaczynski, "An Architecture-driven, Business-specific, and Component-based Approach to Software Engineering," *Proc. Int. Conf. on Software Reusability*, 1993.
- [**Nixon83**] Brian Andrew Nixon, *A Taxis Compiler*. M.Sc. Thesis, Dept. of Computer Science, University of Toronto, April 1983. Also CSRG Technical Note 33, May 1983.
- [**Nixon84**] Brian Nixon (editor), *Taxis '84: Selected Papers*. Technical Report CSRG-160, Computer Systems Research Group, University of Toronto, June 1984.
- [**Nixon87**] Brian Nixon, Lawrence Chung, David Lauzon, Alex Borgida, John Mylopoulos and Martin Stanley, "Implementation of a Compiler for a Semantic Data Model: Experiences with Taxis." In Umeshwar Dayal and Irv Traiger (Editors), *ACM SIGMOD '87, Proceedings of Association for Computing Machinery Special Interest Group on Management of Data, 1987 Annual Conference*, San Francisco, CA, May 27–29, 1987, *SIGMOD Record*, Vol. 16, No. 3, Dec. 1987, pp. 118–131.
- [**Nixon89**] Brian A. Nixon, K. Lawrence Chung, David Lauzon, Alex Borgida, John Mylopoulos and Martin Stanley, "Design of a Compiler for a Semantic Data Model." In Joachim W. Schmidt and Costantino Thanos (Editors), *Foundations of Knowledge Base Management*. Berlin: Springer-Verlag, 1989, pp. 293–343. Also Technical Note CSRI-44, Computer Systems Research Institute, University of Toronto, May 1987 (Incorporates material from [Nixon87] and [Chung88]).
- [**Nixon90**] Brian Nixon and John Mylopoulos, "Integration Issues in Implementing Semantic Data Models." In François Bancilhon and Peter Buneman (Editors), *Advances in Database Programming Languages*. New York: ACM Press, and Reading, MA: Addison-Wesley, 1990, pp. 187–217. (Workshop on Database Programming Languages, Roscoff, France, September 1987.)
- [**Nixon91**] Brian Nixon, "Implementation of Information System Design Specifications: A Performance Perspective." In Paris Kanellakis and Joachim W. Schmidt (Editors), *Database Programming Languages: Bulk Types & Persistent Data —*

- The Third International Workshop.* August 27–30, 1991, Nafplion, Greece. San Mateo, CA: Morgan Kaufmann, 1992, pp. 149–168.
- [Nixon93] Brian A. Nixon, “Dealing with Performance Requirements During the Development of Information Systems.” *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA, January 4–6, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1992, pp. 42–49.
- [Nixon94a] Brian A. Nixon, “Representing and Using Performance Requirements During the Development of Information Systems.” In Matthias Jarke, Janis Bubenko, Keith Jeffery (Eds.), *Advances in Database Technology — EDBT '94*, 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 1994, Proceedings. Berlin: Springer-Verlag, 1994, pp. 187–200.
- [Nixon94b] Brian A. Nixon, Some Possibilities for Integrating Research Results from Database Programming Languages and Information System Quality Requirements. Note written at Fachbereich Informatik, Universität Hamburg, April 1994.
- [Nixon97a] Brian Andrew Nixon, *Performance Requirements for Information Systems*. Ph.D. Thesis, Dept. of Computer Science, University of Toronto, 1997. Also forthcoming DKBS technical report.
- [Nixon97b] Brian A. Nixon, “An Approach to Dealing with Performance Requirements During Information System Development.” Extended Abstract. *Proceedings of the Doctoral Consortium of the Third IEEE International Symposium on Requirements Engineering*. Annapolis, MD, U.S.A., January 6, 1997, pp. 73–88 (Includes material from [Nixon97a]).
- [Nixon98] Brian A. Nixon, “Managing Performance Requirements for Information Systems.” In *Proceedings of the First International Workshop on Software and Performance, WOSP '98*, Santa Fe, New Mexico, U.S.A., October, 1998, pp. 131–144 (An extension and revision of [Nixon97b]).
- [Nixon99] Brian A. Nixon, “Management of Performance Requirements for Information Systems.” Manuscript, April 1999 (Includes material from [Nixon98]).
- [Nuseibeh93] B. Nuseibeh, J. Kramer and A. Finkelstein, “A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications,” *IEEE Trans. on Software Engineering*, Vol. 20, No. 10, Oct. 1994, pp. 760–773.
- [Oakland89] J. S. Oakland, “Total Quality Management,” *Proc. 2nd International Conference on Total Quality Management*, Oxford: Cotswold Press Ltd., 1989, pp. 3–17.
- [Olle82] T. W. Olle, H. G. Sol and A. A. Verrijn-Stuart (editors), *Information Systems Design Methodologies: A Comparative Review*. North-Holland, 1982.
- [Ontario80] “The Ontario Health Insurance Plan Computer System,” Chapter 17, Volume II. In *Report on the Commission of Inquiry into the Confidentiality of Health Information*, Vol. I, II, III. Toronto: J. C. Thatcher, Queen’s Printer for Ontario, Sept. 30, 1980.
- [Opdahl92] Andreas Lothe Opdahl, *Performance Engineering During Information System Development*. Dr.ing. thesis and Report 1992:5, Information Systems Group, Faculty of Computer Science and Electrical Engineering, Norwegian Institute of Technology, University of Trondheim, Norway, November 19, 1992.

- [**Opdahl94**] Andreas L. Opdahl, "Requirements Engineering for Software Performance." *Proceedings of REFSQ'94, the First International Workshop on Requirements Engineering: Foundation of Software Quality.* Utrecht, The Netherlands, June 1994, pp. 16–32.
- [**Parker84a**] Donn B. Parker and Susan H. Nycum, "Computer Crime," *Communications of the ACM*, vol. 27, no. 4, Apr. 1984, pp. 313–315.
- [**Parker84b**] Donn B. Parker, "The Many Faces of Data Vulnerability," *IEEE Spectrum*, vol. 21, no. 5, May 1984, pp. 46–49.
- [**Parker91**] D. B. Parker, "Restating the Foundation of Information Security," *2nd Annual North American Information System Security Symp.*, Oct. 21–23, Toronto, 1991.
- [**Parmakson93**] P. Parmakson, *Representation of Project Risk Management Knowledge*. M.Sc. Thesis, Institute of Informatics, Tallinn Technical Univ., Tallinn, Estonia, 1993.
- [**Parnas72**] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, Dec. 1972, pp. 1053–1058.
- [**Perry84**] T. S. Perry and P. Wallich, "Can Computer Crime be Stopped?", *IEEE Spectrum*, Vol. 21, No. 5, May 1984, pp. 34–45.
- [**Perry92**] Dewayne E. Perry and Alexander L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, 1992, pp. 40–52.
- [**Peckham88**] Joan Peckham and Fred Maryanski, "Semantic Data Models." *Computing Surveys*, Vol. 20, No. 3, Sept. 1988, pp. 153–189.
- [**Pfleeger89**] Charles P. Pfleeger, *Security in Computing*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [**Picciotto87**] J. Picciotto, "The Design of an Effective Auditing Subsystem," *Proc. Symposium on Security and Privacy*, 1987, pp. 13–22.
- [**Pohl96**] K. Pohl, *Process-Centered Requirements Engineering*. Wiley, 1996.
- [**Potts88**] C. Potts and G. Bruns, Recording the Reasons for Design Decisions, *Proc., 10th Int. Conf. on Software Eng.*, 1988, pp. 418–427.
- [**Potts94**] C. Potts, K. Takahashi and A. Anton, "Inquiry-Based Requirements Analysis." *IEEE Software*, March 1994, pp. 21–32.
- [**Pressman95**] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th Edition, New York: McGraw-Hill, 1995.
- [**Quatrani98**] T. Quatrani, *Visual Modeling with Rational ROSE and UML*, Addison-Wesley, 1998.
- [**Rabitti88**] F. Rabitti, D. Woeld and W. Kim, "A Model of Authorization for Object-Oriented and Semantic Databases," *Proc. Int. Conf. EDBT*, 1988.
- [**Ramesh92**] Balasubramaniam Ramesh and Vasant Dhar, Supporting Systems Development by Capturing Deliberations During Requirements Engineering. *IEEE Transactions on Software Engineering*, ent. Vol. 18, No. 6, June 1992, pp. 498–510.
- [**Rantfil78**] Robert M. Rantfil, *R & D Productivity*, 2nd edition. Los Angeles: Hughes Aircraft Co., 1978

- [Reubenstein90] Howard Reubenstein, *Automated Acquisition of Evolving Informal Descriptions*, Ph.D. Thesis, Also Tech. Report 1205, MIT Artificial Intelligence Laboratory, 1990.
- [Reubenstein91] Howard B. Reubenstein and Richard C. Waters, The Requirements Apprentice: Automated Assistance for Requirements Acquisition. *IEEE Transactions on Software Engineering*, Vol. 17, No. 3, March 1991, pp. 226–240.
- [Revenue Canada80] Revenue Canada, Taxation, *Objections and Appeals*, Information Circular 80–7. Ottawa, June 30, 1980.
- [Revenue Canada89] Revenue Canada, Taxation, *Inside Taxation*. Ottawa, 1989.
- [Revenue Canada91] Revenue Canada, Taxation, *1991 Taxation Statistics*, Analyzing 1989 T1 Individual Tax Returns and Miscellaneous Statistics. Ottawa, 1991.
- [Revenue Canada92a] Revenue Canada, Taxation, *Taxation Operations Manual*, Vol. 70: Objections and Appeals. Ottawa, 1992.
- [Revenue Canada92b] Dept. of National Revenue, Taxation, Appeals Branch, *Quarterly Statistical Report for the Period Ended March 31, 1992*. Ottawa, 1992. Also reports for the (quarterly) periods ended: June 2, 1991; September 27, 1991; and Jan. 3, 1992.
- [Revenue Canada92c] Dept. of National Revenue, Taxation, Appeals Branch, *Number of days to reassess Notices of Objection for reassessments between January 1, 1992 and March 1992*. Ottawa, 1992. Also reports for the periods between: Oct. 1, 1991 and Dec. 31, 1991; Apr. 1, 1992 and June 26, 1992; and June 27, 1992 and Sept. 25, 1992.
- [Rios Zertuche90] Daniel Rios Zertuche and Alejandro P. Buchmann, *Execution Models for Active Database Systems: A Comparison*. Technical Memorandum TM-0238-01-90-165, GTE Laboratories, Waltham, MA, January 30, 1990.
- [Robinson90] William N. Robinson, “Negotiation Behavior During Requirement Specification.” *Proceedings, 12th International Conference on Software Engineering*, Nice, France, March 26–30, 1990, pp. 268–276.
- [Rockart91] J. F. Rockart and J. E. Short, “The Networked Organization and the Management of Interdependence,” In M. Scott Morton (ed.), *The Corporation of the 1990's – Information Technology and Organizational Transformation*, 1991.
- [Roman85] Gruia-Catalin Roman, “A Taxonomy of Current Issues in Requirements Engineering,” *IEEE Computer*, Vol. 18, No. 4, Apr. 1985, pp. 14–23.
- [Ross77] D. T. Ross and K. E. Shoman, “Structured Analysis for Requirements Definition.” *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, January 1977, pp. 6–15.
- [Rozen91] Steve Rozen and Dennis Shasha, Rationale and Design of BULK. In Paris Kanellakis and Joachim W. Schmidt (Editors), *Database Programming Languages: Bulk Types & Persistent Data — The Third International Workshop*. August 27–30, 1991, Nafplion, Greece. San Mateo, CA: Morgan Kaufmann, pp. 71–85.
- [Rozen93] Steve Rozen, *Automating Physical Database Design: An Extensible Approach*. Ph.D. Dissertation, Dept. of Computer Science, New York University, 1993.

- [Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy & W. Lorensen, *Object-Oriented Modelling and Design*, Prentice Hall, 1991.
- [Schell83] R. R. Schell, "Evaluating Security Properties of Computer Systems," *Proc. IEEE Symposium on Security and Privacy*, 1983, pp. 89–95.
- [Schell87] Roger R. Schell and Dorothy E. Denning, "Integrity in Trusted Database Systems," In Marshall D. Abrams and Harold J. Podell (Eds.) *Tutorial: Computer and Network Security*, IEEE Computer Society Press, 1987, pp. 202–208.
- [Schmidt88] J. W. Schmidt, H. Eckhardt, F. Matthes, *DBPL Report*. DBPL Memo 112–88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt am Main, Draft, November 1988.
- [Schmidt89] Joachim W. Schmidt, Ingrid Wetzel, Alexander Borgida, John Mylopoulos, Database Programming by Formal Refinement of Conceptual Designs. *Data Engineering*, Special Issue on Database Programming Languages, Vol. 12, No. 3, Sept. 1989, pp. 53–61.
- [Schmidt94] J.W. Schmidt and F. Matthes, The DBPL Project: Advances in Modular Database Programming. *Information Systems*, 1994.
- [Schneider92] G. M. Schneider, J. Martin, and W. T. Tsai, "An Experimental Study of Fault Detection in User Requirements Documents", *ACM Trans. on Software Eng. and Methodology*, 1(2), Apr. 1992, pp. 188–204.
- [Schulmeyer87] G. Gordon Schulmeyer, "Software Quality Lessons from the Quality Experts," In G. Gordon Schulmeyer and James I. McManus (Eds.) *Handbook of Software Quality Assurance*, New York: Van Nostrand Reinhold, 1987, pp. 25–45.
- [Sevcik81] Kenneth C. Sevcik, Data Base System Performance Prediction Using an Analytical Model. *Proceedings, Seventh International Conference on Very Large Data Bases*, Cannes, France, September 9–11, 1981, pp. 182–198.
- [Shaw89] M. Shaw, "Larger Scale Systems Require Higher-Level Abstractions," *Proc., 5th International Workshop on Software Specification and Design*, Pittsburgh, PA, May 1989, pp. 143–146.
- [Shlaer88] *Object-Oriented Systems Analysis: Modeling the World in Data*. S. Shlaer & S. Mellor, Prentice-Hall, 1988.
- [Shlaer92] *Object Lifecycles: Modeling the World in States*. S. Shlaer & S. Mellor, Prentice-Hall, 1992.
- [Simon81] Herbert A. Simon, *The Sciences of the Artificial*, Second Edition. Cambridge, MA: The MIT Press, 1981.
- [C. Smith86] Connie U. Smith, Independent General Principles for Constructing Responsive Software Systems. *ACM Transactions on Computer Systems*, Vol. 4, No. 1, Feb. 1986, pp. 1–31.
- [C. Smith90] C. U. Smith, *Performance Engineering of Software Systems*. Reading, MA: Addison-Wesley, 1990.
- [D. Smith85] D. R. Smith, G. B. Kotik, and S. J. Westfold, "Research on Knowledge-Based Software Environments at Kestrel Institute," *IEEE Transactions on Software Engineering*, Special Issue on Artificial Intelligence and Software Engineering, vol. SE-11, no. 11, Nov. 1985, pp. 1278–1295.

- [G. Smith89] G. W. Smith, "Multilevel Secure Database Design: A Practical Application." *5th Annual Computer Security Applications Conf.*, Tucson, Arizona, Dec. 4–8, 1989, pp. 314–321.
- [J. Smith77] John Miles Smith and Diane C. P. Smith, "Database Abstractions: Aggregation and Generalization." *ACM TODS*, Vol. 2, No. 2, June 1977, pp. 105–133.
- [J. Smith83] John M. Smith, Stephen A. Fox and Terry A. Landers, *ADAPLEX: Rationale and Reference Manual*. Technical Report CCA-83-08, Computer Corporation of America, Cambridge, MA, May 1983.
- [Sommerville92] I. Sommerville, *Software Engineering*. fourth edition, Addison-Wesley, 1992.
- [Spivey87] J. Spivey, *The Z Notation Reference Manual*, Programming Research Group, Oxford University, 1987.
- [Stanley95] Martin T. Stanley, *Telos sh* tool and documentation. Dept. of Computer Science, University of Toronto, 1995.
- [Steinke90] G. Steinke, "Design Aspects of Access Control in a Knowledge Base System," *SECURICOM '90*, Paris, March, 1990.
- [Sterne91] D. F. Sterne, "On the Buzzword 'Security Policy,'" *Proc. IEEE Symposium on Security and Privacy*, 1991, pp. 219–230.
- [Sullivan86] L.P. Sullivan, Quality Function Deployment. *Quality Progress*, June 1986, pp. 39–50.
- [Sutcliffe96] Alistair Sutcliffe, "A Conceptual Framework for Requirements Engineering." *Requirements Engineering*, Vol. 1, No. 3, 1996, pp.170–189.
- [Svanks81] M. I. Svanks, *Integrity Analysis: A Methodology for EDP Audit and Data Quality Control*, Ph. D. Thesis, Dept. of Computer Science, University of Toronto, 1981.
- [TCSEC85] U.S. Department of Defense, *Trusted Computer Systems Evaluation Criteria*, DOD 5200.28-STD, Dec. 1985.
- [TDL87] *The Taxis Design Language (TDL)*. Final Version on TDL Design. Esprit Project 892, DAIDA, Development of Advanced Interactive Data Intensive Applications. Deliverable DES1.2, Sept. 1987.
- [Taguchi90] Genichi Taguchi and Don Clausing, "Robust Quality," *Harvard Business Review*, Jan. – Feb. 1990, pp. 65–75.
- [Tan89] Yang Meng Tan, A Program Design Assistant. Manuscript, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, June 1989.
- [Teitelman81] T. Teitelman, and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, Vol. 24, No. 9, Sept. 1981.
- [Thomson88] B. Thomson, E. S. Lee, P. I. P. Boulton, M. Stumm and D. M. Lewis, *A Trusted Network Architecture*, Computer Systems Research Institute, Univ. of Toronto, Oct. 1988.
- [Thayer90] R. Thayer and M. Dorfman (eds.) *System and Software Requirements Engineering*. IEEE Computer Society Press, 1990.

- [**Tompkins86**] F. G. Tompkins and R. Rice, "Integrating Security Activities Into the Software Development Life Cycle and the Software Quality Assurance Process," *Computers & Security*, vol. 5, no. 3, Sept. 1986, pp. 218–242.
- [**Toulmin58**] S. Toulmin, *The Uses of Argument*. Cambridge, England: Cambridge University Press, 1958.
- [**Tsur84**] Shalom Tsur and Carlo Zaniolo, An Implementation of GEM — Supporting a Semantic Data Model on a Relational Back-end. In Beatrice Yormark (Editor), *SIGMOD '84 Proceedings*, Boston, MA, June 18–21, 1984, *SIGMOD Record*, Vol. 14, No. 2, pp. 286–295.
- [**van Lamsweerde95**] A. van Lamsweerde, R. Darimont and P. Massonet, "Goal-Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt." *RE '95, The Second IEEE International Symposium on Requirements Engineering*, York, England, 27–29 March 1995, Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 194–203.
- [**Vassiliou90**] Y. Vassiliou, M. Marakakis, P. Katalagarianos, L. Chung, M. Mertikas and J. Mylopoulos, IRIS — A Mapping Assistant for Generating Designs from Requirements. In *Proceedings, CAiSE '90: The 2nd Nordic Conference on Advanced Information Systems Engineering*, Stockholm, May 1991, pp. 307–338.
- [**Visa Canada90**] Visa Canada Association, *Visa Canada 1990 Regional Report*. Toronto, 1990.
- [**Visa International91**] Visa International, *1990 Annual Report*, Canada Region, 1991
- [**Waters85**] Richard C. Waters, "The Programmer's Apprentice: A Session with KBEmacs." *IEEE Transactions on Software Engineering*, Vol. 11, Nov. 1985, pp. 1296–1320.
- [**Weddell87**] Grant E. Weddell, *Physical Design and Query Optimization for a Semantic Data Model (assuming memory residence)*. Ph.D. Thesis, Dept. of Computer Science, University of Toronto, April, 1987. Also Technical Report CSRI-198, Computer Systems Research Institute, University of Toronto.
- [**Westfold84**] S. J. Westfold, "Very-High-Level Programming of Knowledge Representation Schemes," *Proc. AAAI*, 1984, pp. 344–349.
- [**Wetzel93**] I. Wetzel, K.-D. Schewe, J.W. Schmidt and A. Borgida, Specification and Refinement of Databases. In M. Jarke (Ed.), *Database Application Engineering with DAIDA*. Berlin: Springer-Verlag, 1993, pp. 283–318.
- [**Wieringa97**] R. Wieringa, *Requirements Engineering: Frameworks for Understanding*. Wiley, 1997.
- [**Winograd86**] *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Pub. Corp., 1986.
- [**Wong81**] Harry K. T. Wong, *Design and Verification of Interactive Information Systems Using TAXIS*. Technical Report CSRG-129, Computer Systems Research Group, University of Toronto, April 1991.
- [**Wood80**] C. Wood, E. B. Fernandez and R. C. Summers, "Data Base Security: Requirements, Policies, and Models," *IBM Syst. J.*, vol. 19, no. 2, 1980, pp. 229–252.

- [Woodie83] P. E. Woodie, "Security Enhancement through Product Evaluation," *Proc. IEEE symposium on Security and Privacy*, 1983, pp. 96–101.
- [Wordsworth87] J. Wordsworth, *A Z Development Method*, IBM UK Lab Ltd., Winchester, 1987.
- [Yakemovic90] K. C. B. Yakemovic and J. Conklin, "Observations on a Commercial Use of an Issue-Based Information System", *Proc., Computer Supported Cooperative Work*, 1990, pp. 105–118.
- [Yin89] Robert K. Yin, "Research Design Issues in Using the Case Study Method to Study Management Information Systems." In James I. Cash, Jr. and Paul R. Lawrence (editors), *The Information Systems Research Challenge: Qualitative Research Methods, Volume 1*. Harvard Business School Research Colloquium. Boston: Harvard Business School, 1989, pp. 1–6.
- [Yu93a] Eric S. K. Yu and John Mylopoulos, "An Actor Dependency Model of Organizational Work — With Application to Business Process Reengineering." *Proceedings of Conference on Organizational Computing Systems*, Milpitas, CA, Nov. 1–4, 1993, pp. 258–268.
- [Yu93b] Eric S. K. Yu, "Modelling Organizations for Information Systems Requirements Engineering." *Proceedings of the IEEE International Symposium on Requirements Engineering*, San Diego, CA, January 4–6, 1993. Los Alamitos, CA: IEEE Computer Society Press, 1992, pp. 34–41
- [Yu93c] E. Yu, An Organization Modelling Framework for Information Systems Requirements Engineering, *Proc., 3rd Workshop on Info. Tech. and Systems, (WITS'93)*, Orlando, Florida, USA, December 4–5 1993, pp. 172–179.
- [Yu94a] E. S. K. Yu and J. Mylopoulos, 'Understanding "Why" in Software Process Modelling, Analysis, and Design.' *Proc., 16th Int. Conf. on Software Eng.*, Sorrento, Italy, May 1994, pp. 159–168.
- [Yu94b] Eric S. K. Yu, *Modelling Strategic Relationships for Process Reengineering*. Ph.D. Thesis, Dept. of Computer Science, University of Toronto, Dec. 1994. Also Technical Repoprt DKBS-TR-94-6.
- [Yu94c] E. Yu and J. Mylopoulos, "Towards Modelling Strategic Actor Relationships for Information Systems Development – with Examples from Business Process Reengineering." In P. De and C. Woo (eds.), *Proceedings, 4th Workshop on Information Technologies and Systems (WITS'94)*, Vancouver, B.C., December 1994, pp. 21–28.
- [Yu94d] E. Yu and J. Mylopoulos, "Using Goals, Rules, and Methods To Support Reasoning in Business Process Reengineering," *Proc. 27th Hawaii Int. Conf. System Sciences*, Maui, Hawaii, Jan. 4–7, 1994, vol. IV, pp. 234-243.
- [Yu95a] E. Yu, "Models for Supporting the Redesign of Organizational Work," *Proc., Conf. on Organizational Computing Systems (COOCS'95)*, Milpitas, California, August 13–16, 1995, pp. 225–236.
- [Yu95b] E. Yu and J. Mylopoulos, "From E-R to 'A-R' — Modelling Strategic Actor Relationships for Business Process Reengineering." *Int. Journal of Intelligent and Cooperative Information Systems*, World Scientific Publishing, Vol. 4, Nos. 2–3, 1995, pp. 125–144.
- [Yu96] Eric S. K. Yu, John Mylopoulos and Yves Lespérance, "AI Models for Business Process Reengineering." *IEEE Expert*, August 1996, pp. 16–23.

- [**Zachman87**] J. A. Zachman, “A Framework for Information Systems Architecture,” *IBM Systems Journal*, Vol. 26, No. 3, 1987, pp. 276–292.
- [**Zaniolo83**] C. Zaniolo, The Database Language GEM. *Proceedings, ACM SIGMOD Conference on Management of Data*, May 1983, pp. 207–218.
- [**Zave97**] Pamela Zave, “Classification of Research Efforts in Requirements Engineering.” *Computing Surveys*, Vol. 29, No. 4, 1997, pp. 315–321.
- [**Zisman78**] Michael D. Zisman, Use of Production Systems for Modeling Concurrent Processes. In D. A. Waterman and Frederick Hayes-Roth (Editors), *Pattern-Directed Inference Systems*. New York: Academic Press, 1978, pp. 53–68.
- [**Zultner91**] R. E. Zultner, “The Deming Way: Total Quality Management for Software.” *People & Systems Conference*, Boston, Sept. 1991.
- [**Zultner92**] R. E. Zultner, “Quality Function Deployment (QFD) for Software: Structured Requirements Exploration,” In G. G. Schulmeyer and J. I. McManus (Eds.) *Total Quality Management for Software*, New York: Van Nostrand Reinhold, 1992, pp. 297–317.

Index

- 80–20 prioritization rule, 142
80–20 rule, 347
- ability, able, 371
access authorization, 204
access group method, 203
access many attributes per tuple, 271
access many tuples per relation, 271
access rule validation, 206
accuracy and information flow, 171
accuracy argumentation methods, 180
accuracy concepts, 163
accuracy correlation catalogues, 183
accuracy correlation rules, 181
accuracy decomposition methods, 167
accuracy of information items, 163
accuracy operationalization methods, 175
accuracy operationalization methods, categorization of, 178
accuracy operationalizing softgoals, 175
accuracy refinement methods, 167
accuracy requirements, 161
accuracy requirements for information systems, 5
Accuracy Requirements Framework, 162, 194, 392
accuracy requirements, credit card study, 323
accuracy softgoals, 208, 324
accuracy study, 291, 301
accuracy topic decomposition methods, 172
accuracy type decomposition methods, 167
accuracy types, 165
accuracy, class, 166
accuracy, one to one, 166
accuracy, property, 166
accuracy, timely, 166
accuracy, value, 166
- accurate information reception method, 171
acquiring domain knowledge, 298
acquiring NFR-specific knowledge, 297
acquisition of knowledge, 297
activities (postconditions) of script transitions, 257
activities, modelling of, 368
activity (postcondition) of a script transition, 334
activity (postcondition, goal) of a Taxis script, 337
actor boundary, 376
actors, 368, 370
addressing priority softgoals, 313
administrative study, 291, 331
administrative system for government, study of, 293, 383
administrative system study, 292
administrative systems for government, 331
agent, 169
agent, source, intermediate and destination, 169
agents, 165
aggregation-decomposition methods, 173
alarm, 206
alarm method, 327
alternative techniques to achieve goals, 369, 378
ambiguities, 393
ambiguities in requirements, 323
AND contribution, 60, 61
AND interdependency, modifying label values of, 127
AND_HELPs combined contribution, 128, 173, 233
applicability of studies to broader domains, 385

- applicability of studies to specific domains, 385
- applicability of the NFR Framework, 292
- applicabilityCondition, 96
- application domain responses from experts, 386
- application to decision support systems and case-based reasoning, 394
- application to enterprise modelling and business process redesign, 292, 367
- application to software architecture, 292, 351
- application-based enforcement of integrity constraints, 274
- applications, 292, 293
- applications of the NFR Framework, 291, 393
- approach to conducting studies, 297
- archaeology mode, 300
- architects, software, 354
- architectural design alternatives, systematically guiding selection among, 352
- architectural design knowledge, 356
- architectural design, goal-driven and process-oriented, 351
- architectural design, software, 293, 351
- architectural infrastructure, 353
- architectural style, 354
- architecture concepts, cataloguing, 354
- architecture, application to software, 292, 351
- architecture, information systems, 353
- architecture, software, 393
- argument, 33
- argument, see also claim and design rationale, 33
- argumentation decomposition methods, see also refinement methods, 110
- argumentation decomposition, see also decomposition, 55, 123
- argumentation method, 120
- argumentation methods, 90
- argumentation methods and templates, 119
- argumentation methods and templates for performance, 236
- argumentation methods and templates for security, 207
- argumentation methods catalogue, 120
- argumentation methods for accuracy, 180
- argumentation methods, usage of domain characteristics, organizational information, organizational workload, system functionality, and developer expertise, 120
- argumentation, and prioritization, 120, 123
- argumentation, see also refinement, 58
- argumentative structure, 120
- arguments, identification of, 298
- assured service, see availability, 199
- attribute layer, 222, 261
- attribute method, 108
- attribute representation, 316
- attribute selection method, 173, 174, 202
- attribute, derived, 316
- attribute, in functional requirements, 51
- attributes method, 173
- attributes of softgoals, 54
- attributes, storage of, 319
- audit trail, 206
- audit trail based access rule validation, 208
- auditing, 175, 177
- auditing subsystem, 206
- authentication, 205
- authenticity, 194, 199
- authorization, 177
- availability, or assured service, 199
- bank loan system, 296
- behavioural requirements, 94
- better information flow method, 178
- biometrics, 205
- BREAK contribution, BREAKS contribution (-), 63
- building performance into systems, principles, 219, 221, 233, 285, 305
- building quality into systems, 391
- business process redesign, 393
- business process redesign and enterprise modelling, application, 292, 367
- business process reengineering, 367
- Cabinet document management system, 293, 383
- card key, 205
- case studies, 6
- case studies of a variety of NFRs, domains and information systems, 292
- case studies using the NFR Framework, 291
- case studies, methodology, 297
- case-based reasoning, application to, 393, 394
- catalogue, 5, 13, 89, 141, 381
- catalogue of knowledge, 17
- catalogue of operationalization methods, 111

- catalogue of security operationalization methods, 204
 catalogue of security types, 199
 catalogue, argumentation methods, 120
 catalogue, correlation, 89, 134
 catalogue, evaluation, 73
 catalogue, label values, 71
 catalogue, NFR decomposition methods, 91
 catalogue, NFR type, 51, 99
 catalogue, refinement, 54
 catalogue, refinement methods, 89
 catalogue, see also method catalogue, NFR Type catalogue, 17
 catalogue, see also NFR type catalogue, method catalogue, correlation catalogue, 43
 catalogues of knowledge, 381
 catalogues of methods and correlations, 381, 391
 catalogues of methods and correlations in enterprise modelling, 367
 catalogues of performance methods, organizing, 277
 cataloguing software architecture concepts, 354
 categorization method for security, 204
 centralized enforcement of integrity constraints, 274
 centring principle, 221, 232
 certification, 177
 change to the source schema, 316
 change, dealing with, 296, 353, 393
 changes in data models, 395
 channels, 165
 characteristics of confidentiality, 324
 characteristics of domains studied, 293
 characteristics of NFRs, 99, 166
 characteristics of performance requirements, 221
 characteristics of performance types, 221, 259
 characteristics of security requirements, 199
 characteristics, domain, 120
 checkpoint method, 178
 checksum, 206
 claim softgoal, 48, 52
 claim softgoal, see also claim, 33
 claim template, 120
 claim, see also argument and design rationale, 35
 claim, see also claim softgoal, 33
 claim, see claim softgoal, 33, 52
 clarification (disambiguation) of softgoals, 359
 class accuracy, see also property accuracy, 166
 class attribute method, 203
 class method, 202
 Closed list of softgoals and interdependences, 137
 clustering, 288
 code components method, 229
 code inheritance, 314
 code inheritance and IsA hierarchies, 272
 collections of items, and softgoal topic, 105
 combined (family plan) retrieval, 136
 commercial integrity, 199
 common component in refinements of an operationalization, 135
 common operations (common code) method, 273, 315
 common operations method, 273
 compartmentalization method, 203
 completeness, 195
 completeness, see integrity, 199
 component decomposition methods for performance softgoals, 227
 components, 353
 compressed format, 235, 348
 conceptual modelling, structural axes, or organizational primitives, 172
 conceptual source specifications, 250
 condition for correlation rule, 182, 208
 condition of a correlation, 131
 conducting studies, methodology for, 387
 confidentiality, 199
 confidentiality softgoals, 324
 confidentiality, or secrecy, 199
 confirmation, 175
 conflict detection, 189
 conflict, see correlation, 129
 conflicting label (\sqcap or a thunderbolt or C), 71
 conflicts, 393, 395
 connectors, 353
 conservation method, 171
 consistency, 166
 consistency checking, 177
 consistency, external, 166
 consistency, internal, 166
 consistency, internal and external, 172, 195
 constraint, 16, 96
 constraint layer, 261
 constraint, temporal, 339
 constraints, integrity, 331, 334
 consultative processes, 331, 332
 consumer-oriented attributes (software quality factors), 2, 157
 contribution, 13, 22, 85

- contribution type, 22, 26, 60
- contribution, extent, 63
- contribution, see also interdependency, 17, 48, 54
- contribution, sign, 63
- contributions, 375, 381, 391
- correct information flow methods, 169
- correctness of information, 194
- correlation, 13, 129
- correlation (implicit interdependency), 43
- correlation catalogue, 19, 33, 42, 43, 89, 130, 134, 381, 391
- correlation catalogue, specialization of, 363
- correlation catalogues for accuracy, 183
- correlation condition, 131
- correlation rule, 238
- correlation rule catalogue, 42
- correlation rule, condition, 182, 208
- correlation rules for accuracy requirements, 181
- correlation rules for performance softgoals, 238
- correlation rules for security softgoals, 207
- correlation, see also implicit interdependency, 30
- correlation, see also implicit interdependency and tradeoff, 18
- correlations, 48, 89, 358
- correlations in enterprise modelling, 367
- correlations, role of developer expertise, domain information, and functional requirements in usage, 89
- coupling NFRs with Functional Requirements, 80
- creativity of developers, 9
- credit card authorization system, 293, 301, 383
- credit card study, 291, 301
- credit card system study, 292, 293
- criteria for security evaluation, 214
- critical parts of workload, 221
- critical softgoals, 305
- critical softgoals, see also priority softgoals, 49
- criticality, 59, 204
- criticality of performance softgoals, 231
- criticality, see also prioritization, 26
- curative methods, 178
- customized solutions, 392
- cyclic checking, 274, 275, 340

- DAIDA environment for information system development, 4
- data management facilities, 253
- data management, decompositions for, 264

- data model feature decomposition methods, 275
- data model features and languages, subsets, 262
- data models, changes in, 395
- database programming languages, 395
- DBPL language, 250
- dealing with change, 393
- decentralized enforcement of integrity constraints, 274
- decision link, design, 83
- decision support systems, application to, 393, 394
- decision-making process, 51
- decisions, impact of, 137
- decomposition, 30
- decomposition methods for accuracy softgoals, 167
- decomposition methods for performance softgoals, 272, 275
- decomposition methods for security softgoals, 201
- decomposition methods, see also NFR decomposition methods, operationalization decomposition methods, and argumentation decomposition methods, 90
- decomposition of an operationalization, 58
- decomposition of operationalizing softgoals, 116
- decomposition of softgoals, 359
- decomposition on NFR type, 99
- decomposition on softgoal topic, 105
- decomposition, see also refinement, NFR decomposition, operationalization decomposition, and argumentation decomposition, 54
- decomposition, see also softgoal decomposition, 19
- defect detection, 393
- deniable softgoal, 64
- denied label (\times or D), 71
- denied softgoal, 64
- denied softgoals, 39
- dependee, 370
- dependencies, 377
- dependencies and relationships in process descriptions, strategic, 368
- dependency link, 370
- dependency, strategic, 370
- depender, 370
- dependum, 370
- derived attribute, 316
- derived information method, 172, 202
- descriptor of situations in the domain, 134
- descriptors of situations, 182, 208, 237

- descriptors, situation, 134
 design, 4, 16
 design alternative, 54
 design catalogue, see catalogue, 5
 design constraint, 54
 design decision, 47, 48
 design decision link, 83
 design decisions, 3, 5
 design knowledge catalogue (development technique catalogue), 17
 design methods, identification of, 298
 design process, target or destination of, 27
 design rationale, 48, 360, 391
 design rationale, identification of, 298
 design rationale, provision of, 299
 design rationale, represented by claim softgoals, 52
 design rationale, see also claim and argument, 33
 design technique, see also development technique, 51
 design tradeoff, 60
 design, see also development, 47
 design, software architectural, 293, 351
 destination agent, 169
 destination or target of design process, 27
 determining main (top-level) requirements, 386
 developer expertise, incorporated into argumentation, 120
 developer expertise, used in applying refinement methods and correlations, 89
 developer-defined operations, 264
 developer-defined operations method, 265
 developer-directed development process, 47
 developer-directed evaluation procedure, 71
 developer-directed process, 391
 developers, creativity of, 9
 development, 16
 development alternative, 54
 development decision, 48
 development decisions, record of, 224
 development knowledge, 54
 development methods, identification of, 298
 development technique catalogue, see also design knowledge catalogue, and catalogue, 17
 development technique, see also design technique, development technique, 51
 development technique, see method, 18
 development techniques, 224, 251
 development tradeoff, 60
 development, see also design, 48
 developmental security, 199
 dialectical style of reasoning, 8
 disambiguation (clarification) of softgoals, 359
 discretionary security, 201, 204
 distributed intentionality, 368
 diverse NFRs treated together, 326
 divide-and-conquer paradigm, and NFR decomposition methods, 91
 document management system, 293, 383
 documentation in SIG, 387
 domain characteristics, 52
 domain characteristics, incorporated into argumentation, 120
 domain characteristics, studies, 293
 domain expert feedback on studies and framework, 383, 393
 domain feedback, 384
 domain independence of general refinement methods, 104
 domain information, 25
 domain information used to resolve trade-offs, 362
 domain information used to select storage method, 320
 domain information, used in applying refinement methods and correlations, 89
 domain information, used in refinement methods, 92
 domain interviews to obtain feedback, 384
 domain knowledge, 54
 domain knowledge and proportionality of its usage, 386
 domain knowledge, acquiring, 298
 domain responses from experts, 386
 domain-independent refinement method (generic method, 104
 domain-specific correlation catalogue, 363
 domain-specific refinement, 94
 domains, NFRs and information systems, studies of a variety of, 292
 dominance, 59
 dominance, see also prioritization, 26
 dominant parts of workload, 221
 dominant performance softgoals, 231
 dominant softgoals, 313
 dominant softgoals, see also priority softgoals, 49
 dominant workload softgoals, 305
 downward inference, 132
 dynamic code inheritance, 272, 273
 dynamic offset determination, 235
 early collection and cataloguing of domain and NFR knowledge, 18
 early fixing, 233, 320

- early fixing methods, 233
- education, 396
- efficient storage, 333
- elements, 353
- elements, intentional, 368
- emphases of NFR Framework, 385
- empirical studies, 392
- empirical studies of a variety of NFRs, domains and information systems, 292
- empirical studies using the NFR Framework, 291
- encryption, 208
- enterprise modelling and business process redesign, application, 292, 367
- enterprise modelling, operationalizations in, 368
- enterprise modelling, use of catalogues in, 367
- enterprise modelling, use of correlations in, 367
- enterprise modelling, use of methods in, 367
- enterprise modelling, use of satisficing in, 367
- enterprise modelling, use of softgoal in, 367, 368
- enterprise modelling, use of softgoal interdependency graph in, 367
- entities, modelling of, 368
- entity layer, 222, 261
- entity management method, 266, 308, 316
- Entity-Relationship (E-R) model, 222, 368, 395
- epistemological features, 260
- EQUAL contribution, EQUALS contribution (=), 70
- evaluating softgoal achievement, 299
- evaluation, 37, 48
- evaluation catalogue, 73
- evaluation of framework and studies, 383, 393
- evaluation procedure, 13, 70
- evaluation procedure (labelling algorithm), 17, 37
- exception handling method, 178
- execution ordering methods, 236, 273
- exhaustive search, 274
- exhaustive subclass method, 173
- exhaustive subset method, 174
- experiment design, 388
- experimental methodology, 388
- expert consultation, 177
- expert feedback on framework and studies, 383, 393
- explicit aggregation method, 174
- explicit interdependency, 30, 48, 130
- extent of contribution, 63
- external confidentiality, 326
- external consistency, 166, 172, 195, 199
- external security, 199, 202
- factors for dealing with performance requirements, 223
- family plan, 221
- family plan (combined) retrieval, 136
- family plan method, 241
- fast access to information, 333
- features of source specification language, 224
- feedback from studies, 300
- feedback on framework and studies from domain experts, 383, 393
- few attributes per tuple, 270, 313
- figures, legend for, xix, 21
- figures, logos for, xix, 21
- fingerprint, 205
- fixing principle, 222
- fixing, early, 233
- fixing, late, 235
- flow, information, 163
- flow-through method, 229, 243, 277, 310, 320
- focussing search, 369
- follow-up service, partially automated, 332
- formality of the NFR Framework, 385
- framework and studies, feedback from domain experts, 383, 393
- framework, questionnaires, 384
- Framework, see NFR Framework, 47
- from part to whole method, 174
- functional and non-functional requirements, 370
- functional goals, 257
- functional requirement, definition, 6
- functional requirements, 1, 18, 40, 223
- functional requirements (FRs), 80
- functional requirements for a KWIC system, 352
- functional requirements, role of developer expertise, domain information, and functional requirements in usage, 89
- functional requirements, used in refinement methods, 92, 94
- functionality, 1
- functionality of system, incorporated into argumentation, 120
- further operationalization, 58
- generalization-specialization methods, 172
- generic decomposition methods, 91
- generic refinement method (domain-independent), 104

- givens (preconditions) of a script transition, 334
- givens (preconditions) of a Taxis script, 337
- givens (preconditions) of script transitions, 257
- global nature of NFRs, 51
- goal, 369, 376
- goal dependency, 371
- goal of a Taxis script, see activity, 337
- goal, compared with softgoal, 47
- goal-driven process, 137
- goal-driven, process-oriented architectural design, 351
- goal-oriented approach, 47
- goal-oriented methodologies, related work, 143
- goal-oriented requirements engineering, 148
- goals, 368, 375
- goals in logical formalisms, compared with softgoals, 7
- goals, functional, 257
- good enough, 63
- government administrative system, study of, 293, 383
- government administrative systems, 331
- graphical record for review, change and reuse, 353
- green book, 214
- grid of linguistic and organizational features, 260
- grid versus staircase of inherited attributes, 267
- harmonious interactions, 141
- harmonized criteria, 214
- harmony, see correlation, 129
- health insurance system, 296, 383
- HELP contribution, HELPS contribution (+), 63
- HELPS AND combined contribution, 128, 233
- horizontal splitting, 203, 269, 271, 320
- how else to achieve goals, 369, 378
- HURT contribution, HURTS contribution (-), 63
- hybrid code inheritance methods, 272
- i* Framework usage of NFR Framework concepts, 368
- i* Framework, i-star Framework, 368
- identification method, 205
- identification of design rationale (arguments), 298
- identification of development methods, 298
- identifying NFR softgoals, 298
- identifying NFR-related concepts, 298
- identifying priorities, 299
- impact of decisions, 137
- impact of decisions, and evaluation procedure, 70
- impact, individual, 71
- impact, see also contribution, 17
- implementation, 4, 40
- implementation components method, 227, 273, 311, 339
- implementation experience, 251
- implementation methods, identification of, 298
- implementation technique, see also development technique, 51
- implementation techniques for semantic data models, 336
- implementations, target, 250
- implicit contribution, 132
- implicit interdependency, 48, 130
- implicit interdependency, see correlation, 30, 43
- implicit interdependency, see correlation and also tradeoff, 18
- income tax appeal system study, 292
- income tax appeals study, 291
- income tax appeals system, study of, 293, 331, 383
- indexing, 234
- individual activities (postconditions) method, 276
- individual activities method, 338
- individual attribute method, 203
- individual attribute security level method, 203
- individual attributes method, 308, 320
- individual components method, 276, 337
- individual givens (preconditions) method, 276
- individual impact, 71
- individual locations method, 276
- individual transitions method, 276, 337
- individual-bulk operations method, 265
- inexhaustive subclass method, 173
- inference, downward, 132
- inference, see also downward inference and upward inference, 132
- inference, upward, 134
- informal SIG, see also softgoal interdependency graph (SIG), 21
- information accuracy, 171
- information flow, 163, 304, 392
- information flow, methods, 168
- information item, 199
- information item, accuracy of, 163
- information security, 197

- information sensitivity, 201
- information system architecture, 353
- information system development, 4
- information system development languages, 250
- information system performance requirements, 249
- information system security, 197
- information systems, 5, 392
- information systems, assumptions in, 372
- information systems, language layers for performance requirements, 259
- information systems, NFRs and domains, studies of a variety of, 292
- information systems, performance issues, 263
- inheritance hierarchies, 319
- inheritance hierarchies (IsA hierarchies), 267
- inheritance hierarchy layer, 261
- inheritance method, 203
- inputs and outputs for dealing with performance requirements, 223
- insurance claims example, 370
- integration issues for semantic data model implementations, 252
- integrity, 194, 199
- integrity constraint layer, 261
- integrity constraint, temporal, 339
- integrity constraints, 273, 331, 332, 334
- integrity constraints, enforcement, 274
- integrity constraints, operationalization methods for, 340
- intentional concepts, 369
- intentional elements, 368
- intentionality, distributed, 368
- inter-layer interdependency link, 310, 339
- inter-layer interdependency link, 310
- inter-layer interdependency links, 229
- inter-layer refinement, 277, 338, 344
- interacting nature of non-functional requirements, 7
- interaction among features of semantic data models, 252
- interaction, see also correlation, 130
- interactions, 224
- interactions (tradeoffs), 391
- interactions among data models, implementations and performance, 259
- interactive design process and evaluation procedure, 17
- interactive evaluation procedure, 71
- interactive processes, 331, 332
- interdependencies, 3, 22
- interdependency, 17, 48
- interdependency links, 17, 375, 381
- interdependency, explicit, 130
- interdependency, implicit, 130
- interdependency, see also contribution and refinement, 17
- interdependency, see also refinement and contribution, 54
- interdependency, see also refinement, contribution, 48
- interdependency, see also refinement, contribution, implicit interdependency, explicit interdependency, 30
- intermediate agent, 169
- internal confidentiality, 199, 326
- internal consistency, 166, 172, 195
- internal external method, 202
- internal security, 199, 202
- internal-external subtype method, 324
- interrelating and refining NFR concepts, 298
- interrelationships among softgoals, 48
- interviews to obtain feedback, 384
- IsA (specialization) hierarchy, 95
- IsA hierarchies, 314, 319
- IsA hierarchies (inheritance hierarchies), 267
- IsA hierarchies of transactions, 272
- IsA hierarchy layer, 261
- Key Word in Context (KWIC), 351
- kinds of softgoals, 48
- knowledge acquisition, 297
- knowledge catalogue, see catalogue of knowledge, 17
- knowledge of application domains studied, 386
- KWIC system, functional requirements for, 352
- KWIC system, NFRs for, 353
- label, 17, 40, 85
- label of softgoal, 54, 70
- label value catalogue, 71
- labelling algorithm, see evaluation procedure, 17, 37
- language definition, 251
- language layered structure for performance issues, 222
- language layers for performance requirements for information systems, 259
- languages and data model features, subsets, 262
- languages for information system development, 250
- late fixing methods, 235
- layered performance structures, 218

- layered structure for performance issues, 222
- layers for performance requirements for information systems, 259
- layers of performance softgoals, 338
- layers of softgoal interdependency graphs for performance requirements, 229
- layers, linking, 310, 379
- legend for figures, xix, 21
- libraries of existing alternatives, 395
- lifecycle issues, 387
- lighter (weak) reasoning, 8
- limiting access time, 206
- linguistic features, 260
- linking layers of a softgoal interdependency graph, 310
- linking softgoals and SIGs at different layers, 338
- location of a script, 334
- locations of a script, 257
- logos for figures, xix, 21
- logos of figures, 92
- long-term process layer, 261
- long-term process represented as a Script, 335
- long-term process, see also script, 255
- long-term processes, 331, 332, 334, 338, 339
- long-term processes (scripts), 275
- long-term processes, management time softgoals for, 336
- main (top-level) requirements, determination of, 386
- main memory, see also space performance, 221
- major responses on framework and studies from domain experts, 384
- MAKE contribution, MAKES contribution (++), 63
- management time, 258, 335
- management time softgoal, 333
- management time softgoals for long-term processes, 336
- management time softgoals, refinement to time softgoals, 276
- management time to time method, 277, 344
- ManagementTime softgoal, 336
- mandatory internal confidentiality, 327
- mandatory security, 201, 204, 211
- manual consultation, 181
- mapping process, taking functional requirements to a target system, 83
- means-ends links, 375
- medical information system application, 394
- meeting performance requirements, 224
- method, 13, 18, 23
- method application, 92
- method catalogue, 19, 42, 381, 391
- method catalogue, see also catalogue, 18
- method definition, 92
- method parameterization, degrees of, 105, 108
- method parameterization, see also type parameterization of methods, topic parameterization of methods, 94
- method parameters, 94
- methodology for conducting studies, 297, 387
- methodology for experiments, 388
- methodology responses from domain experts re studies, 387
- methods, 48, 356
- methods for decomposition on NFR type, 99
- methods for decomposition on softgoal topic, 105
- methods in enterprise modelling, 367
- methods, identification of, 298
- methods, see also refinement methods, 89
- metrics, 2, 248
- mgmt. time, see management time, 335
- minimal security, 204
- most specialized transaction version, 272
- motivations, 368
- multi-layer password, multiple password, 206
- multiple inheritance, 254
- mutual authentication, 206
- mutual ID, 183
- nature of non-functional requirements, 6
- negative contribution, see also sign of contribution, 63
- negative correlations, 130
- negative implicit contribution, 132
- NFR catalogue, see also catalogue of knowledge, 17
- NFR Characteristics, 99
- NFR concepts, refining and interrelating, 298
- NFR decomposition method catalogue, 91
- NFR decomposition methods, 90
- NFR decomposition, see also decomposition, 55
- NFR Framework, 4, 47, 391
- NFR Framework applied to the *i** Framework, 368
- NFR Framework questionnaires, 384
- NFR Framework's emphases, 385

- NFR Framework, applications, 393
- NFR Framework, empirical studies, 392
- NFR Framework, feedback on, 384
- NFR Framework, formality, 385
- NFR Framework, related work, 142
- NFR Framework, responses from domain experts, 385
- NFR Framework, scalability, 385
- NFR Framework, studies and applications, 291
- NFR Framework, training costs and payoff, 385
- NFR Framework, using, 298
- NFR questionnaires, 384
- NFR softgoal, 20, 50
- NFR softgoals, 48
- NFR softgoals at lower layers, 230, 344
- NFR softgoals, global nature, 51
- NFR softgoals, identification of, 298
- NFR softgoals, operationalization of, 112
- NFR standards, 158
- NFR type, 49, 372
- NFR type catalogue, 18, 42, 51, 99
- NFR Type catalogue, see also catalogue, 17, 18
- NFR type catalogue, used in method, 100
- NFR type decomposition methods, 99
- NFR type, see also type, 20, 21
- NFR Types, 355
- NFR, see also NFR softgoal, 20
- NFR-related concepts, identification of, 298
- NFR-specific decomposition methods, 92
- NFR-specific knowledge, acquisition of, 297
- NFRs (see non-functional requirements), 1
- NFRs during software architectural design, addressing, 353
- NFRs for a KWIC system, 353
- NFRs, diverse, treated together, 326
- NFRs, domains and information systems, studies of a variety of, 292
- NFRs, see also NFR softgoals, 48
- noise addition, 206, 207
- non-functional and functional requirements, 370
- non-functional requirement, definition, 6
- non-functional requirements, 15
- non-functional requirements (NFRs), 1
- non-functional requirements, characteristics, 166
- non-functional requirements, their nature, 6
- non-priority softgoals, 348
- observations from studies, 300
- off-line studies, 300
- offspring softgoal, 17
- omissions, 393, 395
- one to one accuracy, 166
- ontological features, 260
- Open list of softgoals and interdependencies, 137
- OperatingCost methods, 105
- operation components method, 228, 307, 316
- operational method, 264
- operational security, 199
- operationalization, 16, 27, 57
- operationalization decomposition, 58, 116
- operationalization decomposition methods, 110
- operationalization decomposition, see also decomposition, 55
- operationalization in KAOS, 147
- operationalization methods, 90, 111
- operationalization methods catalogue, 111
- operationalization methods for accuracy softgoals, 175
- operationalization methods for integrity constraints, 340
- operationalization methods for performance softgoals, 233, 273, 275
- operationalization methods for security softgoals, 204
- operationalization of NFR softgoals, 112
- operationalization type, 113
- operationalization, see also refinement, 57
- operationalization-target link, 83
- operationalizations, 51, 360, 391
- operationalizations in enterprise modelling, 368
- operationalizations refined into NFR softgoals at lower layers, 231, 379
- operationalizations, selection of, 137
- operationalize, 27, 51
- operationalized by, 27
- operationalizing softgoal, 27, 48, 51, 376
- operationalizing softgoals, 376
- operationalizing softgoals not refined into NFR softgoals, 230, 344
- operationalizing softgoals, decomposition of, 116
- operationalizing softgoals, relationship to tasks, 376
- operations of a script transition, 257
- operations, data management, 264
- opportunities in relationships, 368
- OR contribution, 61
- orange book, 214
- order of execution of operations, 310
- organizational features, 260

- organizational information, incorporated into argumentation, 120
 organizational primitives of conceptual modelling, 172
 organizational priorities, 25
 organizational priorities, incorporated into argumentation, 120
 organizational priority, 18
 organizational stakeholders, strategic relationships, 368
 organizational workload, 18, 25, 223, 257, 293
 organizational workload and performance requirements, 257
 organizational workload, incorporated into argumentation, 120
 organizing performance methods in catalogues, 277
 outputs and inputs for dealing with performance requirements, 223
- parameterization of methods, degrees of, 105, 108
 parameterization of methods, see also type parameterization of methods, topic parameterization of methods, 94
 parameters of methods, 94
 parameters, compared to softgoal topics, 49
 parent softgoal, 17
 partial contribution, see also extent of contribution, 63
 partitioning softgoal topic via Subset method, 326
 password, 205
 payback of using the NFR Framework, 387
 payoff and training cost of the NFR Framework, 385
 people and the usage of the NFR Framework, 387
 perform first method, 310
 performance argumentation methods, 236
 performance component decomposition methods, 227
 performance concepts, 218, 219
 performance correlation rules, 238
 performance decomposition methods, 272, 275
 performance issues for information systems, 263
 performance methods, organized in catalogues, 277
 performance operationalization methods, 233, 273, 275
 performance prediction, 224, 395
- performance prediction, organized by layering, 222
 performance refinement methods, 225
 performance requirements, 217
 performance requirements and layers of softgoal interdependency graphs, 229
 performance requirements for information systems, 5, 249
 Performance Requirements Framework, 247, 249, 392
 performance requirements study, 291
 performance requirements, credit card study, 305
 performance requirements, inputs and outputs for dealing with, 223
 performance requirements, language layers for information systems, 259
 performance requirements, nature of, 219
 performance requirements, tax appeals case study, 331
 performance softgoal topic decompositions, 227
 performance softgoals at lower layers, 230, 344
 performance softgoals, layers, 338
 performance softgoals, prioritization, 231
 performance study, 291, 301, 331
 performance sub-types method, 226
 performance subtype decompositions, 225
 performance type, 258
 performance types, 219
 performance types, characteristics, 221, 259
 performance, definitions, 219
 performance, layered structure for, 222
 perturbation, 206
 Petri net, 257
 physical alarm, 206
 policy manual consultation, 180, 181
 positive contribution, see also sign of contribution, 63
 positive correlations, 130
 postcondition of a script transition, see activity, 334
 postcondition of a Taxis script, see activity, 337
 postconditions of script transitions, see activities, 257
 pre-defined operations (primitive operations), 264
 precautionary methods, 178
 preconditions of a script, see givens, 334
 preconditions of a Taxis script, see givens, 337

- preconditions of script transitions, see givens, 257
- prediction of performance, 224, 395
- prediction of performance, organized by layering, 222
- preferential selection, 180
- preventative methods, 178
- primitive operations (pre-defined operations, 264
- primitive operations method, 265
- primitive—developer-defined operations method, 264
- principles, software performance engineering, 219, 221, 233, 285, 305
- priorities, 293
- priorities for system and organization, 223
- priorities, identification of, 299
- priorities, see also organizational priorities, 120
- prioritization, 33, 55, 59, 180, 204, 307, 316, 324, 347, 349
- prioritization and tradeoffs, 328
- prioritization and AND decompositions, 124
- prioritization method, see also *VitalFewTrivialMany*, *TrivialMany*, 123
- prioritization of performance softgoals, 231
- prioritization of security softgoals, 211
- prioritization of softgoals, 305, 360
- prioritization of softgoals, see also criticality and dominance, 25
- prioritization template, 327
- prioritization, and argumentation, 120, 123
- prioritization, criticality, 59
- prioritization, dominance, 59
- priority, 18
- priority based selection, 181
- priority in the domain, 52
- priority of softgoals, see also critical softgoals, dominant softgoals, 49
- priority softgoals, 305
- priority transaction, 306
- procedure manual, users', 192
- process descriptions, strategic dependencies and relationships in, 368
- process management time, 258
- process modelling, 375
- process, goal-driven, 137
- process-oriented approach, 386, 391
- process-oriented approaches to NFRs, 3
- process-oriented, goal-driven architectural design, 351
- processing versus frequency tradeoff principle, 221, 241
- product-oriented approaches to NFRs, 3
- propagation rules, 87
- proper subset method, 174
- property accuracy, also class accuracy, 166
- proportionality of use of domain knowledge, 386
- providing design rationale, 299
- qualitative and quantitative approaches, 395
- qualitative and quantitative approaches to performance, 285
- qualitative approach, 391
- qualitative approaches to NFRs, 3
- quality assurance and control, related literature, 148
- quality attribute, see also software quality attribute, non-functional requirement, 1, 15
- quality control, 148
- quality function deployment (QFD), 149
- quality of the process that produces a product, 386
- quantitative and qualitative approaches, 395
- quantitative approaches to NFRs, 3
- questionnaires on NFRs, framework and studies, 384
- rapid posting method, 326
- rationales, 368
- redesign of processes, 370
- reduce run-time reorganization method, 235
- refinement catalogue, 54
- refinement method application, 92
- refinement method catalogue, 89
- refinement method definition, 92
- refinement methods, 90
- refinement methods for accuracy softgoals, 167
- refinement methods for performance softgoals, 225
- refinement methods for security softgoals, 200
- refinement methods, and use of functional requirements, 94
- refinement methods, domain independency of general ones, 104
- refinement methods, drawing on domain information, 92
- refinement methods, drawing on functional requirements, 92
- refinement methods, role of developer expertise, domain information, and functional requirements in usage, 89
- refinement methods, see also methods, 89

- refinement, see also decomposition, operationalization, and argumentation, 54
- refinement, see also interdependency, 17, 48, 54
- refinements, 375, 381
- refining an operationalizing softgoal, 342
- refining and interrelating NFR concepts, 298
- refining softgoals, 359
- rejected softgoals, 39
- related work for the NFR Framework, 142
- relating functional requirements to NFRs, 40
- relating goal and target system, 376
- relating NFRs to functional requirements, 40
- relating source and target descriptions, 381
- relating source design and target system, 83
- relating target system to goal, 376
- relational data model layer, 260
- relational database programming language, 250
- relationships and dependencies in process descriptions, strategic, 368
- relationships, strategic, 368
- relative nature of non-functional requirements, 6
- replicate derived attribute, 271
- representation of attributes, 316
- requirements, 4
- requirements acquisition, 387
- requirements, ambiguities, 323
- resolving tradeoffs via domain information, 362
- resource, 369
- resource availability method, 180
- resource dependency, 371
- resource-related accuracy operationalization methods, 178
- resources, 368, 375
- response time, 221
- responses on application domain from experts, 386
- responses on framework and studies from domain experts, 383, 393
- responses on NFR Framework from domain experts, 385
- responses on study methodology from domain experts, 387
- responsiveness, 219, 236
- retracting a selection, 192
- Rome Air Development Center (RADC, Rome Laboratory), 2, 157
- safety, 87
- satisficeable softgoal, 64
- satisficed label (\vee or S), 71
- satisficed softgoal, 64
- satisficed softgoals, 39
- satisficing, 4, 37, 47
- satisficing approach, 9, 391
- satisficing in enterprise modelling, 367
- scalability of the NFR Framework, 385
- scheduling of scripts, 275
- schema change method, 267
- scope of protection, 198
- script, 257
- script (long-term process), 255
- script components method, 276
- script layer, 261
- Script used to represent long-term process, 335
- script, Taxis, 334
- scripts, see also long-term processes, 275
- search, exhaustive, 274
- search, selective, 274
- secondary storage, see also space performance, 221
- secrecy, see confidentiality, 199
- security argumentation templates and methods, 207
- security assurance, 198
- security auditing, 206
- security concepts, 198
- security correlation rules, 207
- security decomposition methods, 201
- security evaluation criteria, 214
- security level, security class, 201
- security levels, 211
- security operationalization method catalogue, 204
- security operationalization methods, 204
- security refinement methods, 200
- security requirements, 197
- security requirements for software systems, 5
- Security Requirements Framework, 197, 213, 392
- security requirements, characteristics, 199
- security requirements, credit card study, 323
- security softgoal, 199, 323
- security study, 291, 301
- security type catalogue, 199
- security via SubType method, 201
- security, information, 197
- selected softgoals, 39
- selection among alternatives, 363
- selection among alternatives, organized by layering, 222
- selection of operationalizations, 137
- selective attribute grouping, 269

- selective checking, 275
- selective search, 274
- semantic data model, 250
- semantic data model implementation experience, 251
- semantic data model implementation techniques, 336
- semantic data model integration issues, 252
- semantic integrity constraint layer, 261
- semantic integrity constraints, 273
- semi-automatic design evaluation procedure, 17
- sensitive information, 207
- sensitivity of information, 201
- separation of duties, 206
- several attributes per tuple, 270
- SIG, see softgoal interdependency graph, 17, 48, 85
- sign of contribution, 63
- SIGs organised in layers, 338
- SIGs, see softgoal interdependency graphs, 5
- single-multiple attribute method, 266
- situation descriptor, 53
- situation descriptors, 134, 182, 208, 237
- soft alarm, 206
- softgoal, 4, 5, 8, 17, 20, 22, 37, 48, 369, 376, 391
- softgoal achievement, evaluation of, 299
- softgoal attributes, 54
- softgoal decomposition, 20, 359
- softgoal dependency, 371
- softgoal in enterprise modelling, 367, 368
- softgoal interaction, 48
- softgoal interdependency graph (SIG), 375
- softgoal interdependencies, 5
- softgoal interdependency, 13
- softgoal interdependency graph, 392
- softgoal interdependency graph (SIG), 13, 17, 85
- softgoal interdependency graph (SIG) as concise documentation, 387
- softgoal interdependency graph in enterprise modelling, 367
- softgoal interdependency graphs, 5, 48
- softgoal interdependency graphs (SIGs), 224
- softgoal interdependency graphs at different layers, linking, 338
- softgoal interdependency graphs, conciseness, 141
- softgoal interdependency graphs, multiple layers for performance requirements, 229
- softgoal interdependenies, 5
- softgoal interrelationships, 48
- softgoal kinds, 48
- softgoal label, 54, 70
- softgoal label, see also value, 40
- softgoal priorities, 49
- softgoal prioritization, 59, 360
- softgoal refinement, 48, 54, 359, 391
- softgoal refinement (disambiguation), 359
- softgoal tag, 49
- softgoal topic, 49, 105
- softgoal topic decomposition methods, 105
- softgoal topics, compared to standard parameters, 49
- softgoal type, 49
- softgoal type and topic, 372
- softgoal, applied to organization modelling, 293
- softgoal, compared with goal, 47
- softgoals, 13, 48, 368, 375, 381
- softgoals at different layers, linking, 338, 379
- softgoals, compared with goals in logical formalisms, 7
- softgoals, kinds, 48
- softgoals, layers, 338
- software architects, 354
- software architectural design, 293, 351, 393
- software architectural design, addressing NFRs during, 353
- software architecture concepts and approaches, 353
- software architecture concepts, cataloguing, 354
- software architecture, application, 292
- software architecture, application to, 351
- software development, related work, 146
- Software Performance Engineering (SPE), 236
- software performance engineering principles, 219, 221, 233, 285, 305
- software process improvement, 393
- software process improvement, study of, 395
- software process redesign, application to, 381
- software quality, 3
- software quality assurance (SQA), 149
- software quality attribute, 2, 15
- software quality criteria (technically-oriented attributes), 2, 157
- software quality factors (consumer-oriented attributes), 2, 157
- software quality metrics, 2
- solutions, customized, 392
- solutions, see operationalizations, 51

- some negative contribution, SOME- contribution, 63
 some positive contribution (SOME+), 63
 SOME+ contribution, some positive contribution, 63
 SOME- contribution, some negative contribution, 63
 source agent, 169
 source specification, 18, 250, 251
 source specification language, features, 224
 source specification of system, 224
 space performance, 221
 space requirements, 319
 space subtypes method, 226
 SPE, see Software Performance Engineering, 236
 specialization (IsA) hierarchy, 95
 specialized, domain-specific correlation catalogue, 363
 specification, 40
 specification language, features, 224
 specificity of performance requirements, 221
 splitting, horizontal, 269
 splitting, vertical, 269
 staircase of inherited attributes, 267
 stakeholders, 369
 stakeholders, strategic relationships among organizational, 368
 standards, NFRs, 158
 static code inheritance, 272, 273
 static offset determination, 234, 313
 storage of attributes, 319
 storage requirements, 321
 strategic dependency mode, 370
 strategic dependency model, 368
 strategic interests, 370
 strategic rationale model, 368, 374
 strategic relationships, 368
 strategic relationships and dependencies in process descriptions, 368
 structural axes of conceptual modelling, 172
 structural decomposition, and topic refinement methods, 91
 structural modelling primitives, 105
 structural requirements, 94
 structured analysis, 368
 student records system, study of, 396
 studies, 292
 studies and framework, feedback from domain experts, 383, 393
 studies of a variety of NFRs, domains and information systems, 292
 studies using the NFR Framework, case, 291
 studies, applicability to broad domains, 385
 studies, applicability to specific domains, 385
 studies, empirical, 392
 studies, feedback from, 300
 studies, methodology, 297
 studies, methodology for conducting, 387
 studies, observations from, 300
 studies, questionnaires, 384
 study of performance, 331
 study of performance requirements, 291
 study of performance, accuracy and security, 291, 301
 study, administrative, 291, 331
 study, credit card, 291, 301
 study, student records system, 396
 study, telecommunications, 396
 style, architectural, 354
 sub-softgoal (subgoal), 20
 sub-type, 19
 sub-type method, performance, 226
 Subclass decomposition method, 92
 Subclass family of methods, 108
 subclass hierarchy, used in refinement method, 94
 Subclass method, 105
 subclass method, 172, 202, 326
 subgoal, see sub-softgoal, 20
 subjective nature of non-functional requirements, 6
 subset method, 174, 202, 326, 346
 subsets of languages and data model features, 262
 subtype decompositions for performance requirements, 225
 SubType family of methods, 105
 subtype method, 323
 SubType method for security, 201
 subtype method, internal-external, 324
 SubType methods, 117
 SubType refinement method, 168
 sufficient contribution, see also extent of contribution, 63
 superset method, 174
 synergistic interactions, 141
 synergy, 181
 synergy, see also correlation, 130
 system functionality, incorporated into argumentation, 120
 systematically guiding selection among architectural design alternatives, 352
 systems analysis, 368
 tag of softgoal, 49
 target, 18, 83
 target implementations, 250

- target or destination of design process, 27
- target system, 40
- task, 369
- task decomposition links, 375
- task dependency, 371
- tasks, 368, 375, 376
- tasks, relationship to operationalizing softgoals, 376
- tax appeals study, 291
- tax appeals system, study of, 293, 331, 383
- Taxis, 286
- Taxis Design Languages, 286
- Taxis semantic data model, 250, 261
- TaxisDL, 262, 286
- TaxisDL design language, 334
- TDL, 262
- TDL, see Taxis, 286
- technically-oriented attributes (software quality criteria), 2, 157
- technology transfer, 396
- telecommunications study, 396
- Telos conceptual modelling language, 382
- Telos knowledge representation language, 277
- template, claim, 120
- templates for refinements, 395
- temporal integrity constraint, 339
- temporal integrity constraints, 336
- throughput, see also time performance, 221
- time for managing long-term processes and integrity constraints (Mgmt-Time, 335
- time performance, 221
- time softgoal for managing long-term processes, 333
- time softgoals, refined from management time softgoals, 276
- time subtypes method, 226
- time, management, 258
- time-ordered list, 340
- time-space decomposition method, 225
- time-space tradeoff, 305, 320, 346, 350
- time-space tradeoffs, 328
- timely accuracy, 166, 324
- tool support, 393
- top-level (main) requirements, determination of, 386
- topic, 20, 21
- topic decomposition methods, 105
- topic decomposition methods for accuracy, 172
- topic decompositions for performance softgoals, 227
- topic of softgoal, 49, 105
- topic parameterization of methods, 94
- topic relation condition, 208
- topic, softgoal, 372
- topics, compared to standard parameters, 49
- total quality management (TQM), 150
- tradeoff, 60
- tradeoffs, 33, 224, 251, 358, 369
- tradeoffs (correlations), 386
- tradeoffs (interactions), 391
- tradeoffs and prioritization, 328
- tradeoffs resolved via domain information, 362
- tradeoffs, see also correlation, 130
- tradoff, see correlation and also implicit interdependency, 18
- training costs and payoff of the NFR Framework, 385
- training issues and the NFR Framework, 387
- transaction calls and IsA hierarchies, 272
- transaction hierarchies, 272, 314
- transaction layer, 222, 261
- transaction, priority, 306
- transition components method, 276, 278, 337
- transition of a script, 334
- transitions of a script, 257, 336
- trigger mechanism, 274
- triggering, 275
- triggering events, 94
- TrivialMany, see also prioritization method, 124
- truth maintenance systems (TMSs), 87
- tuple manipulation, methods, 271
- tuple storage, methods and correlations, 270
- type (of softgoals), see also NFR type, 20, 21
- type catalogue, see also NFR type catalogue, 51, 99
- type catalogue, see NFR type catalogue, 18
- type decomposition methods, 99
- type decomposition methods for accuracy, 167
- type decompositions for performance requirements, 225
- type of an operationalization, 113
- type of NFR, see NFR type, 49
- type of softgoal, 49
- type parameterization of methods, 94
- type, performance, 258
- type, softgoal, 372
- types, accuracy, 165
- uncompressed format, 233, 234
- underdetermined label (U or blank), 71
- unit flow, 165

- UNKNOWN contribution ("?"), 63
upward inference, 134
users' procedure manual, 192, 212
using the NFR Framework, 298
utility, 199
- validation, 175, 177
validation method, 178
validation resource availability method, 178
validity, 195
value accuracy, 166
value removal, 206
value, see also label, 40
variety of information systems, NFRs and domains studied, 292
verification, 177, 208
vertical splitting, 203, 269, 271
vital few, 347
vital few trivial many prioritization template, 180
vital few, trivial many, 207
VitalFewTrivialMany prioritization method, 123
voice recognition, 205
vulnerabilities in relationships, 368
vulnerability in relationships, vulnerable, 371
weak (lighter) reasoning, 8
weak negative label (W^-), 71
weak positive label (W^+), 71
why questions, motivations and rationales, 368, 378
workload, 18, 52, 223
workload, organizational, 257, 293
workload, see also organizational workload, 120
world model within functional requirements, 167
Year 2000 compliance, 396

About the Authors

Dr. Lawrence Chung received his Ph.D. in Computer Science in 1993 from the University of Toronto, where he had previously received the B.Sc. and M.Sc. degrees. He is currently an Assistant Professor of Computer Science at the University of Texas at Dallas. His research interests include software engineering, requirements engineering, non-functional requirements, information systems engineering and re-engineering, and software architectures.

Dr. Brian A. Nixon received his Ph.D. in Computer Science in 1997 from the University of Toronto, where he edited this book as a post-doctoral fellow, and had previously received the B.Sc. and M.Sc. degrees. His Ph.D. thesis deals with performance requirements for information systems. His research interests include information system development (requirements, design, implementation, performance), requirements engineering, software quality, semantic data models, and applications of artificial intelligence to databases, software engineering and programming languages.

Dr. Eric Yu is an Assistant Professor at the Faculty of Information Studies at the University of Toronto. He received his Ph.D. from the Department of Computer Science, University of Toronto in 1995. Before his Ph.D. studies, he was a Member of Scientific Staff at Bell-Northern Research (Nortel Networks) in Ottawa. His research interests include information systems, requirements engineering, software architecture, the modelling of organizations and business processes, and knowledge management.

Dr. John Mylopoulos is professor in the department of Computer Science of the University of Toronto. He holds a Ph.D. from Princeton University, was a senior fellow of the Canadian Institute for Advanced Research (CIAR), is currently an AAAI fellow, and is serving as president of the Very Large Databases Endowment. Mylopoulos received the first ever outstanding services award given out by the Canadian Society for Computational Studies of Intelligence, and was the 1994 co-recipient of the most influential paper award of the International Conference on Software Engineering. His research interests include conceptual modelling, software repositories, requirements engineering and knowledge management.