Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9401730

MegSDF: Mega-Systems development framework

Zemel, Tamar, Ph.D.

New Jersey Institute of Technology, 1993

Copyright ©1993 by Zemel, Tamar. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

MegSDF
MEGA-SYSTEMS DEVELOPMENT FRAMEWORK

by
Tamar Zemel

**A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy**

Department of Computer and Information Science

May 1993

ABSTRACT

MegSDF
Mega-Systems Development Framework

by
Tamar Zemel

A framework for developing large, complex software systems, called Mega-Systems, is specified. The framework incorporates engineering, managerial, and technological aspects of development, concentrating on an engineering process. MegSDF proposes developing Mega-Systems as open distributed systems, pre-planned to be integrated with other systems, and designed for change.

At the management level, MegSDF divides the development of a Mega-System into multiple coordinated projects, distinguishing between a meta-management for the whole development effort, responsible for long-term, global objectives, and local managements for the smaller projects, responsible for local, temporary objectives.

At the engineering level, MegSDF defines a process model which specifies the tasks required for developing Mega-Systems, including their deliverables and interrelationships. The engineering process emphasizes the coordination required to develop the constituent systems. The process is active for the life time of the Mega-System and compatible with different approaches for performing its tasks.

The engineering process consists of System, Mega-System, Mega-System Synthesis, and Meta-Management tasks. System tasks develop constituent systems. Mega-Systems tasks provide a means for engineering coordination, including Domain Analysis, Mega-System Architecture Design, and Infrastructure Acquisition tasks. Mega-System

Synthesis tasks assemble Mega-Systems from the constituent systems. The Meta-Management task plans and controls the entire process.

The domain analysis task provides a general, comprehensive, non-constructive domain model, which is used as a common basis for understanding the domain. MegSDF builds the domain model by integrating multiple significant perceptions of the domain. It recommends using a domain modeling schema to facilitate modeling and integrating the multiple perceptions.

The Mega-System architecture design task specifies a conceptual architecture and an application architecture. The conceptual architecture specifies common design and implementation concepts and is defined using multiple views. The application architecture maps the domain model into an implementation and defines the overall structure of the Mega-System, its boundaries, components, and interfaces.

The infrastructure acquisition task addresses the technological aspects of development. It is responsible for choosing, developing or purchasing, validating, and supporting an infrastructure. The infrastructure integrates the enabling technologies into a unified platform which is used as a common solution for handling technologies. The infrastructure facilitates portability of systems and incorporation of new technologies. It is implemented as a set of services, divided into separate service groups which correspond to the views identified in the conceptual architecture.

Copyright © 1993 by Tamar Zemel
ALL RIGHTS RESERVED

APPROVAL PAGE

MegSDF
Mega-Systems Development Framework

Tamar Zemel

____

Dr. Wilhelm Rossak, Dissertation Advisor — Date
Assistant Professor of Computer and Information Science, NJIT

____

Dr. James A. M. McHugh, Committee Member — Date
Professor and Associate Chairperson of Computer and Information Science, NJIT

____

Dr. Roland T. Mittermeir, Committee Member — Date
o.Univ.Prof.Dip.Ing.Dr. of Computer and Information Science, Universitaet Klagenfurt, Austria

____

Dr. Peter A. Ng, Committee Member — Date
Chairperson and Professor of Computer and Information Science, NJIT

____

Dr. Lonnie R. Welch, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Tamar Zemel

Degree: Doctor of Philosophy in Computer Science

Date: May 1993

Undergraduate and Graduate Education

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1993

- Master of Science in Computer Science,
  Technion - Israel Institute of Technology (IIT), Haifa, Israel, 1982

- Bachelor of Science in Mathematics,
  Technion - Israel Institute of Technology (IIT), Haifa, Israel, 1977

Professional Background

- 1981-1990 RAFAEL Armament Development Authority, Haifa, Israel
  1987-1990 Real-Time Software Department
  Senior Software Engineer for embedded real-time systems. Designed and implemented software systems using advanced software engineering methods, MIL-STD-2167 and related methodologies were introduced to the real-time software department during these projects.
  1981-1987 Management Information Systems Department
  Senior Software Engineer for Management Information Systems. Developed software systems using fourth generation programming languages for human resource management and project control systems.

- 1980-1981 Israel Electric Company, Haifa, Israel
  Software Engineer, developed mechanical engineering management information systems.

Presentations and Publications

- Zemel, Tamar and Rossak, Wilhelm, "A Framework for the Development of Complex Computer Based Systems as Mega-Systems," Workshop on Computer Based Systems Engineering, London, England, December, 1992.

- Zemel, Tamar, Rossak, Wilhelm, and Thimm, Heiko, "Domain Analysis as a Major Component of Integrated Systems Development," in Proceedings of SERF '92, 1992 Software Engineering Research Forum, Indialantic FL, USA, November 1992, pp. 217-224.

- Zemel, Tamar and Rossak, Wilhelm, "Mega-Systems - The Issue of Advanced Systems Development," in Proceedings of IEEE Second International Conference on Systems Integration, Morristown NJ, USA, June 1992, pp. 548-555.

- Zemel, Tamar, "The Organization Subsystem: A Prime Component in Human Resource MIS in RAFAEL," in Proceedings of the Israeli Association for Computerized Systems of Human Resource Management, Second Annual Meeting, Tel-Aviv, Israel, 1986.

- Rossak, Wilhelm, Zemel, Tamar, and Lawson, Harold W., "A Meta-Process Model for the Planned Development of Integrated Systems," International Journal of Systems Integration, Kluwer Academic Publ., Dordrecht, Holland, 1993, to appear.

- Rossak, Wilhelm and Zemel, Tamar, "Engineering Large and Complex Systems with Integration Architectures," PD-Vol. 49 - Computer Applications and Design Abstractions, ETCE '93, Houston TX, USA, February 1993, pp. 189-195.

- Rossak, Wilhelm, Welch, Lonnie, Zemel, Tamar, and Eder, Johann, "A Generic Systems Integration Framework for Large and Time-Critical Systems," in Halang W., Stoyenko A. (eds.): NATO Advanced Study Institute (NATO ASI 910698) in Real-Time Computing, Mullet Bay, Saint Maarten, Springer Verlag, October 1992, to appear.

- Rossak, Wilhelm, Zemel, Tamar, and Ng, Peter A., "Systems Integration - A Framework," Tutorial on Systems Integration for the IEEE Second International Conference on Systems Integration, Morristown NJ, USA, June 1992.

- More than ten internal reports of RAFAEL (confidential)

This Dissertation is dedicated to

my husband Michael,

our children Meir, Yael, and Liat Zemel, and

my parents Zipporah and Israel Bregman

whose love, care, and support

made my work possible

ACKNOWLEDGMENT

The author wishes to express her sincere gratitude to her supervisor, Professor Wilhelm Rossak, for his guidance, friendship, and moral support throughout this research.

Special thanks to the other members of committee: Professor James McHugh for contributing his time and valuable insights, to Professor Roland Mittermeir for his challenging ideas, to Professor Peter A. Ng for his encouragement in all stages of my studies, and to Professor Lonnie Welch for his helpful comments and support.

The author is grateful to the State of New Jersey and Bellcore - Bell Communication Research, for funding of those projects in the Systems Integration Laboratory having direct impact on this dissertation.

The author appreciates the timely help and suggestions form John Mills from Bellcore, the members of the System Integration laboratory: Babita Masand, Heiko Thimm, Vassilka Kirova, Leon Jololian, Yaling Czhou, Amar Mahidadia, Selma Aganovic, and Harriet Chinque, and Dr. Fortune Mhlanga.

Thanks to Bella Goldbereg for providing timely support on insurance questions, and to Eugene Weigner, Carole Poth, and Linda Kowalski for providing timely and expert assistance on editorial questions.

And finally thanks to family and friends for their support and encouragement along the way.

TABLE OF CONTENTS

Chapter **Page**

1 PROBLEMS IN DEVELOPMENT OF CURRENT SOFTWARE SYSTEMS ... 1
2 MEGA-SYSTEMS ... 23
3 A FRAMEWORK FOR MEGA-SYSTEM DEVELOPMENT ... 34
4 MegSDF PROCESS MODEL ... 53
5 DOMAIN ANALYSIS FOR MEGA-SYSTEMS ... 65
6 MEGA-SYSTEM ARCHITECTURE DESIGN ... 126
7 INFRASTRUCTURE ACQUISITION IN MegSDF ... 221
8 THE META-MANAGEMENT, SYSTEM, AND MEGA-SYSTEM SYNTHESIS TASKS ... 254
9 A SCENARIO ... 272
10 CONCLUSIONS AND SUMMARY ... 285
BIBLIOGRAPHY ... 305
GLOSSARY ... 320

TABLE OF CONTENTS

Chapter **Page**

1 PROBLEMS IN DEVELOPMENT OF CURRENT SOFTWARE SYSTEMS ... 1
1.1 The Evolution of Software Systems ... 2
1.2 Analysis of Problems in the Development of Software Systems ... 4
1.2.1 Characteristics of Large and Complex Software Systems ... 4
1.2.2 Aspects of Software Development ... 10
1.2.3 The Impact of Software System Characteristics on Aspects of Software Development ... 14
1.2.4 Summary of the Problems ... 21
2 MEGA-SYSTEMS ... 23
2.1 Huge Systems ... 24
2.2 Systems of Systems ... 26
2.3 Generic Systems ... 28
2.4 Generic Systems of Systems ... 30
2.5 Relationships among Mega-Systems ... 31
3 A FRAMEWORK FOR MEGA-SYSTEM DEVELOPMENT ... 34
3.1 Existing Models and Frameworks ... 34
3.1.1 The COSMOS Model ... 35
3.1.2 The GenSIF Framework ... 35
3.1.3 The POWDER Methodology ... 36
3.1.4 The SIF Framework ... 37
3.1.5 System of Systems Engineering Model ... 38
3.1.6 The Megaprogramming Framework ... 39
3.1.7 Summary of Existing Methods ... 40
3.2 Requirements for a Framework ... 42
3.3 Outline of MegSDF Framework ... 43
3.3.1 Development Organization ... 44
3.3.2 Engineering Coordination ... 47
3.3.3 The Pre-Planned Approach ... 48
3.3.4 Development as Open Distributed System ... 51
4 MegSDF PROCESS MODEL ... 53
4.1 A Method to Describe an Engineering Process ... 53
4.2 Mega-System Development Process Model ... 56
4.2.1 Purpose ... 56
4.2.2 Interfaces ... 56
4.2.3 Processing ... 57
4.2.4 Timing ... 58
4.2.5 Sub-Tasks ... 59
4.3 Mega-System Task ... 60
4.3.1 Purpose ... 60
4.3.2 Interfaces ... 61

TABLE OF CONTENTS (CONTINUED)

Chapter | Page

4.3.3 Processing ... 62
4.3.4 Timing ... 63
4.3.5 Sub-Tasks ... 64
5 DOMAIN ANALYSIS FOR MEGA-SYSTEMS ... 65
5.1 Requirements for Domain Analysis ... 66
5.1.1 The Role of Domain Analysis ... 66
5.1.2 Requirements for MegSDF's Domain Model ... 68
5.1.3 Contrast with Domain Analysis in Reusability and System Analysis ... 71
5.2 Domain Modeling ... 72
5.2.1 The Content of the Model ... 73
5.2.2 Structuring the Model ... 77
5.2.3 Definitions of Domain-Analysis Concepts ... 92
5.3 The Domain Analysis Process ... 96
5.3.1 Purpose ... 96
5.3.2 Interfaces ... 97
5.3.3 Processing ... 97
5.3.4 Timing ... 98
5.3.5 Sub-Tasks ... 99
5.4 Comparison with Existing Methods ... 103
5.4.1 Comparison with System Analysis Approaches ... 104
5.4.2 Comparison with other Domain Analysis Approaches ... 107
5.5 An Example for Domain Analysis in the Insurance Domain ... 110
5.5.1 The Static Dimension ... 111
5.5.2 The Functional Dimension ... 119
6 MEGA-SYSTEM ARCHITECTURE DESIGN ... 126
6.1 Requirements for Mega-System Architecture Design ... 127
6.1.1 The Role of Mega-System Architectures ... 127
6.1.1 Requirements for Mega-System Architectures ... 133
6.2 The Mega-System Architecture ... 136
6.2.1 Parts of the Mega-System Architecture ... 136
6.2.2 The Conceptual Architecture ... 137
6.2.3 Application Architectures ... 156
6.3 Mega-System Architecture Design Process ... 159
6.3.1 Purpose ... 159
6.3.2 Interfaces ... 160
6.3.3 Processing ... 161
6.3.4 Timing ... 161
6.3.5 Sub-Tasks ... 162
6.4 Existing Architectures ... 168

TABLE OF CONTENTS (CONTINUED)

Chapter Page

6.4.1 Systems Architectures 168
6.4.1.1 Application Machine 168
6.4.1.2 Best's Architecture 173
6.4.2 Mega-System Architectures 178
6.4.2.1 The OSCA Architecture 178
6.4.2.2 A Network of Application Machines 187
6.4.2.3 The CAN-Kingdom 191
6.4.2.4 The Advanced Networked Systems Architecture (ANSA) 196
6.4.3 Examples of Projects with Software Architectures 201
6.4.3.1 Ship-2000 201
6.4.3.2 ESF - FSE Reference architecture 209
6.4.4 Classification and Comparison of Existing Architectures 217
7 INFRASTRUCTURE ACQUISITION IN MegSDF 221
7.1 Requirements for Infrastructure Acquisition 222
7.1.1 The Role of Infrastructure Acquisition 222
7.1.2 Requirements for an Infrastructure 226
7.2 An Infrastructure 228
7.2.1 MegSDF Infrastructure 228
7.2.2 An Infrastructure Model 231
7.3 The Infrastructure Acquisition Process 236
7.3.1 Purpose 237
7.3.2 Interfaces 237
7.3.3 Processing 238
7.3.4 Timing 238
7.4 Examples of Existing Infrastructures 239
7.4.1 The NIST Reference Model 240
7.4.2 ANSAware 244
7.4.3 IBM's Systems Application Architecture (SAA) 249
8 THE META-MANAGEMENT, SYSTEM, AND MEGA-SYSTEM SYNTHESIS TASKS 254
8.1 The Meta-Management Task 254
8.1.1 The Role of the Meta-Management Task 254
8.1.2 The Process of the Meta-Management Task 258
8.2 The System Tasks 260
8.2.1 The Role of the System Tasks 260
8.2.2. The System Development Process 262
8.3 Mega-System Synthesis in MegSDF 266
8.3.1 The Role of Mega-System Synthesis 266
8.3.2 The Process of Mega-System Synthesis 268

TABLE OF CONTENTS (CONTINUED)

Chapter **Page**

9 A SCENARIO 272
9.1 Current Status 272
9.2 A Solution Based on MegSDF 277
9.3 Advantages of Using MegSDF 279
10 CONCLUSIONS AND SUMMARY 285
10.1 Requirements Verification 285
10.1.1 Realization of Framework Requirements 285
10.1.2 Quality Attribute Map 286
10.2 Prerequisites for Success 291
10.3 Summary 292
APPENDIX A MegSDF PROCESS DIAGRAMS 295
A.1 MegSDF First Level 295
A.2 Mega-System Task 296
A.3 Domain Analysis 297
A.4 Mega-System Architecture Design 298
A.5 Conceptual Architecture Design 299
A.6 Application Architecture Design 300
A.7 Infrastructure Acquisition 301
A.8 Meta-Management 302
A.9 System Task 303
A.10 Mega-System Synthesis 304
BIBLIOGRAPHY 305
GLOSSARY 320

LIST OF TABLES

Table		Page
1.1	Problems Faced in Development of Large and Complex Systems	22
2.1	A Comparison of Mega-Systems	33
3.1	Existing Models	41
5.1	Difficulties and Problems Addressed by MegSDF Domain Analysis	67
5.2	A Template for Element-Types	81
5.3	An Example of an Object-Element-type	83
5.4	An Example of a Building-Object-Element	85
5.5	Object-Perception-Element-Type	87
5.6	Building-Perception-Element	87
5.7	A Comparison of MegSDF Approach with Modeling Approaches	106
5.8	A Comparison of MegSDF with other Domain Analysis Approaches	109
5.9	Mapping Perceptions' Objects to Domain Model Objects	116
5.10	Mapping Perceptions' Relationship	118
5.11	Mapping Perceptions' Processes to Domain Model Processes	123
5.12	Mapping Flows, Sources, Terminators, and Data Stores	124
6.1	Difficulties and Problems Addressed by the Mega-System Architecture	130
6.2	An Outline for a Conceptual Architecture	155
6.3	An Outline for an Application Architecture	159
6.4	Application Machine Structural View Mapping	170
6.5	Application Machine Communication View Mapping	170
6.6	Application Machine Control View Mapping	171
6.7	Application Machine Environment View Mapping	172
6.8	Application Machine Application Architecture Mapping	172
6.9	Best's Architecture Structural View Mapping	175
6.10	Best's Architecture Communication View Mapping	175
6.11	Best's Architecture Control View Mapping	176
6.12	Best's Architecture Data View Mapping	177
6.13	Best's Architecture Environment View Mapping	177
6.14	Best's Architecture Application Architecture Mapping	178
6.15	The OSCA Architecture Structural View Mapping	181
6.16	The OSCA Architecture Communication View Mapping	182
6.17	The OSCA Architecture Control View Mapping	183
6.18	The OSCA Architecture Data View Mapping	184
6.19	The OSCA Architecture Environment View Mapping	186
6.20	Network of AM Structural View Mapping	188
6.21	Network of AM Communication View Mapping	188
6.22	Network of AM Control View Mapping	189
6.23	Network of AM Data View Mapping	190
6.24	Network of AM Environment View Mapping	190

LIST OF TABLES (CONTINUED)

Table **Page**

6.25 Network of AM Application Architecture Mapping 191
6.26 CAN-Kingdom Structural View Mapping 192
6.27 CAN-Kingdom Communication View Mapping 193
6.28 CAN-Kingdom Control View Mapping 194
6.29 CAN-Kingdom Application Architecture Mapping 195
6.30 ANSA Structural View Mapping 198
6.31 ANSA Communication View Mapping 199
6.32 ANSA Control View Mapping 200
6.33 ANSA Data View Mapping 200
6.34 Ship-2000 Structural View Mapping 203
6.35 Ship-2000 Communication View Mapping 205
6.36 Ship-2000 Control View Mapping 206
6.37 Ship-2000 Data View Mapping 207
6.38 Ship-2000 Environment View Mapping 208
6.39 Ship-2000 Application Architecture Mapping 209
6.40 ESF Structural View Mapping 213
6.41 ESF Communication View Mapping 214
6.42 ESF Control View Mapping 215
6.43 ESF Data View Mapping 215
6.44 ESF Environment View Mapping 216
6.45 Systems Architectures 218
6.46 Mega-System Architectures 219
6.47 Projects that Use Architectures 220
7.1 Difficulties and Problems Addressed by the Infrastructure 223
7.2 Mapping of the Structural View Concepts into Services 232
7.3 Mapping of the Communication View Concepts into Services 233
7.4 Mapping of the Control View Concepts into Services 234
7.5 Mapping of the Data View Concepts into Services 235
7.6 Mapping of the Environment View Concepts to Services 236
7.7 Comparison of MegSDF Views and NIST Service Groups 243
7.8 Comparison of MegSDF views and ANSAware services. 248
7.9 A Comparison of MegSDF views and SAA Service Groups 253
8.1 Difficulties and Problems Addressed by the Meta-Management Task 255
10.1 Impacts of MegSDF's Tasks on Quality 290

LIST OF FIGURES

Figure | Page

1.1 Post-Facto Integration 3
1.2 Aspects Involved in Development of Software Systems 11
2.1 Mega-Systems 24
2.2 A Huge System 25
2.3 A System of Systems 27
2.4 A Generic System 29
2.5 A Generic System of Systems 31
2.6 Detailed Classification of Mega-System 32
3.1 The POWDER Model 37
3.2 Mega-System Development Organization 45
3.3 Mega-System 52
4.1 The Graphical Notation for a Traditional SADT Activity 54
4.2 Graphical Notations for MegSDF Activities 55
4.3 MegSDF First Level 58
4.4 A Schedule for Mega-System Development 59
4.5 Process Diagram for the Mega-System Tasks 63
5.1 The Relationship of the Domain Model 68
5.2 A Domain Model as an Integration of Multiple Perceptions 76
5.3 The Integration of Perception-Elements into Elements 89
5.4 The Components of Domain Analysis 91
5.5 A Process Diagram for Domain Analysis 99
5.6 The Static Dimension of the Insurer Perception 112
5.7 The Static Dimension of the Insured Perception 113
5.8 The Static Dimension of the Agent Perception 114
5.9 The Integrated Static Dimension 119
5.10 The Functional Dimension of the Insurer 120
5.11 The Functional Dimension of the Insured Perception 121
5.12 The Agent Perception Functional Dimension 122
5.13 The Integrated Functional Dimension 125
6.1 The Role of the Mega-System Architecture 132
6.2 Architectural Style, Conceptual and Application Architectures 138
6.3 Relationship of the Application Architecture to Other MegSDF Elements . . 157
6.4 Mega-System Architecture Design Process 162
6.5 Conceptual Architecture Design Process 165
6.6 Application Architecture Design 167
6.7 Best's Architecture Process Flow 174
6.8 The OSCA Architecture 182
6.9 The Structural View of the ESF Architecture 211
7.1 Relation of Infrastructure to other MegSDF Components 225
7.2 The Infrastructure Acquisition Process 239

LIST OF FIGURES (CONTINUED)

Figure **Page**

7.3 The NIST Reference Model 242
7.4 ANSAware's Capsule and Nucleus 246
7.5 The Elements of IBM's SAA 252
8.1 Meta-Management Process Diagram 259
8.2 Relationship of System Tasks to other Elements of MegSDF 261
8.3 System Process Diagram 264
8.4 Mega-System Synthesis Process Diagram 270
A.1 MegSDF First Level Process Diagram 295
A.2 Mega-System Tasks Process Diagram 296
A.3 Domain Analysis Process Diagram 297
A.4 Mega-System Architecture Process Diagram 298
A.5 Conceptual Architecture Design Process Diagram 299
A.6 Application Architecture Design Process Diagram 300
A.7 Infrastructure Acquisition Process Diagram 301
A.8 Meta-Management Task Process Diagram 302
A.9 System Task Process Diagram 303
A.10 Mega-System Synthesis Process Diagram 304

CHAPTER 1

PROBLEMS IN DEVELOPMENT OF CURRENT SOFTWARE SYSTEMS

The current state of software development has been described as a crisis [BROO 87], [PRES 92] symptoms of which include customers dissatisfied with the quality of the systems they acquire, developers who underestimate the efforts required for developing software systems, and demands for new systems and capabilities far in excess of the ability of software engineers to provide them. Some of the roots of this crisis are to be found in changes in the characteristics of software systems over the past few decades. Consequently, the solutions to these problems must consider the impacts of these changing characteristics on the various aspects of software development.

The following sections describe the recent evolution of software systems, the characteristics of current software systems, the various aspects of software development, and the impacts of current systems' characteristics on these development aspects. Based on this discussion, we will subsequently propose a framework for the development of large, complex software systems.

1.1 The Evolution of Software Systems

Programs, and later on software systems that included several programs, were originally developed to solve specific problems for specific users or well-defined groups of users. These systems operated in homogeneous environments. The traditional software-engineering approaches were successfully used to develop this kind of system.

Subsequent reductions in hardware prices, advances in technology, and the maturity of customers and developers have led to the development of systems of markedly increased size and complexity [MAYE 89], [CSTB 90], [MOOR 92]. More recently, users have had to rely on multiple independent systems to solve sets of related problems. These systems often run on different platforms or in heterogeneous environments. Users have realized, however, that it is inefficient to use such multiple systems. Instead, they have come to expect integrated solutions which may even yield additional values that cannot be achieved by independent solutions.

Two approaches are currently used to meet the demand for integrated solutions. The first approach is called post-facto integration [POWE 90]. This approach integrates several systems using ad-hoc, non-systematic methods (Figure 1.1). The systems to be integrated were developed to solve specific problems in the domain each with a limited perception of the domain, without an awareness of future integration requirements, and with no relation between the systems. The addition of a new system, replacement of an existing system, or incorporation of new technology requires extensive effort. The approach is called post-facto because the integration is designed and performed after the

development of the constituent systems has been completed. The second approach is to develop a huge system [YOUR 92]. In this approach, a large, complex, interrelated system is developed. The various components of the system are tightly coupled and consequently their maintenance is horrendously difficult.

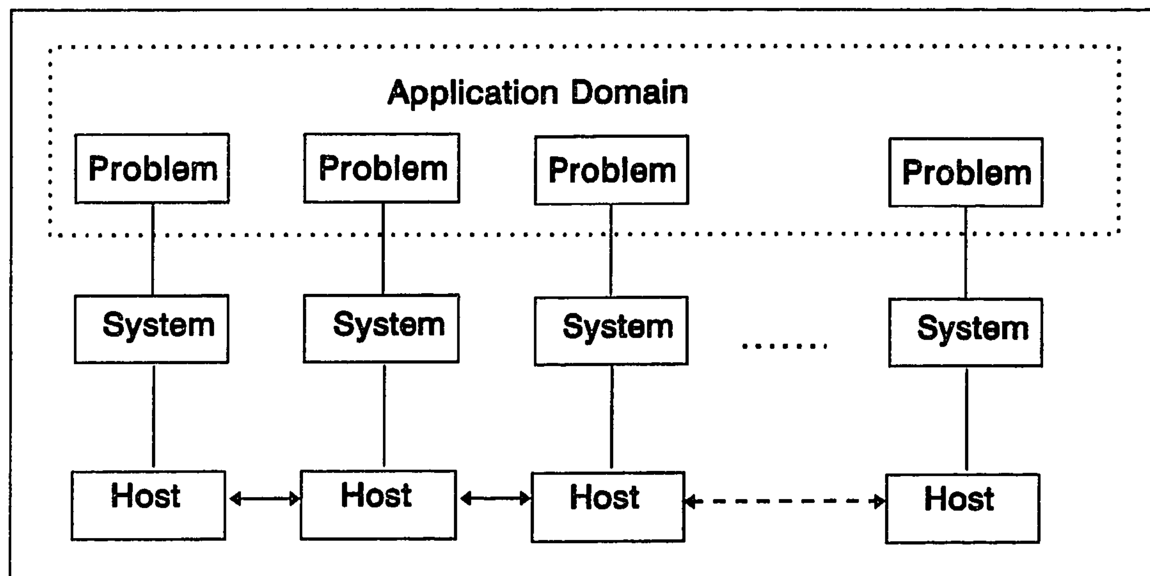

Figure 1.1 Post-Facto Integration

These approaches may both be considered as technology-driven since they use new technologies in an uncoordinated manner, without the adoption of improved and suitable engineering and management models. They are essentially "bottom-up" approaches, based on traditional development techniques appropriate to smaller problems, single systems, specific platforms, and shorter life cycles. They do not prepare systems for future integration and so entail drastic efforts when integration is required [POWE 90]. Moreover, they concentrate primarily on the engineering aspects of how to develop

systems. They do not adequately address the difficulties that exist in the development of the newer systems.

The evolution of software systems suggests that a new approach to software development is required. However, before describing such an approach, it is imperative to first understand the impact current systems characteristics have on the development process.

1.2 Analysis of Problems in the Development of Software Systems

The following sections describe, more precisely, the characteristics of typical current software systems, and the impacts of these characteristics on software systems development.

1.2.1 Characteristics of Large and Complex Software Systems

Large, complex systems used in various domains generally have more than one of the following characteristics [EISN 91], [MITT 91]:

- Consist of more than one system
- Developed by more than one group of developers
- Have more than one customer/user
- Operate in a heterogeneous environment
- Have a long life cycle

These characteristics are interdependent and interrelated. Often one characteristic implies the presence of others. For example, the presence of the characteristic "consist of more than one system" generally implies that these systems are "developed by more than one group of developers" and "operate in a heterogeneous environment". The following paragraphs describe these characteristics in more detail.

Consist of More Than One System

Most software development efforts involve "more than one system". These systems often integrate a number of smaller systems, which had been developed independently, into a larger system. Such integration is primarily a response to customer requirements. Customers insist on the integration of currently independent systems into larger systems in order to obtain additional values that cannot be attainable otherwise [CLAR 92].

On the other hand, integration is sometimes the initiative of the developers. Software developers tend to cooperate to enlarge their market share. For example, the developers of Lotus 1-2-3, a spreadsheet software, decided to cooperate with the developers of Ami Pro, a word-processing software, and the developer of another software product cc:Mail to provide their customers a "software suite" [MOSE 92]. The developers believed that customers preferred comprehensive, integrated solutions rather than merely a set of independent tools.

Another case where "more than one system" occurs is the program family [PARN 76]. A program family is a group of systems with similar functionalities. Each system in the family has a specific configuration and is developed for a different customer.

Configurations may differ in the set of functionalities, the technical environment in which the systems operate, or the interfaces of the systems.

Developed by More Than One Group of Developers

Systems that are integrated as just described have, typically, been developed by "more than one group of developers", at different points in time, and with diverse schedules. The size and complexity of such systems often lead to their development as the cooperative effort of multiple groups of developers. An extreme example of cooperative development is the space-station Freedom [MOOR 92], however the same phenomenon occurs with smaller systems too.

Development with more than one group of developers may reduce some aspects of risk (although this way of development might increase communication problems and there by increase other aspects of risk). For example, it may be possible to buy parts that were already developed by other groups of developers. It is also reasonable to assign special tasks to specialized groups, thereby gaining from their experience. Moreover, when there is uncertainty regarding the feasibility of a system, it may be possible to assign the same system to different groups. In this case the various groups concurrently develop different solutions based on different technologies and approaches; this increases the chance of obtaining an effective solution.

Finally, when several groups develop different parts of the system, the dependence of the customer on the developers is thereby reduced. Each group develops only a limited part of the system and so can be replaced with limited, local effect. Furthermore, when

several groups develop the various parts of a system in parallel, the duration of the development is usually shorter.

Have More Than One Customer/User

Many software systems are developed to support a large group of users or customers. The user groups are often themselves heterogeneous, consisting of diverse users, each with his own particular role and his own requirements for the system. Consequently, such systems have an immense variety of interrelated functions. These systems additionally tend to support their user groups as a whole by providing a means of communication among the members of the group. CASE tools, for example, support software development teams consisting of system analysts, designers, and project managers. These systems support the work of each member of the group, but also allow the transfer of information between members.

The high cost of software development makes it infeasible to develop "tailor-made" systems for every customer. It is more reasonable to sell a system with modifications to several customers, thus sharing the cost of development efforts. In such situations, software systems are developed as program families [PARN 76] or as parts of product lines.

Another instance of "more than one customer" is when a system is initially developed as an in-house solution to some company need, but is subsequently sold as a product to other companies, becoming a source of income for the original company. Copies of the system can be reproduced with minimal expense, and sold to various customers. Another situation where "more than one customer" occurs is in the context of

increasing the effectiveness of a system. For example, cooperation among several banks, which are customers of a system of Automatic Taller Machines (ATM), might increase the number of installed ATMs, thereby improving the regional coverage of the ATMs, consequently enhancing the convenience of the banks' clients. This improvement might increase the number of clients at all banks [CLEM 91].

The presence of a large number of customers also characterizes package systems. Examples of packages are word-processors, e.g., Word-Perfect [SALK 91] or spreadsheets, e.g., Lotus [GRIF 91]. These systems are developed for common, general usage. Given the enormous number of "unknown" users, it is impossible to have a specific configuration defined for each user. The heterogeneity of the user groups increases the need for flexibility of the system. One way to provide this flexibility is by allowing users to customize and adjust their systems according to their own preferences whenever possible.

It is important to note that the heterogeneous group of users developers face today often means dealing with different kinds of user. For example, the system may be developed for a known group of users. On the other hand, it may be developed for unknown users that might be represented by an "opinion center" [MITT 91] or a "virtual user", e.g., management, marketing, or sales personal.

Operate in a Heterogeneous Environment

In the past, most software systems were developed to run on homogeneous environments, i.e., in a specific hardware configuration, one type of operating system, a single programming language, a specific database management system, and a single

communication network. However, with constantly changing technologies there is a variety of environments in which systems might run.

Current systems operate in heterogeneous environments consisting of more than one platform, several operating systems, various databases, and communication tools. This is often a result of the integration of several systems where each system operates in a different environment. Moreover, many systems have been developed to operate in heterogeneous environments as a practical requirement. Thus, the presumption of a homogeneous environment no longer holds.

Sometimes the same software system is developed to operate in different environments in order to increase the market share of its developers. An example of this phenomenon is the Word Perfect word-processing software that operates on UNIX, DOS, and Windows. Another example is the Lotus Suite that operates on Personal Computers under DOS and Windows. This software suite has now been developed to run on HPs under UNIX to enhance its effectiveness as a communication tool [JOHN 92]. This type of heterogeneity allows users in heterogeneous environments (e.g., an environment that consists of several mainframes operating under UNIX and a number of personal computers running under DOS and Windows) to use the same software on any machine.

It is important to point out that there are two kinds of heterogeneity. The first kind refers to systems that span heterogeneous environments, the second kind refers to systems where each variation of the system operates in a single homogeneous environment only.

Have a Long-Life Cycle

The life-cycle of software systems is constantly being lengthened. For example, the Air Traffic Control (ATC) of the Federal Aviation Administration system has been used for twenty years, its successor is designed to operate for twenty to thirty years [HUNT 87]. The complexity and size of software systems induce both high costs and long development periods. Because of economic constraints, customers do not have sufficient resources to acquire new systems, so they are constrained to use the existing systems for longer periods of time. These systems have generally undergone repeated generations of change and may be now virtually unmaintainable [PRES 92]. These systems are often called "legacy systems" [YOUR 92] or "aged systems" [PRES 92] since it is simultaneously excessively difficult to maintain them and is too expensive to redevelop them, so users are forced to keep them.

The process of acquiring the various systems which participate in an integrated solution is continuous and evolutionary. The constituent systems of an integrated system are often developed in non-overlapping time-frames. Thus, once again the development period of the final integrated system is much longer than the development period of traditional systems.

1.2.2 Aspects of Software Development

Software development involves many interrelated aspects, e.g., engineering, managerial, technological, psychological, sociological, economic, legal, and political (Figure 1.2). We

shall discuss each of these aspects in the following paragraphs, but as software engineers, we shall concentrate on the effect of the various aspects on the development process.

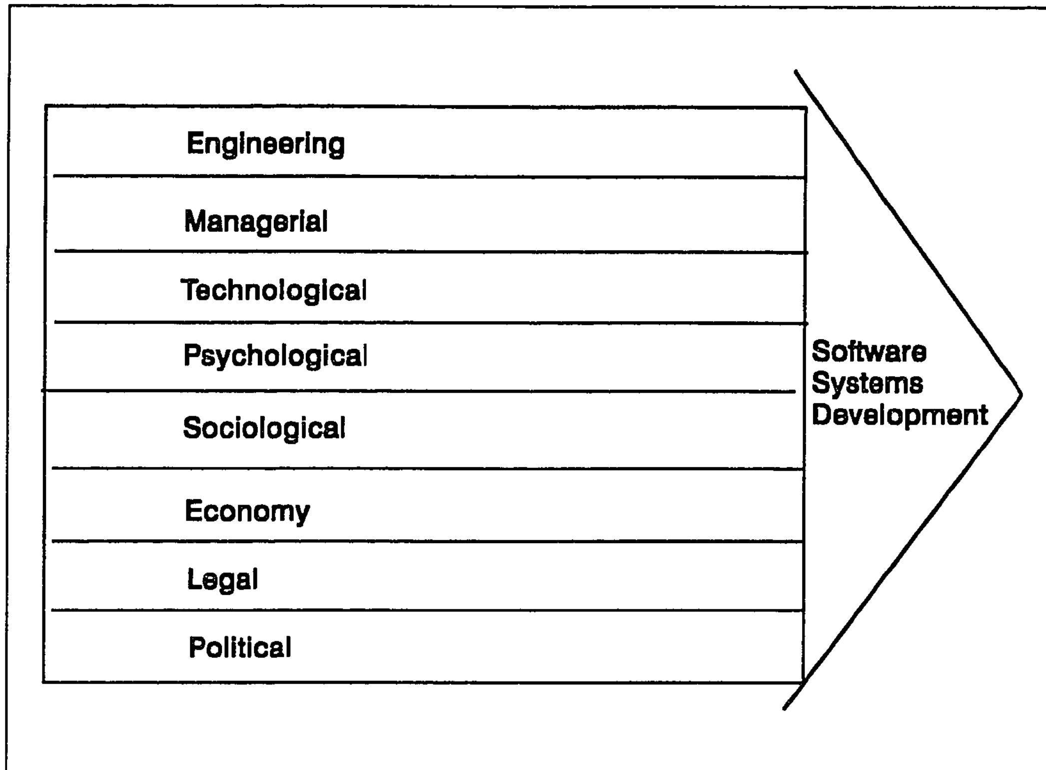

Figure 1.2 Aspects Involved in Development of Software Systems

Engineering aspects include processes, methods, techniques, and tools used to develop software systems. Processes specify the activities required for efficient development of high quality software systems. Techniques support implementation of distinct activities. Tools enable efficient implementation of techniques [GEHD 91].

Managerial aspects include the organization of development groups, software metrics, software (cost) estimating methods, configuration management, quality assurance, and risk analysis. Organization deals with the responsibilities, roles, and structure of a development group [BAKE 72], [RETT 90]. Software metrics are the means of evaluating

the quality of software products as well as the productivity of the development process [ARTH 85], [JONE 86]. Software estimating methods are used to estimate the cost and the required resources for development of a system [BOEH 82]. Risk analysis deals with methods for identification, projection assessment, and management of risks during the software development [BOEH 89], [CHAR 89], [PRES 92].

Technological aspects deal with the enabling technologies that support the development and operation of software systems, e.g., hardware, database management systems, and communication. There is an immense variety of technologies developed without agreed-upon standards. Therefore, the implementation of a system that requires the use of various technologies is difficult. Moreover, new technologies are continually emerging and must be incorporated in order to ensure the effectiveness of a system and the competitiveness of its users [CLEM 91]. Our work concentrates on the special requirements that complex and large systems impose on technologies.

Psychological aspects deal with human factors that must be considered in the design of a software system [LUND 91], implementation of methods for solving problems in software development, and the psychological impact of group organization.

Broadly speaking, ***sociological*** aspects address the impact software systems have on society and social transformations that result from computerization [KLIN 90]. In regard to software development point of view, this aspect deals with the effects of the software development process and group organization on developers.

Regarding ***economic*** effects, software systems, especially strategic systems, can have major economic effects on companies that acquire them. Strategic systems may even

be critical to the success or failure of companies [CLEM 91]. Furthermore, cooperation and partnerships among developers may be motivated by economic constraints and have significant economic benefits for the various partners.

Legal aspects address privacy of information, property rights of software developers, and even "equal opportunity" for companies. New laws to ensure the privacy rights of individuals whose data is stored in data repositories, or at least to allow them the option of knowing what information is kept about them, have been legislated in various countries, e.g., [ODSG 78].

Property rights, patents, and copyrights of software developers must be considered in using and developing software [ACKE 92]. For example, the case of Apple against Microsoft Corp. and Hewlett Packard Co. deals with Apple's copyrights for the windowing concepts [DALY 92]. There have also been governmental regulations designed to ensure equal opportunity to access software functionalities to eliminate unfair competitive advantages in strategic systems. For example, in the case of the SABRE on-line reservation system, American Air Lines, which owned the SABRE system, had privileges that other customers of the system did not have [BETT 92].

In the legal/political arena, we find national/international efforts dealing with standardization of software products and their development. Examples for these efforts are the Department of Defense DoD-STD-2167a [DoD_STD-2167a], IEEE standards, the ISO Norm 9000, and international regulations concerning quality and safety [BHAN 93].

The previously mentioned aspects are all important, but our framework concentrates on the engineering, managerial, and technological aspects of software

development. Moreover, we shall consider the managerial and technological aspects only in respect to their role as means for supporting the engineering process.

1.2.3 The Impact of Software System Characteristics on Aspects of Software Development

We contend that the underlying causes of the software "crisis" are rooted in the characteristics described in the previous section. This section describes how problems in software development are engendered by these characteristics. Because systems characteristics and development aspects are interrelated, some problems will be discussed from several viewpoints.

1.2.3.1 Problems in Engineering

The engineering aspects deal mainly with processes, methods, techniques, and tools used to develop software systems. We next describe the engineering aspects and the engineering problems induced by the previously identified characteristics.

More Than One System

The integration of independently developed systems into a coherent larger system is typically extraordinarily difficult [CSTB 90]. Usually, the systems that have to be integrated were developed previously and with no awareness of other systems or future integration requirements, but it has become necessary to integrate them to gain added values. Most such systems for integration might be described as "legacy systems," [YOUR

92] since they are large, inflexible, and old. They are too large to be redeveloped, and yet they are very difficult to change or modify.

More Than One Group of Developers

Often when a system is developed by more than one group of developers, the various groups focus primarily on the development of their own part. They deal with a limited portion of the domain and have only a limited knowledge of the domain [CURT 88]. When a family of systems is developed by more than one group, the groups may become tend to enmeshed in incidental environmental features rather than focusing on the application problem solutions [LAWS 92a]. In either case, a global approach is lacking; the system as a whole is neglected or has less than the requisite priority [NEUM 91].

Another problem occurs when each group of developers uses its own standards, procedures, methods, and tools. This leads to non-uniform integrated systems that have multiple types of user interfaces, different ways of error handling, etc. Their non-uniformity makes them difficult to use and hard to maintain.

More Than One Customer/User

When a system is developed for more than one customer, every customer operates his system in specific circumstances (technical and organizational environment). This implies that the requirements of each customer might differ, inducing an increase in complexity.

Often different customers have different configurations of the system. It is more difficult to develop and maintain a system with various configurations since any change has to be evaluated and occasionally incorporated into the various configurations.

Heterogeneous Environment

A heterogeneous environment implies use of various technologies. Occasionally, additional efforts may be required in order to find engineering solutions to close technological gaps and bridge technologies [NOTK 88]. Similarly, when a system is technology-driven and uses an emerging technology, additional efforts are required to solve immaturity problems, primarily in interfacing with existing technologies.

Long Life Cycle

Generally, a software system is a part of a larger domain. This domain has a major influence on the requirements for the system and evolves independently [LEHM 90]. Often the domain is influenced by the system itself. Changes in the domain may imply changes in the system's requirements. The possibility of significant changes in the domain and, therefore, in the requirements, is increased if the life cycle of the system is long [CSTB 90]. Thus, long life cycles of software systems lead to unstable requirements.

Given the current size and complexity of systems, it seems more rational for systems to evolve, rather than be developed at once [TICH 93]. It may be impossible to replace the whole system at once, but parts can be added, updated, or replaced over time to adjust the system according to new requirements and emerging technologies.

Summary of Engineering Aspects

We contend that the reason for most of the difficulties related to the engineering aspects is the use of unsuitable approaches to solve highly complex problems. Complex systems are currently developed using traditional approaches for software development, e.g., the waterfall [BOEH 76], prototype [GOMA 90], and the spiral model [BOEH 88]. These

methods assume the development of one system with rather stable requirements and are based on a sequential, phased process [MITT 91]. Therefore, they do not fit the development of more than one system with several groups of developers; nor do they suit a large domain with unstable requirements; and finally they do not support long-term development in a dynamic environment.

1.2.3.2 Problems in Management

As the size and complexity of software system is increased, management tasks become more difficult. We have to deal with many developers and for a longer period of development [PRES 92]. Any difficulties in software development, e.g., risk identification and elimination, communication, and coordination problems, are scaled up [CURT 88].

More Than One System

If more than one system is developed we have to deal with two levels of objectives: the overall integrated system's (general) objectives and the local objectives of the various constituent systems. These objectives occasionally contradict each other. There is usually no clear distinction between management aspects of the integrated solution and management aspects of the various participating systems; consequently local aspects tend to swamp or preempt general objectives.

More Than One Group of Developers

Communication and coordination problems that exist in the development of one system developed by a single group are scaled up and become critical when a system is developed by more than one group of developers [CURT 88]. Often, the different groups belong to

different organizations (which occasionally are competitors). These organizations have different (and occasionally contradictory) goals and aims. Generally, the groups work at different sites or even in different countries, e.g., the developers of the space-station Freedom are from USA, Italy, Japan, and other countries [MOOR 92]. Without effective coordination and communication, there will be wasted efforts in developing solutions developed previously by other groups.

Another problem is caused when various groups have different cultures. This leads to different interpretations of the domain and of the system by the various groups. In the case of problems, each group may try to blame the other groups [CURT 88], [YOUR 92].

More Than One Customer/User

The fact that a system has more than one customer induces an increase in complexity. Typically, every customer has his own aims and needs and therefore his own preferences and priorities. These preferences may be different or contradictory. The various requirements must be analyzed and an optimized solution and development schedule determined.

Heterogeneous and Dynamic Environment

A heterogeneous and dynamic environment increases the complexity of systems. Often management deals with standardization of technologies by stabilizing the environment. In this approach, only elements compatible with the standards may be used. In dynamic environments, the management has to ensure that services that were provided previously will still be provided in the future. Moreover, management has to evaluate the various emerging technologies to keep the developed systems efficient and effective.

Long Life Cycle

Often, in the development of large and complex systems, management deals with short-term objectives and neglects long-term objectives [YEH 91]. When the life cycle of a system becomes longer it is essential to emphasize long-term objectives and to derive short term objectives from them.

Summary of the Managerial Aspects

We contend that the difficulties related to managerial aspects arise because there is typically no specific management entity that deals principally with general and long-term objectives in conglomerate projects where the number of participants is very high. Without a distinction between the various objectives, the short term and local objectives overwhelm the global, long term, and more essential problems. There is a need for a management that will manage and coordinate the various groups and determine policy and directions for the whole system.

1.2.3.3 Problems in Technology Handling

In this section we discuss technological aspects of the development of software systems. Technologies enable the implementation of software systems. There is an immense variety of technologies already on the market, and new technologies emerge at such a fast rate that it is hard to deal with them productively.

More Than One System

The integration of several systems may lead to heterogeneous environments. Moreover, the current tendency is to "down-size", i.e., to replace a central system running on a

mainframe by a network of smaller systems running on a set of interconnected, smaller platforms. Thus, we must address distributed processing technologies and ensure the consistency of their operations.

The case of a family of systems, where each system operates in a different environment, requires methods that will improve the portability of the system, i.e., transferring the system to a new environment.

More Than One Developer

Typically, each group of developers is specialized in a specific set of technologies. This fact may lead to a heterogeneous environment. Another difficulty is caused if each developer struggles independently with the problems induced by the heterogeneous and dynamic environment, leading to redundant efforts and a non-uniform system.

More Than One Customer

Generally, various customers have different technological environments. Thus, typically systems with more than one customer have to be adjusted to the various environments.

Heterogeneous and Dynamic Environment

When a system is designed to operate in a heterogeneous environment, we have to deal with a variety of technologies. Currently, the various technologies are developed without agreed-upon standards. To enable operation of systems in heterogeneous environments, there is a need to first "bridge" the technologies [NEFF 92].

Similar problems are caused by the dynamics of the environment. Emerging technologies must be incorporated to ensure the effectiveness of systems.

Long Life Cycle

As the life cycle of systems becomes longer, the effects of emerging technologies may become more critical. It is sometimes not merely optional but essential to incorporate emerging technologies in order to ensure the competitiveness of the systems and their customers [CLEM 91].

The "aging problem" happens when a system becomes ineffective because it uses "old" technologies and must be updated or replaced. When the life cycle becomes still longer, the aging problem recurs every time a system incorporates another emerging technology.

Summary of Technological Aspects

We contend that problems in the technological aspect are caused by the need to bridge and incorporate various technologies. Efforts will be wasted if every group of developers independently solve the difficulties induced by these problems, instead of developing common, domain-wide solutions that will be used by all the development groups.

1.2.4 Summary of the Problems

The previous sections describe problems in the engineering, managerial, and technological aspects of software development. These problems are summarized in Table 1.1.

Table 1.1 Problems Faced in Development of Large and Complex Systems

Aspect	Characteristic	Difficulties	Problems
Engineering	More than one system	Additional efforts are required for the integration of systems	Current methods do not fit development of more than one system, with multiple and unstable requirements
	More than one group of developers	The overall view of the system is neglected	
	More Than one customer	Multiple requirements	
	Heterogeneous environment	Engineering solutions are required to close technology gap	
	Long life cycle	Unstable requirements	
Management	More than one system	General objectives are neglected	There is no clear distinction between general, long-term objectives and local, short-term objectives
	More than one developer	Coordination and communication problems on a larger scale	
	More than one customer	Different aims and needs	
	Heterogeneous environment	No standardization of tools	
	Long life cycle	Long term objectives are neglected	
Technology	More than one system	Heterogeneous environment	There is a need to bridge the various technologies and efficiently incorporate emerging technologies as a common domain-wide solution
	More than one developer	Each development group has to struggle independently with Heterogeneity and dynamic environments	
	Heterogeneous environment	Bridging different technologies and incorporation of new technologies is required	
	More than one customer	Customization to user environment	
	Longer life cycle	Dynamic environment requires incorporation of new technologies	

CHAPTER 2

MEGA-SYSTEMS

Chapter 1 describes difficulties in development of software systems exhibit one or more of the following characteristics:

- Consist of more than one system,
- Developed by more than one group of developers,
- Have a large and heterogeneous group of users,
- Have More than one customer,
- Operate in a heterogeneous technical environment.

Since the development of systems with these characteristics is complicated and requires more effort than the development of traditional systems, we propose calling these systems: ***Mega-Systems***, because they are "beyond" traditional systems.

We contend that problems in the development of these systems are caused by the use of improper approaches and that it is possible to develop them more efficiently by using new approaches appropriate to the special characteristics of these systems. However, to propose such approaches it is first required to understand the structure of these systems and the relation between their components. This chapter defines Mega-Systems and classifies them. This classification can also be used to identify possible Mega-Systems.

There are several kinds of Mega-Systems: ***Huge Systems*** (HS), ***Systems of Systems*** (S2), and ***Generic Systems*** (GS), distinguished by the manner in which their

elements are related. The following sections define these kinds of Mega-Systems. Figure 2.1 illustrates our taxonomy of Mega-Systems.

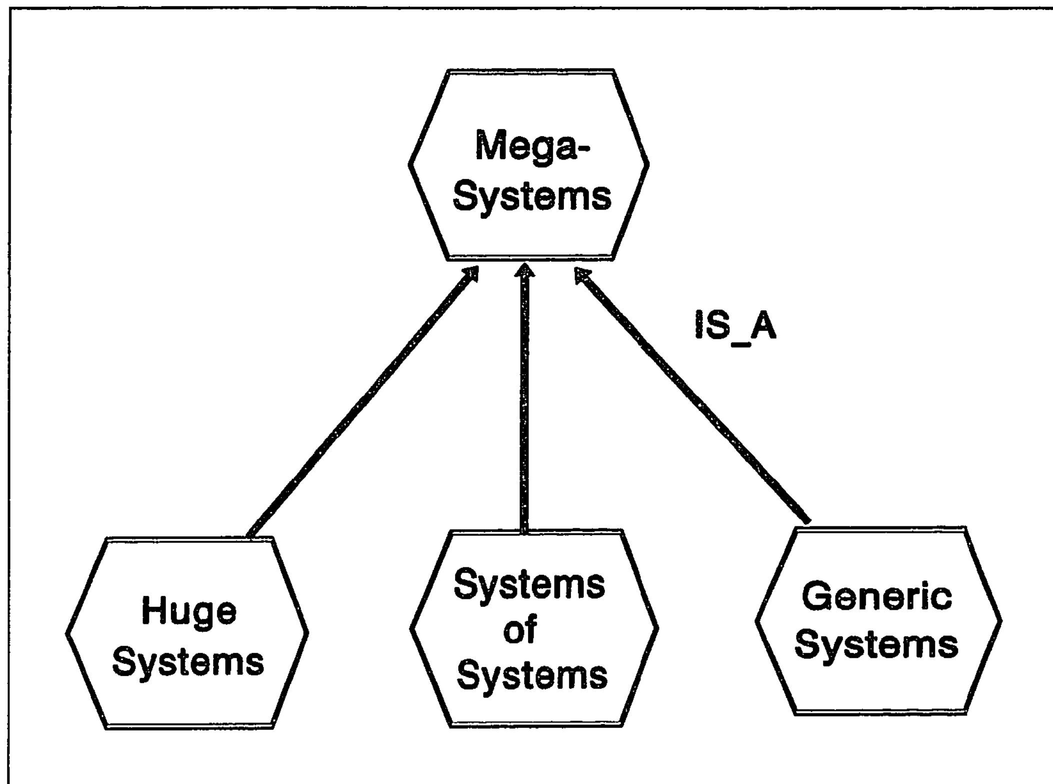

Figure 2.1 Mega-Systems

2.1 Huge Systems

Huge systems as defined in [YOUR 92] are large and complex software systems with hundreds of thousands to millions of lines of code, and hundreds of programs and modules. They are typically composed of multiple, large, and interrelated subsystems, each of which is designed to operate only as ***part of*** the huge system and in the its

environment. Huge systems often intensively process large, complex databases in a manner that precludes separation into smaller parts. Figure 2.2 illustrates the relationship of the parts of a huge system.

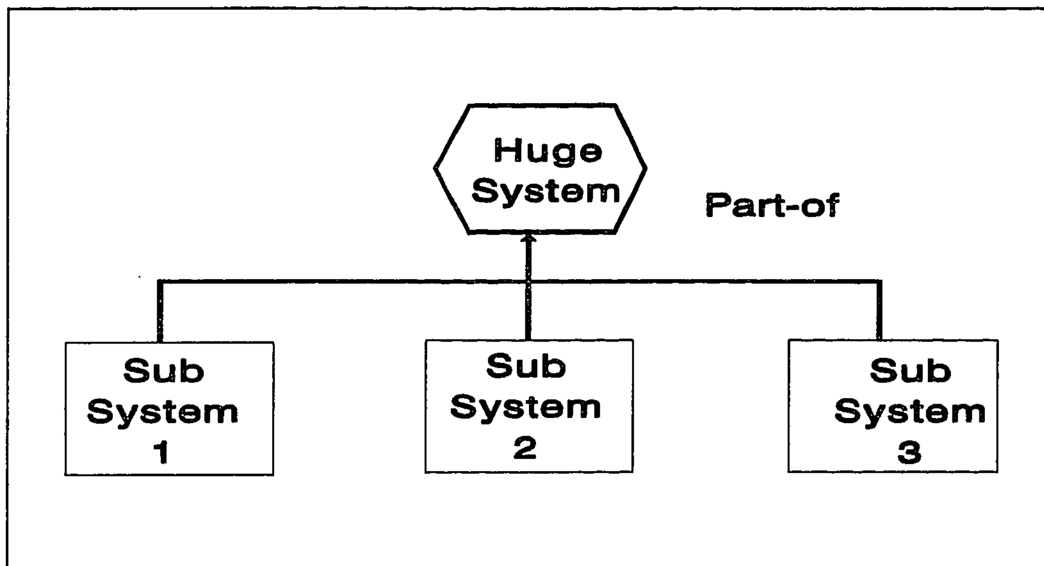

Figure 2.2 A Huge System

An integrated CASE tool, developed to support the different phases of a software development life-cycle which is based on system specific database management system and user interface tool is an example of a huge system. Such a system includes many programs, probably spread over many subsystems, but no part acts as a stand-alone system.

Huge systems are developed for a particular group of users, e.g., a specific group within a large organization. The group usually has a common role and requires a precise set of functionalities.

Huge systems are developed and maintained by a large group of developers. The developer group may be divided into smaller groups based on either system functions,

(where each group develops a specific part of the system), or the professions of the developers, (analysts, designers, and programmers). However, these groups typically belong to a single company or organization.

While huge systems are developed like traditional systems, their size, complexity, and length of life cycle induce development, maintenance, and integration difficulties. Their development is long and their maintenance is continuous and difficult, because they evolve over time and undergo generations of change. The interrelations between the subsystems make modifications problematic. Due to the usual scaling up effect, the amount of management and coordination needed for their development is much greater than for a traditional system [CSTB 90], [EISN 91], [MITT 91], [YOUR 92].

2.2 Systems of Systems

A second type of Mega-System is the "system of systems" (S2) [EISN 91], [ROSS 91a]. Systems of systems integrate several independently developed systems. Each component system is a product by itself, but is ***integrated with*** other autonomous systems to form a Mega-System.

An example of a system of systems is the FAA's advanced automation system for air traffic control [HUNT 87]. This system of systems is composed of several "large scale" systems that operate within the context of the overall, coherent mission of providing safe, cost-effective, passenger and freight, air transportation. The "air traffic control system"

integrates systems that provide communications, navigation, radar, control, and other automation capabilities [EISN 91].

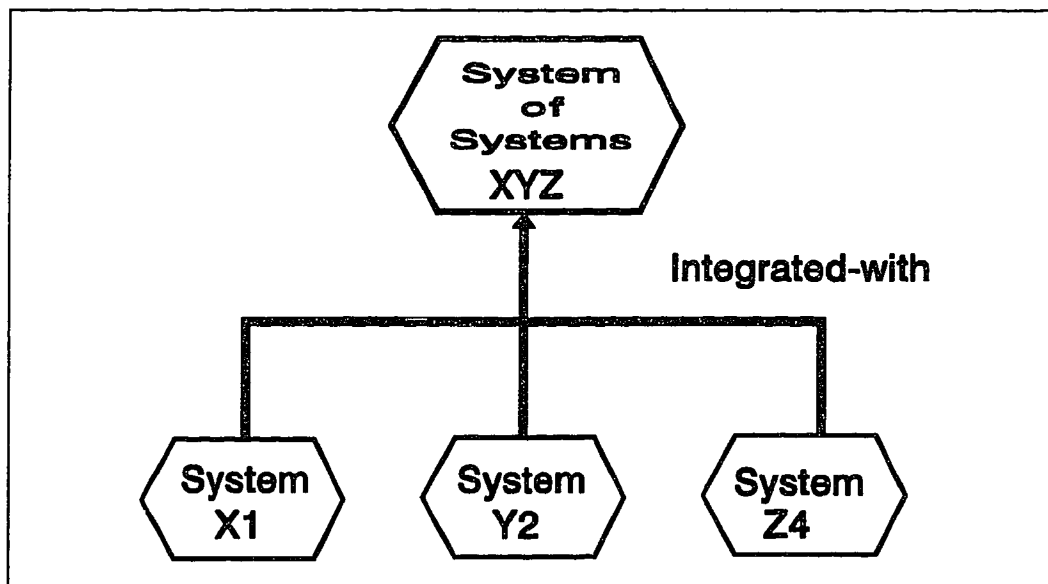

Figure 2.3 A System of Systems

Systems of systems, usually, have a large, heterogeneous group of users. These users have different roles and require different functionalities. The systems of systems may also facilitate the work of these users as a group.

The systems forming the system of systems are generally developed by separate groups of developers, at separate sites, with different schedules. These developer groups belong to different organizations which have different aims and goals, and often have different standards, techniques, and methods for developing systems.

In contrast to huge systems, we may not have full knowledge of the functionality of the system of systems in advance. Each part (system) of the system of systems can be a stand-alone system, so it evolves over time; decentralized growth of the systems is typical .

Most systems that are integrated into a system of systems were developed without planning for future integration and with limited consideration of other systems. Thus, most systems of systems are integrated in a post-facto manner [POWE 90]. However, some systems of systems are developed in a pre-facto manner that requires knowing all components of the system in advance and developing them from scratch [POWE 90].

2.3 Generic Systems

The third type of Mega-System is the generic system. A generic system is a specification of a set of interrelated functionalities and the actual systems derived from this specification. A functionality is specified on an abstract and conceptual level by formal definitions or natural languages. Different systems are then derived by instantiation or specialization of the abstract functionalities.

Instantiation is done by implementing the given set of functionalities using a specific programming language, specific hardware environment, etc. ***Specialization*** is done on the level of specifications by adding new or removing existing functionalities without changing the essential characteristics. (Although "essence" is a qualitative and subjective criteria, we suggest using it to avoid cases in which systems are derived by removal of all/most the original functionalities and addition of new functionalities. Quantitative criteria, e.g., the number of functionalities, will be useless in this case). Subsequent to specialization, new systems can be derived by instantiation. The derivation

can be done manually or by code or application generators. Figure 2.4 illustrates the relationship between the components of a generic system.

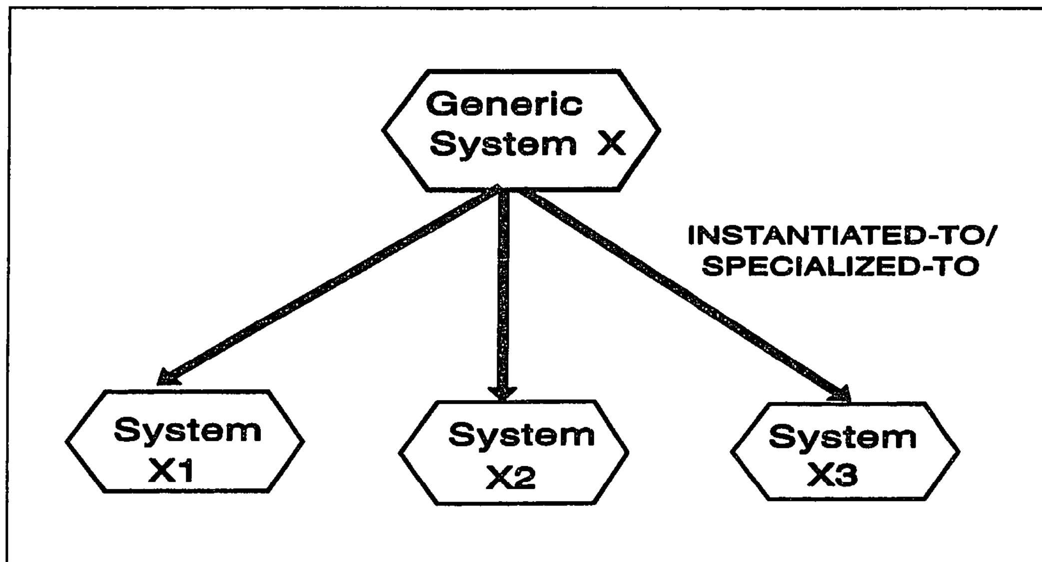

Figure 2.4 A Generic System

A radar system is an example of an embedded generic system. Radar systems can be installed in planes, ships, or ground stations. All systems share common functionalities such as user-interface, signal-processing, and communication. However, any individual system has a special configuration instantiated/specialized from the original set of functionalities of the generic radar system and suited to the requirements of its customer.

Systems derived from a generic set of functionalities are typically developed for different customers, so generic systems generally have multiple user groups. Each derived system is developed by a different group of developers. Though these groups belong to the same organization and develop systems with similar functionalities, they not always work in coordination.

Generic systems are developed using traditional methods. Lack of coordination between developer groups leads to redundant functionalities and inefficiencies. Unlike huge systems and system of systems, the components of generic systems are similar systems. Although the basic functionality of the generic system is specified in advance, extensive adaptations of the system are possible.

2.4 Generic Systems of Systems

Difference in types of user groups and time frames for the use of systems suggest defining an additional type of Mega-Systems, the Generic Systems of Systems which can be considered as a subclass of both systems of systems and generic systems. Generic Systems of Systems solve a problem for a domain, with no precise time frame and without definite users.

A generic system of systems has the flexibility of a generic system and can be specialized and instantiated into different configurations, but each functionality is implemented as a system and each configuration as a system of systems. Figure 2.5 illustrates the relationship between the components of a generic system of system.

An example of a generic system of systems is a system for insurance agencies. An instantiation of the system for a large insurance agency will operate on a mainframe with multiple terminals. An instantiation for small agencies will use personal computers connected by a network. These instantiations differ in their environments. It is also

possible to specialize the set of original functionalities by adding or removing systems. For example, a specialized system for a general agency could include life, vehicular, and property insurance, and accounting systems. A specialized system for a life insurance agency might include an accounting and a life insurance system.

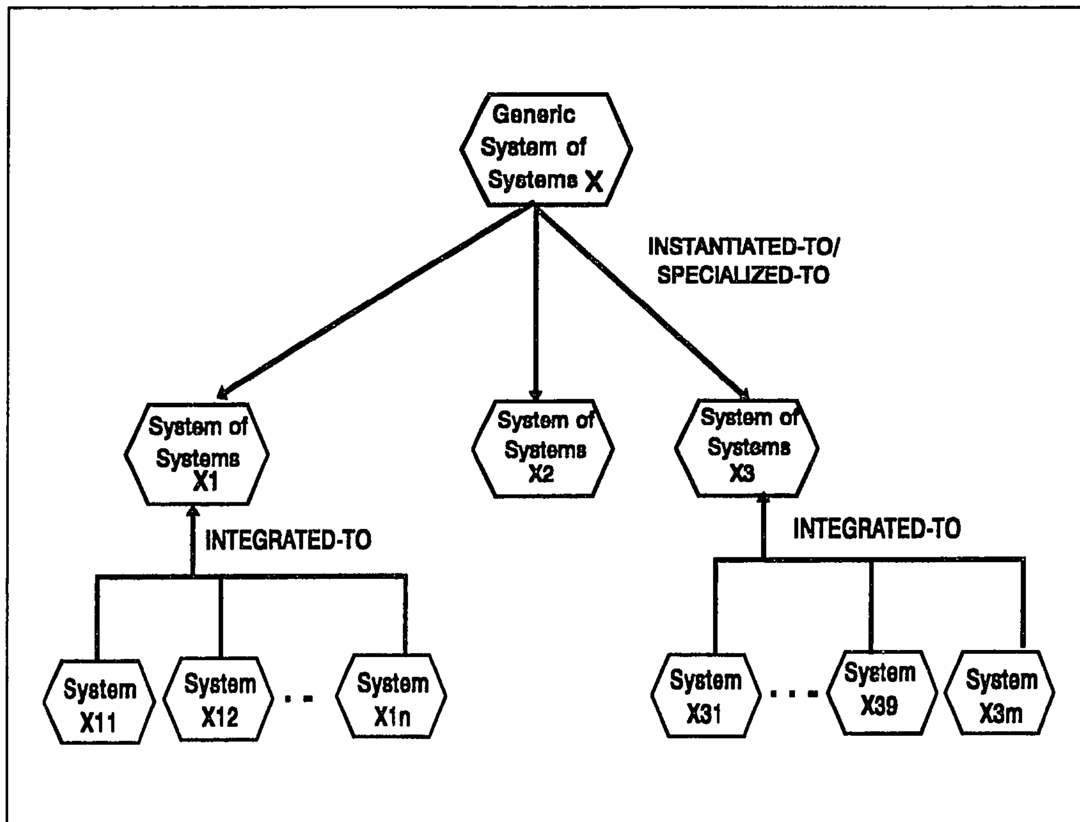

Figure 2.5 A Generic System of Systems

2.5 Relationships among Mega-Systems

Figure 2.6 summarizes the relationships among the types of Mega-Systems. Mega-Systems are large, complex systems. Huge Systems, System of Systems, and Generic

Systems are different sub-classes of Mega-Systems. System of Systems and Generic Systems (GS) have a common sub-class: Generic System of Systems (GS2).

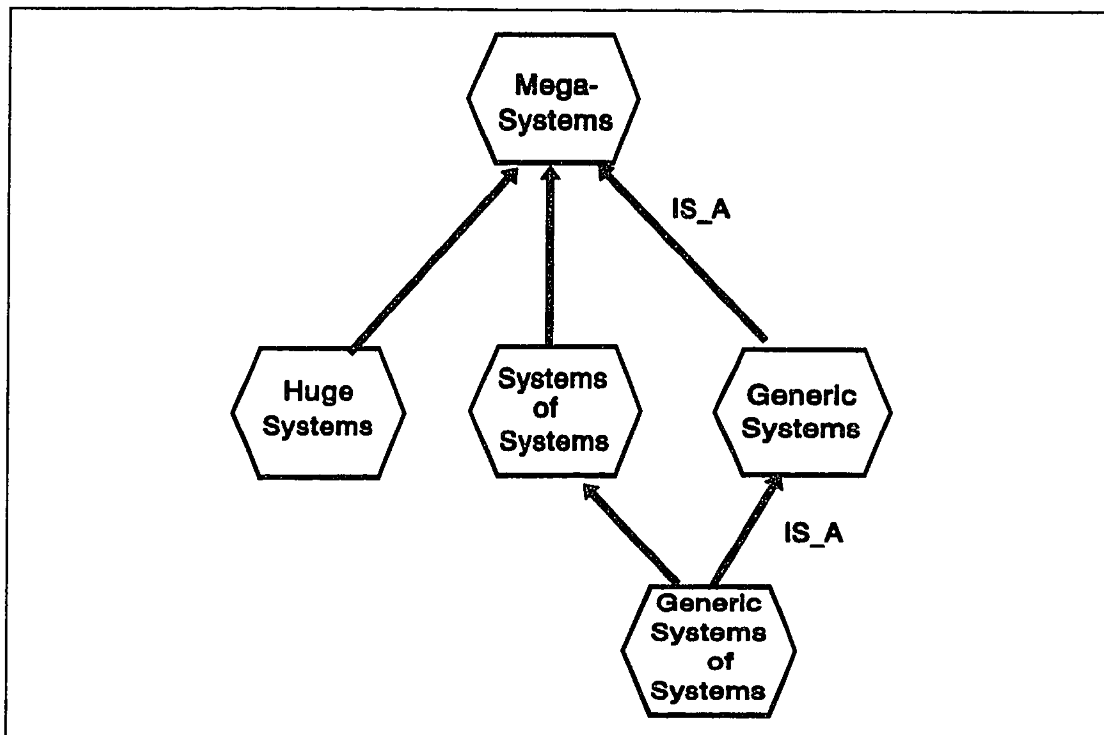

Figure 2.6 Detailed Classification of Mega-System

Table 2.1 summarizes the characteristics of traditional systems and Mega-Systems. It allows us to classify the type of a system and consequently determine a suitable approach for its development. For example, a system with stable requirements and a limited user group, might be developed using traditional approaches. On the other hand, a system with dynamic requirements, several user groups, and multiple configurations, each operating in a different environment, should be developed as a generic system of systems.

Table 2.1 A Comparison of Mega-Systems

ATTRIBUTE	Traditional Systems	Mega-Systems			
		Huge Systems	System of Systems	Generic System	Generic System of Systems
Requirements	Stable, known in advance	Stable, known in advance	Dynamic	Partly stable, adaptations are feasible	Dynamic adaptations are feasible
Customer	Limited user group	Large user group	Hetero-geneous user group	Multiple user groups	Multiple Hetero-geneous user groups
Developers	One group within one organization	Large group within one organization	Several groups that belong to different organizations	Several groups that belong to the same organization	Several groups that belong to different organization
Life Cycle	Short	Long	Long	Long	Long
Components	Sub-systems	Large Sub-systems	Independ-ently developed systems	Derived Systems	Independ-ently developed systems
Relation of system and components	Part-of	Part-of	Integrated-to	Derived form	Instantiated-to and integrated
Environment	Homo-geneous	Homo-geneous or Hetero-geneous	Hetero-geneous	Different environments	Different hetrogeneous environments
Configurations at a given point of time	One	One	One	Several (each for a different customer)	Several (each for a different customer)
Management	One project	One big project	Several projects	Several projects	Several projects

CHAPTER 3

A FRAMEWORK FOR MEGA-SYSTEM DEVELOPMENT

This chapter describes the characteristics required of a framework for the development of Mega-Systems. Section 3.1 describes existing models for development of large and complex systems. Section 3.2 specifies requirements for a framework. Section 3.3 outlines the main concepts of MegSDF.

3.1 Existing Models and Frameworks

Mega-Systems are currently developed using traditional approaches, e.g., the waterfall and its variations [BOEH 76], Prototyping [GOMA 90], the Spiral Model [BOEH 88], etc. Several solutions have been suggested for developing large scale systems and for systems integration which are related to MegSDF. There are basically two approaches. The first emphasizes development organization aspects: the activities, elements and organization of software development, e.g., COSMOS [YEH 91], GenSIF [ROSS 91a, b, c], POWDER [MITT 91], SIF [GEHD 91], and System of Systems Engineering [EISN 91]. The second type emphasizes mega-programming languages that allow interaction of systems, e.g., MPL [WIED 92], and LILEANNA [TRAC 91]. These languages were developed in the context of DARPA's Megaprogramming projects. MegSDF addresses the development

process, though megaprogramming tools can be used within the process to improve and support development.

3.1.1 The COSMOS Model

Yeh et al. [YEH 91] have defined COSMOS, A COmmon Sense management MOdel for Systems, based on the notion that developers of large systems must consider long term objectives. Long term applications require flexibility and ease of maintenance/enhancement, since it is impossible to eliminate changes in the system. In order to accommodate these changes efficiently, trade-offs should be considered from three perspectives: Activities, Communication, and Infrastructure. To maintain a balance between those perspectives COSMOS suggests two process levels: Control and Execution. These levels apply to any perspective. The tasks of each level and for each perspective are defined. The model is applicable to software and non-software systems.

COSMOS proposes developing a system through a series of small changes. At each change, the balance among the three perspectives must be maintained and implemented by the two process levels.

3.1.2 The GenSIF Framework

Rossak [ROSS 91a, b, c] has proposed GenSIF, A Generic Systems Integration Framework, which divides the development of a system of systems into several projects. GenSIF includes two levels of management: an upper management level (meta-level) , and several lower (project level) managements. The meta-level management is responsible for

leading the development of the system of systems, as well as for communication and coordination between sub-projects. The lower level project managements are responsible for developing each system.

GenSIF includes domain analysis, integration architecture, and infrastructure as main concepts. The framework defines two levels for an integration architecture [ROSS 91c]. The first, conceptual level architecture describes guidelines and standards for the development of the entire system. The second level is the technical infrastructure. This level deals with the standardized services that are an essential part of any system. These services include communication, data storage, and user interface.

3.1.3 The POWDER Methodology

Mittermeir developed POWDER, a recursive methodology for Prototyping Of Wicked Development Efforts with Reuse [MITT 91]. POWDER is a methodology for software development, based on generally applicable techniques used to solve wicked problems. The methodology divides the development into sub-projects, and divides the process into control and execution levels. The control level is responsible both for steering the development and for the integration platform. The execution level is responsible for the actual work done in the different sub-projects. A large sub-project at the execution level may require further control and execution sub-levels. Thus, POWDER supports a recursive organization. The framework includes descriptions of the responsibilities of each level. The method for implementation of each sub-project is chosen according to the attributes of the sub-project. The POWDER model can be used for any system and type

of integration. The framework includes guidelines for choosing a development approach for various types of projects. Figure 3.1 describes the organization/task structure of the model from our viewpoint.

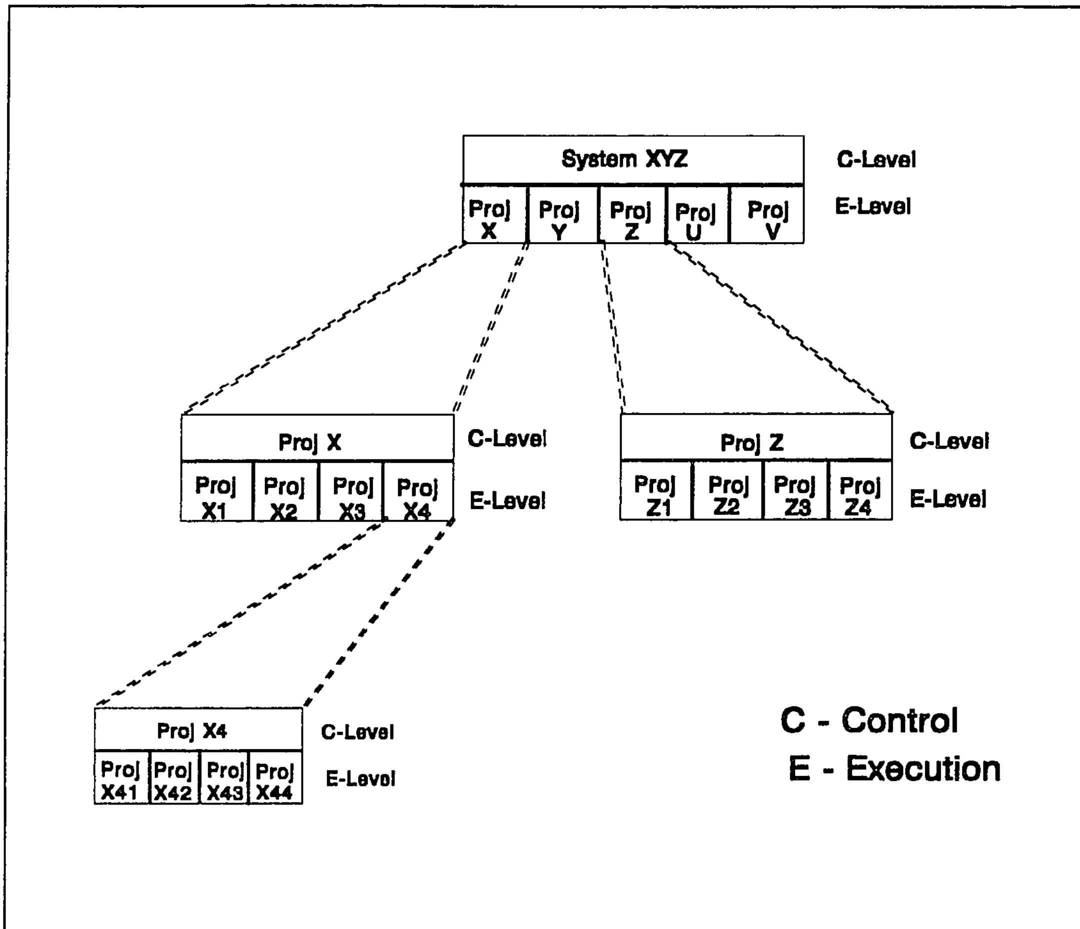

Figure 3.1 The POWDER Model

3.1.4 The SIF Framework

[GEHD 91] proposed SIF, a Systems Integration Framework. SIF identifies problem "tracks" including: technology project management, technology change management,

technical platform development, custom applications development, testing and implementation, package selection and implementation, application operation, data modeling, and software re-engineering. Each track has its own methods, techniques and tools.

SIF suggests that the first step of each systems integration process should be the identification of the track(s) the system belongs to. For each track, deliverables and milestones, activities and their dependencies, techniques and tools, and appropriate quality assurance measurements are defined. The interrelation between the tasks are then addressed. The process is iterative and dynamic. The approach leads to customized solutions where each system is developed by methods, techniques, and tools tailored to the special needs of the system. The model is useable for any type of system development or systems integration.

3.1.5 System of Systems Engineering Model

Eisner et al. [EISN 91] suggest a model for system of systems (S2) engineering. They characterize a system of systems as a multi-functional system with several independently acquired, interdependent systems. The local optimization of a system in a system of systems does not guarantee global optimization of the entire system. The combined operation of the systems satisfies the overall coherent mission. System of systems engineering requires developing autonomously managed systems under an overall supervising management.

System of systems engineering is based on a meta-system engineering framework which uses three categories: integration engineering, integration management, and transition engineering. Integration engineering involves all the engineering necessary to fully integrate the component systems. Integration management focuses on the management aspects of systems of systems, emphasizing scheduling, budgeting/costing, configuration management, and documentation. Transition engineering focuses on assuring an orderly transition from the collection of stand-alone systems to the integrated system of systems.

Eisner et al. contend CASE-tools are critical in engineering systems of systems. CASE tools can enhance developers' productivity, and facilitate impact studies, interface analysis, performance analysis, scheduling, budgeting, and documentation.

3.1.6 The Megaprogramming Framework

DARPA has encouraged research on the problems of scaling up software engineering, for which they introduced the term "megaprogramming". Wiedrehold et al. [WIED 92] propose a framework and Megaprogramming Language (MPL) for megaprogramming using software components called megamodules. Megamodules capture the functionality of services provided by large organizational units, e.g., banks, airline reservations, or city transportation systems. Computations spanning more than one megamodule are specified by megaprograms using a megaprogramming language. Megamodules encapsulate data, behavior, and knowledge, and support multiple concurrent activities. A megamodule is

operated and maintained autonomously and is a potential component of many megaprograms. Megamodules can be developed by traditional technologies.

Megamodules require module interaction mechanisms that support their encapsulation, heterogeneous interfaces, and dynamic evolution. [WIED 92] proposes a MegaProgramming Language (MPL) to allow flexible composition of megamodules and support synchronous and asynchronous coordination schemes, decentralized data transfer, parallelism and conditional execution. It supports the autonomous operation of megamodules and allows asynchronous operations controlled by the megaprograms. Input/output parameters are presented with database-like schemas.

MPL separates input/output management from the invocation mechanism in CALL statements. MPL includes operations for megamodule interaction, e.g., inspection of interfaces and contents of megamodules, and examination of the status of a megamodule.

A megaprogramming system consists of a collection of distributed megamodules linked by a network. A megaprogramming environment includes a repository and dictionary that support megamodule execution and maintenance.

3.1.7 Summary of Existing Methods

Table 3.1 compares the approaches just described. Basically, the approaches call either for two levels of managements or two levels of programming. The COSMOS model suggests developing systems by an evolutionary approach consisting of a sequence of small changes. Other approaches propose dividing the development effort into smaller projects. The models do not define explicit processes for developing such systems, methods of

partitioning into sub-problems, or methods for assuring engineering coordination of projects.

Table 3.1 Existing Models

Model	Organization	Parts	Main concepts
COSMOS	Control Execution	Small Changes	Balance activities, communication, and infra-structure in each change. Long term objectives should be considered. Flexibility and ease of maintenance are essential.
GenSIF	Meta-level management and several lower level management	Projects	Domain analysis, integration architecture, and infrastructure are main elements.
SIF	Not defined	Problem Tracks, e.g., technology management, application development	It is required to determine what problems tracks characterize the system and their implications.
POWDER	Control and Execution with recursive structure	Projects	Each project should use an appropriate approach for its development.
System of Systems Engineering	Meta management	Systems	Integration engineering, integration management, and transition engineering. CASE tools are mandatory
MPL	Each megamodule is autonomous and developed and maintained separately	Megaprograms and Megamodules	Traditional methods for development of megamodules. Megaprogramming language with separation of I/O management from invocation.

3.2 Requirements for a Framework

The problems in developing large, complex software systems discussed in chapter 1 lead us to conclude that a new approach for developing Mega-Systems is required. The complexity and the variety of problems dictate more than an engineering solution; other aspects of development must also be incorporated in the framework for developing Mega-Systems. In order to address the problems in software systems development (summarized in table 1.1), a framework for developing Mega-Systems must be:

- General,
- Comprehensive,
- Operative, and
- Open

The framework must be ***general***. That is, it must be useful for different application domains such as data-processing (banking, insurance, manufacturing) and real-time applications (naval systems, avionic). It should also suit the different types of Mega-Systems.

The complexity of development and the number of developers involved in Mega-Systems require a ***comprehensive*** framework that incorporates engineering, managerial, and technological aspects [DAVI 92]. A solution that addresses only the difficulties involved in engineering aspects will be insufficient.

The framework must be ***operative.*** That is, it must specify the activities required to develop a Mega-System, their deliverables, and their interconnection and sequencing, and integrate them into a coherent, efficient process model.

The framework must also be ***open*** and flexible. Developers of Mega-Systems must have the option of selecting an appropriate technique for implementing an activity. The technique must fit both the characteristics of the problem and the experience and knowledge of the developers. This also applies to the selection of tools that support a specific technique. The framework must be adjustable to the actual needs of the domain.

3.3 Outline of MegSDF Framework

We propose MegSDF - a framework for Mega-Systems development - which satisfies the general requirements for a framework, and addresses the problems in development identified in chapter 1 and the limitations of existing models for development. The main concepts of the framework include:

- Two levels of organization,
- Engineering coordination,
- A pre-planned approach, and
- Development as open, distributed systems.

We will briefly motivate these concepts, then, in the following sections elaborate on them.

The complexity of Mega-Systems development scales up management issues, so that management aspects must be included in the framework. We propose an organization with two levels of management in order to guarantee/enforce the distinction between, and attention to, overall development and coordination, as opposed to purely local considerations.

The characteristics of Mega-Systems also lead us to propose a new engineering process for their development. The process is specified by a process model which includes: definitions of activities, their relations, deliverables, and sequencing. The process promotes engineering coordination of all systems developed in the domain by using (what we call) a domain model, a Mega-System architecture, and an infrastructure, which are derived in Mega-System tasks.

To facilitate future changes, integration of new functionalities, and incorporation of emerging technologies, we recommend a pre-planned approach. Finally, to realize the previous concepts we propose developing the Mega-Systems as open, distributed systems.

3.3.1 Development Organization

The size and complexity of Mega-Systems preclude their development as single systems. Therefore, Mega-Systems consist of multiple systems. Naturally, developing their constituent systems without coordination is ineffective. To provide for the requisite global coordination, we propose developing a Mega-System as a ***mega-project*** that includes multiple coordinated projects. Each project develops a smaller constituent system of the Mega-System. To ensure that the distinction between general, long term issues as opposed

to local. short-term issues is maintained, we define two levels of management. ***Meta-management*** controls the mega-project. Projects are controlled by lower level ***project managements***. Figure 3.2 illustrates the proposed organization of systems, management, and projects. (Though huge systems currently do not include systems as components, we recommend that in the future huge systems be developed as systems of systems).

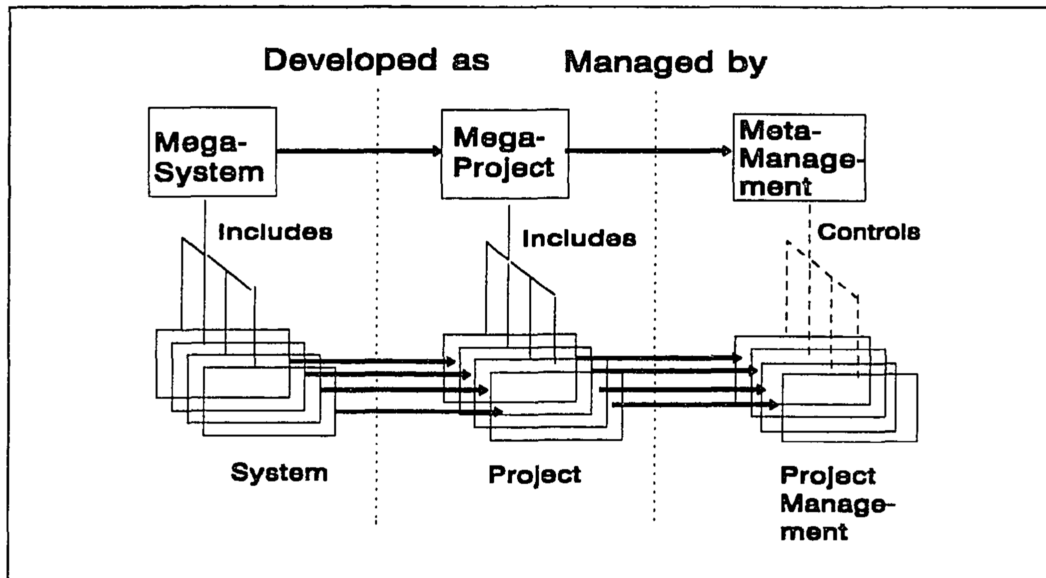

Figure 3.2 Mega-System Development Organization

Meta-Management

Overall management is essential for a mega-project [EISN 91], [MITT 91], [ROSS 91a], [YEH 91]. The meta-level management guides and controls the development of the whole system. It determines policies and directions for the system and guarantees communication and coordination between the different projects. Meta-management communicates with the customers to guarantee the effectiveness of the trends and directions of the system. It maintains a balance among the multiple requirements and divergent needs of the

customers. Meta-management determines global priorities and schedules. Meta-management should include managers of the smaller projects, as in POWDER [MITT 91]; this promotes efficient communication and coordination. Meta-management decisions should be based on risk analysis [CHAR 89] to identify the real problems early and allocate resources appropriate to solve them.

Lower Level Management

Lower level project management controls either development of a small system or the customization of a Mega-System according to customer needs. It is responsible for local and temporary issues. Each project should be developed as a part of the whole system and be coordinated with other projects. Each constituent system should be developed according to its own attributes as recommended by [MITT 91]. The development approach should be selected based on the experience and development tools of the developers.

The relation between meta-level and lower level management should be flexible. The type of management - centralized or decentralized - depends on the project attributes. Risk analysis can be used in determining the relation between management levels. The degree of autonomy of project management may vary among different projects. A project may be so large that its management needs to define sub-tasks to accomplish it. For example, developing a system using the waterfall model may entail multiple sub-tasks, where each sub-task corresponds to a phase of the model. The approach is recursive, similar to POWDER [MITT 91].

3.3.2 Engineering Coordination

The drawbacks of current development models are rooted in the lack of engineering coordination. While meta-management balances customer requirements, and determines an appropriate schedule, there remains a need for concepts and tools to facilitate engineering coordination of the projects. The constituent systems should not be developed as isolated solutions to limited parts of the problem. Thus, our framework must provide: an overall, general view of the problem space, a plan for the system as a whole which clarifies the role of each constituent system within the entire Mega-System, and recommendations for uniform use/handling of technologies.

In our process, ***Domain Analysis*** provides a universal, general, comprehensive ***domain model.*** It provides a common understanding of the problem, and facilitates early identification of future requirements.

The ***Mega-System architecture design*** defines common design and implementation concepts in a ***conceptual architecture*** and the overall structure of the system in an ***application architecture***. The conceptual architecture ensures the integratability and uniformity of the constituent systems. These concepts can also enhance productivity of development by defining common solutions. The application architecture maps the application domain to implementation and identifies the interrelation of the constituent systems.

Infrastructure acquisition provides a unified environment of enabling technologies through an ***infrastructure***. The infrastructure is used as a common solution for technologies handling by the different projects.

All these elements promote engineering coordination for the entire development effort. The next chapters further elaborate on these elements and integrate them into an engineering process.

3.3.3 The Pre-Planned Approach

Mega-Systems tend to become long-term solutions. Their size and complexity entail extensive development effort and correspondingly high investment, so it is impossible to develop a Mega-System over a short period and infeasible to replace it after a short time. On the other hand, application domains are dynamic and systems themselves influence these dynamics [LEHM 90]. Changing requirements are unavoidable, and so systems must be planned for change [CSTB 90], [YEH 91]. Furthermore, the length of system life cycles often implies that the technologies in which the systems were originally developed will become obsolete; obsolete technologies must be replaced to assure systems effectiveness and user competitiveness [CLEM 91].

In the light of these characteristics, we must plan for flexible systems with long life cycles [CSTB 90]. The development of such systems should be evolutionary; different parts should be developed, modified, or replaced over time according to the needs of the application domain. This type of development requires a dynamic organization. Meta-management is responsible for defining the various parts, for deciding when to start developing a part, and for stopping or suspending the development of a part. While the meta-management is active during the whole life of the Mega-System, projects for the

development of the different parts are active according to the progress of the Mega-System.

Most existing Mega-Systems were originally designed as regular systems. They became Mega-Systems that integrate (or incorporate) multiple systems only because their characteristics changed over time in response to customer needs. Power [POWE 90] has proposed classifying the process of systems integration in which systems of systems are formed on the basis of the order of design and implementation of the component systems and the whole system of systems. This classification includes both post-facto (or a posteriori) integration and pre-facto (a priori) integration, as well as a mixture of these types.

Post-facto Integration refers to the integration of multiple systems that were developed before the system of systems was even specified. Post-facto integration is constrained by its need to integrate existing systems usually developed by separate groups, with diverse standards and procedures, according to isolated requirements, and not designed to be integrated. Interfacing such systems requires extensive effort. Compromises are often required, either in easing requirements to allow reuse of existing software, or in redeveloping systems to comply with requirements. Despite its inherent complexity, post-facto integration may be appropriate in some cases because the use of existing systems reduces risk and uncertainty.

Pre-facto Integration addresses the integration of systems that are planned and developed to work together. All the parts or systems of such an integration are assumed to be known in advance. Each part is designed to operate in the context of the system of

systems. The objective of pre-facto integration is to improve the productivity of development and systems quality and flexibility. Since no part of the final system already exists, it is possible to design and implement the system and its parts very efficiently. Even though the constituent systems are designed and developed separately, they are planned with the knowledge that they must be integrated into a single system. However, despite its efficiency, the pre-facto approach tends to be inflexible to change. Furthermore, although pre-facto integration is more desirable from the integrator's viewpoint, experience has shown that it is infeasible to use only pre-facto integration: systems must also integrate components developed before the design of the system began [POWE 90] and also adapt to long life cycles with on-going changes.

Power's classification of systems integration is also applicable to Mega-Systems. Thus, a pre-facto Mega-System is one designed to be a Mega-System in advance: all the requirements for its parts and configurations are known prior to design and implementation. In contrast, a post-facto Mega-System is a set of systems developed as traditional systems, which later, due to new requirements, becomes a Mega-System: its parts and configurations are not known in advance.

Pre-Planning

In reality, it is impossible to foresee what future requirements will be. Hence, the pure pre-facto approach is infeasible. On the other hand, the post-facto approach is inefficient and entails excessive integration effort. We recommend using a ***pre-planned*** approach in order to overcome these problems.

The pre-planned approach advocates defining concepts and tools that will facilitate future integration, changes in requirements, and incorporation of new technologies. It includes elements of pre-facto and post-facto integration. It specifies an environment that facilitates integration of systems, as in the pre-facto approach. However, this environment is open and does not require knowledge of all elements of the system in advance, allowing the integration of existing systems, required for post-facto integration. The previously mentioned means for engineering coordination support these concepts by allowing early identification of future needs and facilitating integration of systems.

3.3.4 Development as Open Distributed System

In order to realize the preceding concepts (two levels of organization, engineering coordination, and pre-planning) Mega-Systems should be developed as open, distributed systems consisting of multiple interdependent, but self-contained, systems. ***Open*** refers to the fact that the systems include well defined interfaces which facilitate future integration. ***Distributed*** means the Mega-System is composed of smaller constituent systems forming a federation of systems [SHET 90]. Each constituent system is autonomous but prepared to share functionality, data, etc., with other current (or prospective) systems of the Mega-System. The constituent systems are defined following the domain model and according to the application architecture; their design conforms to the common design principles of the conceptual architecture. To accomplish these characteristics, the Mega-System is implemented using an infrastructure that enables

interaction of constituent systems, bridges underlying technologies, and uniformizes heterogeneous environments. Refer to figure 3.3 for an overview.

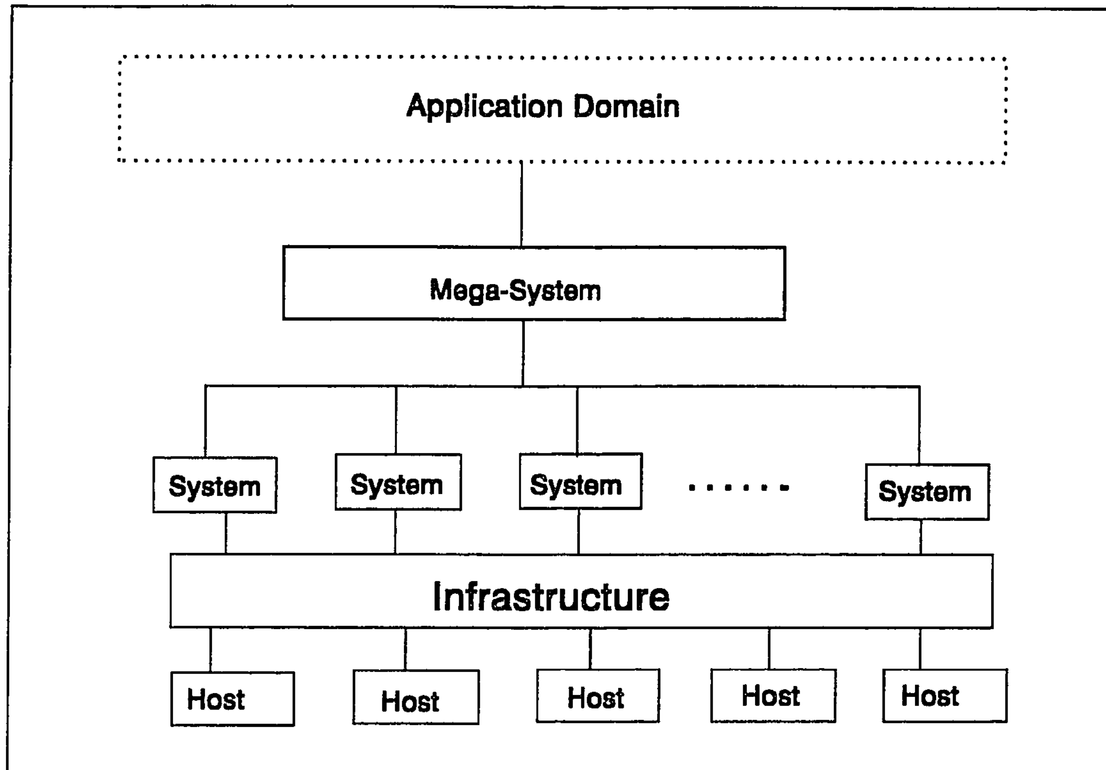

Figure 3.3 Mega-System

CHAPTER 4

MegSDF PROCESS MODEL

A framework for development of Mega-Systems has to specify the required activities for development of a Mega-system and the interrelation between its activities in order to be operative. The activities are defined in a process model [KOKO 89], [CURT 92], [TAYL 92] which has to be instantiated [PERR 89a] for every Mega-System development.

Research on software development processes has many facets. One approach evaluates software processes and proposes ways to improve them, e.g., [HUMP 88], [KRAS 92], [SCHL 92]. Other approaches try to improve the representation of the process model to support its control and automation [TULL 88], [PERR 89]. The MegSDF process model is used to specify the activities required for the development of a Meg-System.

4.1 A Method to Describe an Engineering Process

The graphical notations of Structured Analysis (SA) [DeMA 78], [WARD 86] and Structured Analysis and Design Technique (SADT) [ROSS 77], [DICK 78] have been used to define the software development process in [FREE 87] (as also suggested by [FRAN 92], [BLUM 92]). We will use a method that synthesizes both these approaches. A process is denoted by a process diagram that includes several tasks or sub-processes and

data and control flows that connect them. A task, together with its inputs and outputs, is described using the SADT activity primitive (Figure 4.1).

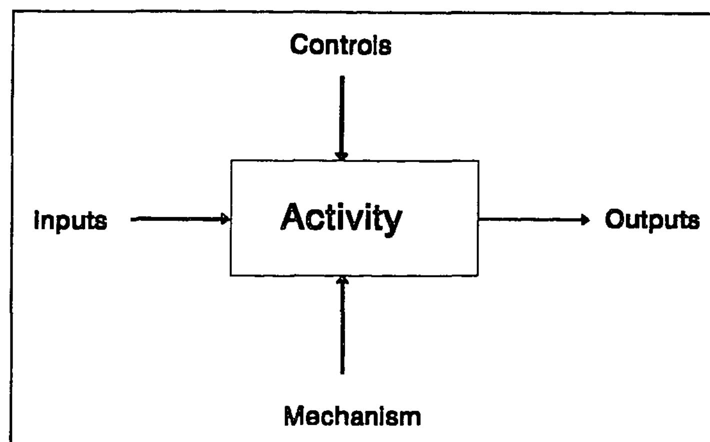

Figure 4.1 The Graphical Notation for a Traditional SADT Activity

We differentiate between management and engineering tasks using Ward-Mellor's notation [WARD 86]. An engineering task, e.g., domain analysis, is drawn as a solid box (Figure 4.2a). A management task, e.g., resource allocation, is drawn as a dashed box (Figure 4.2b). A flow is drawn as an arrow. A complex task is exploded in further process diagrams.

The MegSDF engineering process requires execution of multiple similar tasks concurrently, e.g., system tasks. Such tasks have the same entries and exits and the same processing, but are executed on different instantiations of the inputs and outputs and according to different schedules. These tasks are denoted as multiple boxes, (Figure 4.2c) similar to the multiple processes in the SA extension of [BLUM 92].

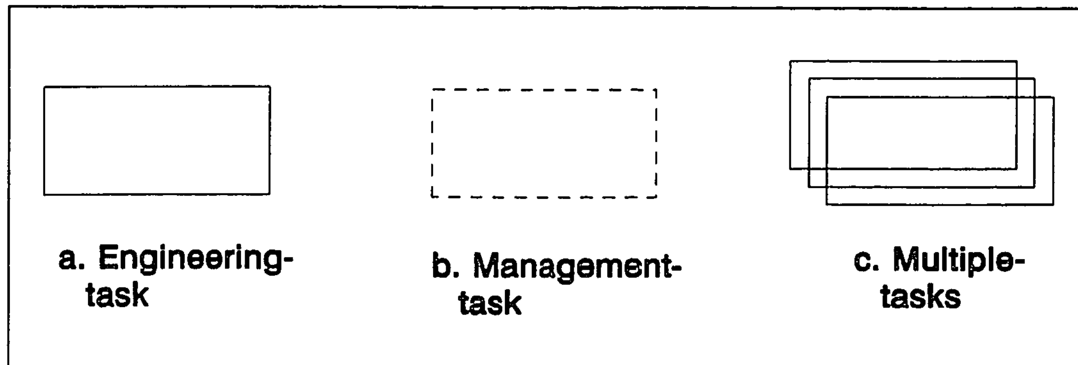

Figure 4.2 Graphical Notations for MegSDF Activities

Following [DICK 78], our synthesized method uses five types of flows: Inputs, Outputs, Mechanisms, Circumstance, and Execution controls. Inputs, Outputs, Mechanisms, and Circumstances flows are drawn as solid arrows. Execution Controls are drawn as dashed arrows. To avoid overloading the figures we use shared flows. A shared flow is connected to all tasks of the process, but drawn only to the boundary of the process diagram. Shared flows can be inputs, outputs, execution control, mechanism, or circumstance.

We follow the SADT positioning rules for flaws:

- Inputs enter from the left side of a task,
- Outputs exit from the right side of a task,
- Mechanisms enter from the lower side of a task,
- Circumstances enter from the top side of a task.
- Execution controls enter from the top side of a task.
- Execution controls exit from the right side of a management task only,
- Any entry or exit connected to multiple tasks refers to all tasks,
- Shared flows are drawn from/to the boundary of the process diagram.

Just as in structured analysis, the definition a process consists of: name, purpose, interfaces, processing, process diagram, timing, and description of tasks (sub-processes) of the actual process (if these tasks are not described separately). Process interfaces are divided into inputs, mechanism, circumstance, execution control, outputs, and task execution control outputs.

4.2 Mega-System Development Process Model

We define the process for Mega-Systems development according to the concepts of the framework discussed in chapter 3 using the notations of the previous section. This section focus on the first level of the process model and discusses the interaction between the main tasks. Subsequent chapters discuss the tasks of the process in detail.

4.2.1 Purpose

The purpose of this process is to develop a Mega-System.

4.2.2 Interfaces

Inputs

• Domain Data - Information regarding the domain in which the Mega-System is intended to operate.

• Customers/Users requirements - Requirements of the Customers/Users of the systems.

• Existing and Projected Technologies - Information about technologies that are already available and projected technologies that will be available in the future.

Mechanisms

• Modeling Approaches - Commonly available modeling approaches.

• Architectural Styles - Styles of conceptual architectures for Mega-Systems.

• Software Engineering Methods - Methods for developing software systems that can be used to develop systems in the domain.

Outputs

• Mega-System

4.2.3 Processing

The constituent systems are developed, under the supervision of the ***meta-management***, in the **system** tasks. The engineering aspects of the development are coordinated by the ***Mega-System*** tasks that provide the ***domain model***, a ***Mega-System architecture***, and a common ***infrastructure***. The Mega-Systems is constructed from the constituent systems in the ***Mega-System Synthesis*** *Task*. Feedback from the system and synthesis tasks are used to improve the engineering coordination tools provided by the Mega-Systems task and the global plans and schedule of the Meta-management. Figure 4.3 illustrates the interaction between the tasks.

The process assumes that verification, validation, and quality assurance are done as part of every task or sub-task to ensure that an effective and efficient system is provided to the customers.

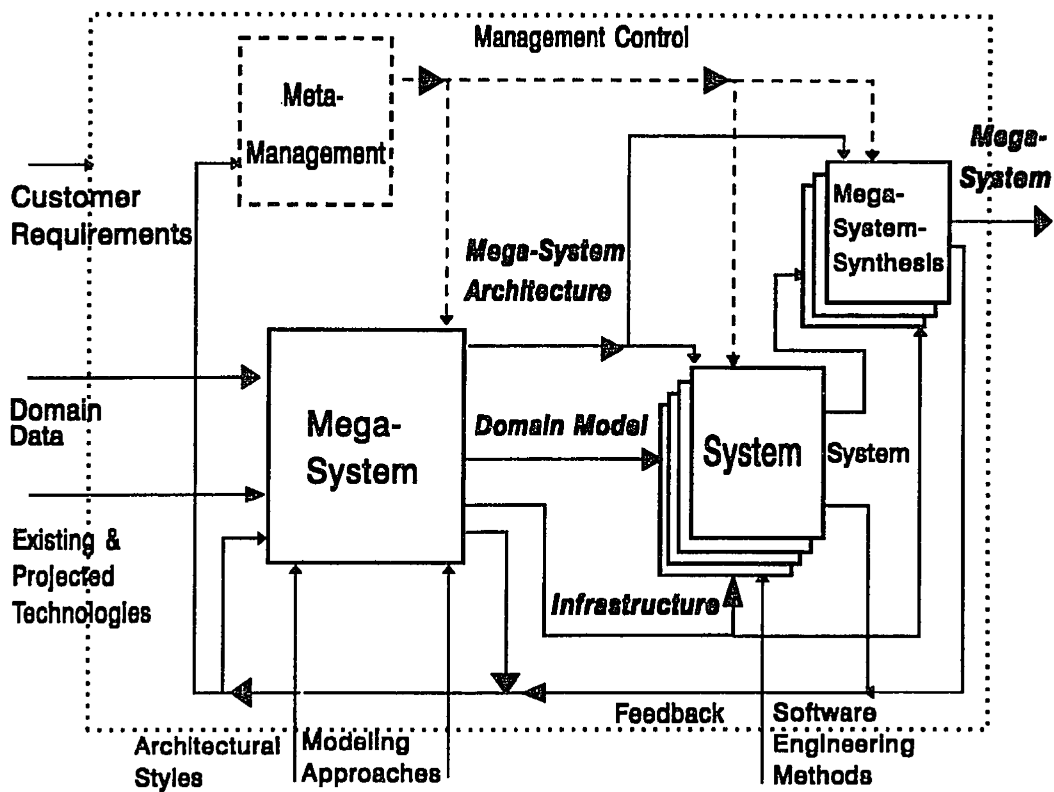

Figure 4.3 MegSDF First Level

4.2.4 Timing

The process of Mega-System development is continuous, persisting as long as systems are developed and maintained in the domain. Therefore, meta-management tasks and Mega-Systems tasks should be active for the life of the Mega-System. Mega-System synthesis should be active according to customers requirements. Systems tasks are active according to the necessities of the process. Multiple system or synthesis tasks may operate concurrently.

Figure 4.4 illustrates a possible schedule for development of a Mega-System. The tasks are drawn as lines over the time axis. After initialization of the Mega-System, meta-management and Mega-System tasks are activated and remain active during the life of the Mega-System. System and synthesis tasks are activated and deactivated according to the actual needs. The Mega-System may integrate systems that were developed before its initialization.

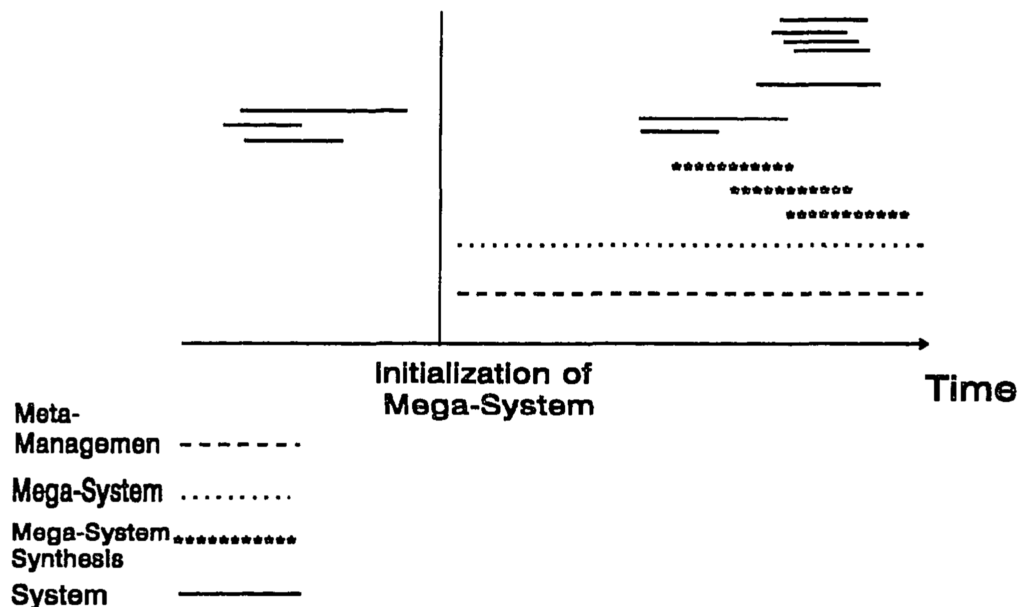

Figure 4.4 A Schedule for Mega-System Development

4.2.5 Sub-Tasks

The essential tasks of the MegSDF process are the Mega-System tasks discussed in section 4.3 and further elaborated in chapter 5, 6 and 7. Meta-management, system and Mega-System tasks are discussed in chapter 8.

4.3 Mega-System Task

The Mega-System task performs the engineering coordination for the process, focusing on general issues and long term objectives. It consists of domain analysis, Mega-System architecture design, and infrastructure acquisition sub-tasks.

Domain analysis provides a general, comprehensive ***domain model*** in order to improve understanding of the problem. The domain model is used to facilitate identifying future requirements, including requirements for integration with other systems. It is also used to balance multiple and ambiguous requirements.

A ***Mega-System architecture*** designs the system in the large. An ***application architecture*** specifies the boundary of the system within the domain and identifies the main parts of the system. The conceptual architecture specifies design and implementation concepts to ensure uniformity and integratability of the constituent systems.

The ***infrastructure acquisition*** provides a unified environment of enabling technologies through an ***infrastructure***, used for all projects that develop systems in the domain.

The interaction of the Mega-System tasks is described below.

4.3.1 Purpose

The purpose of the Mega-Systems tasks is to provide models, concepts, and tools for engineering coordination of the entire process.

4.3.2 Interfaces

Inputs

• Domain Data - Information regarding the domain in which the Mega-System is intended to operate.

• Customers/Users requirements - Requirements of the Customers/Users of the systems.

• Existing and Projected Technologies - Information about extant and projected technologies including enabling technologies and infrastructures. Technologies compatible with the Mega-System attributes are chosen from this input and used to implement the Mega-System.

• Feedback - Engineering information from the system and Mega-System synthesis tasks.

Mechanisms

• Modeling Approaches - Commonly available modeling approaches.

• Architectural Styles - Styles of conceptual architectures for Mega-Systems.

Control Inputs

Management Control - The schedule and milestones assigned to the Mega-System tasks by the meta-management task.

Outputs

• Domain Model - A model of the application domain, defined in section 5.2

• Mega-System Architecture - The architecture of the Mega-System, defined in section 6.2.

• Infrastructure - The chosen infrastructure, defined in section 7.2.

• Feedback - Status and engineering data required by the meta-management for managing the whole process.

4.3.3 Processing

A ***domain analysis*** task defines a domain model based on the domain information. The ***Mega-System architecture design*** task specifies the architecture of the Mega-System, based on the domain model and existing and projected technologies. The **infrastructure acquisition** task selects and acquires a common infrastructure based on the concepts of the conceptual architecture. Feedback from the infrastructure acquisition task is used to improve the Mega-System architecture. Feedback from the Mega-System architecture is used to improve the domain model. Feedback from the system and synthesis tasks is used to improve the engineering coordination tools. All Mega-Systems tasks use customers/users requirements as essential information. Figure 4.5 illustrates the interrelation of the Mega-System tasks.

The process assumes that verification, validation, and quality assurance are done as part of every task or sub-task to ensure effective and efficient engineering coordination tools are provided to the developers of systems in the domain.

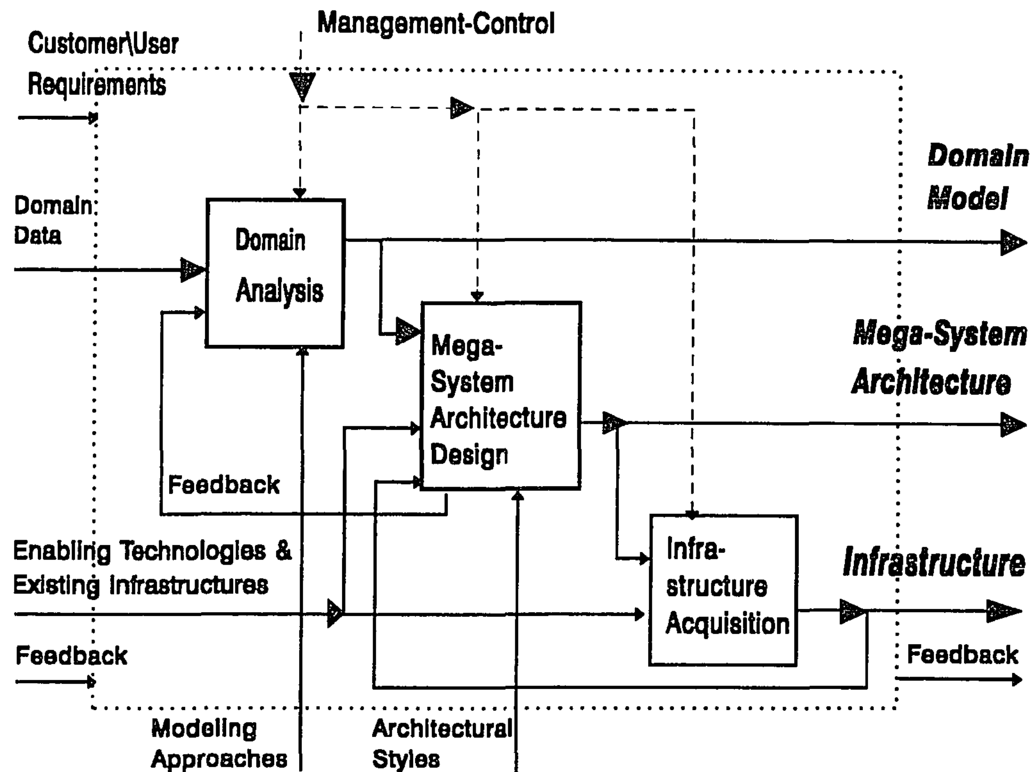

Figure 4.5 Process Diagram for the Mega-System Tasks

4.3.4 Timing

Mega-System tasks provide engineering coordination for all systems developed in the domain. Consequently, these tasks are active for the duration of systems development and maintenance in the domain. The domain analysis task tracks changes in the domain. The infrastructure acquisition task must stabilize the interfaces to the ever evolving technologies. The Mega-System architecture design task translates changes in the domain reflected in the domain model into implementation changes in the application architecture. It also stabilizes the implementation environment by providing common design and implementation concepts.

4.3.5 Sub-Tasks

Domain analysis, Mega-System architecture design, and Infrastructure acquisition are the backbone of our Framework. Each of these tasks is discussed in more detail in one of the following chapters. Chapter 5 describes domain analysis, chapter 6 describes Mega-System architecture design, and chapter 7 describes infrastructure acquisition.

CHAPTER 5

DOMAIN ANALYSIS FOR MEGA-SYSTEMS

Domain analysis in MegSDF is intended to provide a general, universal, comprehensive, non-constructive ***domain model*** to be used as a common basis for understanding of the domain. The Domain model is used by the various system tasks as an essential input for requirement specification. It supports the "pre-planned" approach by modeling the entire domain and not a limited part of it. The domain model is used to improve the understanding of the role of any constituent system and its relationship with its environment (and not as an isolated system). Furthermore, the domain model is used by the Mega-System architecture design task for both conceptual and application architecture design. Unlike domain modeling for software reuse, MegSDF domain model does not include constructive elements.

An application domain is perceived differently by entities with different relationships to the domain. The domain model in MegSDF is built as an integration of ***significant perceptions*** of the domain by its perceivers. Each significant perception representing the phenomena of the domain from the viewpoint of a specific perceiver.

The process of domain modeling in MegSDF includes two phases. In the first phase, a ***domain modeling schema*** (domain schema) consisting of ***element-types*** (modeling primitives) for the domain is defined. In the second phase, the significant perceptions, built using the ***domain-schema***, are integrated into a common domain model.

The process of domain analysis is continuous. Any essential change in the domain, as well as feedback from other tasks, should be evaluated and reflected in the domain model as required.

This chapter defines the domain analysis task. Section 5.1 describes the role of domain analysis in MegSDF, its required characteristics, and contrasts it with current methods of domain and system analysis. Section 5.2 defines MegSDF's domain modeling approach and a technique that structures the modeling process. A process based on the technique is defined in section 5.3. Section 5.4 compares our approach with other methods of system and domain analysis. An example illustrating the domain analysis process concludes the chapter.

5.1 Requirements for Domain Analysis

5.1.1 The Role of Domain Analysis in MegSDF

Domain analysis was identified in chapter 4 as one of the Mega-System tasks. The purpose of domain analysis is to specify a domain model used to support the development of software systems in the analyzed domain. The domain model serves as a common basis for understanding of the domain. It is used as a reference model, thesaurus, or knowledge base, which captures the essential information required to understand the application domain.

Domain analysis is intended to address and rectify the following difficulties in software development: neglect of overall, long term issues; the need to deal with multiple, unstable requirements of customers with different aims and needs; and coordination and communication problems. Table 1 summarizes these objectives (using an inverted sub-table of the problem list table 1.1).

Table 5.1 Difficulties and Problems Addressed by MegSDF Domain Analysis

Difficulties	Caused By	Aspect	Problems
The overall view of the system is neglected	More than one group of developers	Engineering	Current methods do not fit development of more than one system, with multiple and unstable requirements
Multiple requirements	More than one customer		
Unstable requirements	Long life cycle		
General objectives are neglected	More than one system	Management	There is no clear distinction between general, long-term objectives and local, short-term objectives
Coordination and communication problems on a larger scale	More than one developer		
Different aims and needs	More than one customer		
Long term objectives are neglected	Long life cycle		

The domain model serves as a basis for refinement or specialization during the requirement specification phases of the various system tasks (projects) which develop constituent systems. It is an input for the Mega-System architecture design task, to which

it represents the domain. Feedback from the system and Mega-System Architecture design tasks includes recommendations for improvement to and corrections of the domain model. Figure 5.1 illustrates the relationship of the domain model to the other elements of MegSDF.

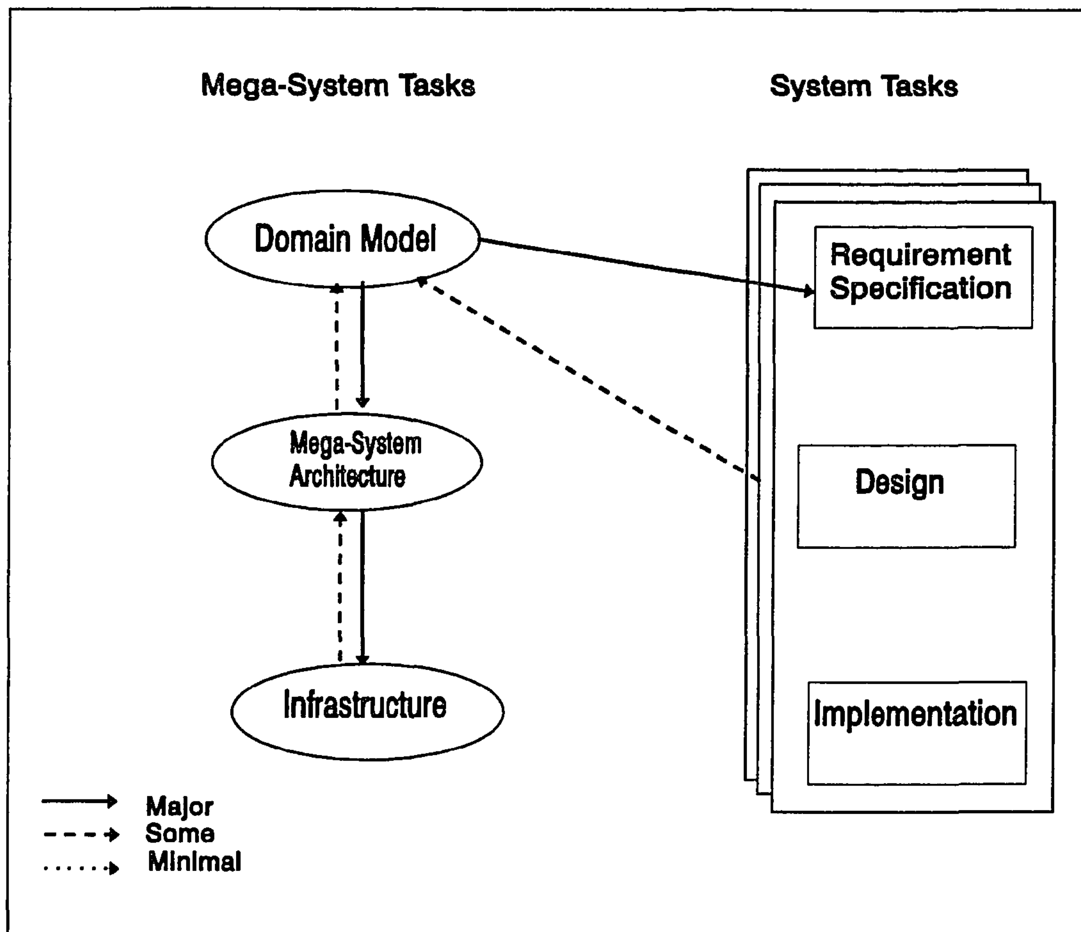

Figure 5.1 The Relationship of the Domain Model

5.1.2 Requirements for MegSDF's Domain Model

Since MegSDF must be general, i.e., applicable to any domain and any type of Mega-System, the process of domain analysis, as part of the framework, must be applicable to

any application domain and to any type of Mega-System. Consequently, the process itself must be both flexible and domain independent.

A domain model is intended to provide a common basis for understanding. To do so, it must be:

- Universal,
- General,
- Comprehensive,
- Nonconstructive, and
- User-friendly.

Universality is required because the model is used by every project developing any system in the domain. For example, a university domain model will be used for the registrar system, the accounting system, as well as the foreign student system. Furthermore, since we are seeking integratable systems, it is essential to identify the relationship of each system to the other parts of its environment. A universal model of the whole application domain will facilitate the development of such integratable systems.

Generality is required because the model is not intended to be used for a specific instance (system) of the domain, but rather as a common model for all systems for the domain. For example, a university domain model represents all universities, not a specific university. As a general model, a university domain model includes such concepts as academic year and terms, but the actual number of terms, their lengths and schedules, vary with the university and so are not represented in the model. A model that fit only a specific instance or a particular system would not provide a common basis for understanding for

all systems in the domain. The analysis of a specific case (instance) is, from this viewpoint, only traditional system analysis. Of course, we must limit the generality of the model to ensure its usability. If the model is too general it will include too many alternatives and become unmanageable. While if it is too abstract, it will lack adequately detailed information. For example, an aircraft carrier domain model should represent all aircraft carriers, not battleships or arbitrary military vessels, but not be restricted to a specific aircraft carrier, e.g., the Enterprise.

Comprehensiveness is required since the model serves as a common basis for understanding, and so must include all the essential kinds of information regarding the domain. The model should include information about the things in the domain, their interactions, concepts. and any useful knowledge.

A domain model should be ***non-constructive***, that is, it should not concentrate on the constructive aspects: design and implementation. A conceptual model for an application domain without constructive elements provides a broader basis for systems implementation. It also improves the reusability of the domain model, because constructive elements usually belong to the solution domain and tend to restrict a model to a specific solution, hiding the essential concepts of the domain. We propose that constructive aspects be dealt with separately, during Mega-System architecture design and infrastructure acquisition.

Finally, the domain model must be ***user-friendly***, since it is intended for use by system analysts, architecture designers, etc., and not only by software systems, e.g.,

application generators. Machine readability is required to support the model by CASE tools, but is not an intrinsic element of the technique.

5.1.3 Contrast with Domain Analysis in Reusability and System Analysis

The notion of domain analysis in MegSDF differs from its use in software reusability. The concept of domain analysis for software reuse was introduced by [NEIG 81] as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain." Similarly, [PRIE 90] defines domain analysis as "a process where information used in developing software systems is identified, captured, structured and organized for further reuse." In both cases, domain analysis is used only to identify reusable components.

Arango and Prieto-Diaz [ARAN 91] recommend representing the specification and implementation concepts for reuse in a domain model which includes information on at least three aspects of a problem domain:

- Concepts which allow the specification of systems in the domain,
- Plans which describe how to map specifications to code, and
- Rationales for the specification concepts, their inter-relationships, and their relationship to implementation plans.

They also recommend dividing domain analysis into conceptual and constructive analyses. The conceptual analysis identifies the information required to specify systems in the domain. The constructive analysis identifies information required to implement systems for the domain. Additionally, they suggest specifying and implementing an infrastructure

that facilitates software reuse. Libraries of programs or software archives ([MITT 87], [ROSS 87a]) are examples of such reuse infrastructures.

Domain analysis as described by [ARAN 91], [ISCO 91], [NEIG 81], [PRIE 90], [PRIE 91a, b, c], [THAY 90] primarily addresses software reuse for families of similar systems. In contrast MegSDF's domain analysis is intended for systems of systems, consisting of dis-similar systems of different types. Furthermore, it provides a conceptual model only, which is used primarily as a basis for future integration of systems in the domain, and not only to support code or program generation.

Our approach to domain analysis might be considered as a generalization of system analysis [BOOC 91], [COAD 91a], [RUMB 91], [YOUR 89] or conceptual analysis [YEH 80], but its scope is much broader. It is intended for systems of system and for families of systems, with long life cycles, not only for instances of systems.

5.2 Domain Modeling

This section introduces the underlying modeling approach. The model is based on phenomena, different perceptions of the domain, and significant aspects of phenomena as discussed in section 5.2.1. Section 5.3.2 describes a technique that structures the model. Section 5.2.3 summarizes the concepts of domain modeling.

5.2.1 The Content of the Model

5.2.1.1 Phenomena

A domain model is a universal, general, comprehensive, non-constructive model of an application domain. The domain model abstracts the ***phenomena*** of the domain, and omits details about specific instances of the domain. For example, a domain model for a university might abstract student, department, registration, enrollment in a course, a policy for student acceptance, and the difference between Mathematics and the Applied Mathematics departments. We call the abstractions of the domain phenomena in the domain model ***"elements."*** The characteristics of a phenomenon in the domain are represented as ***attributes*** of an element in the domain model. For example, the attributes of an element representing a student might be: name, address, student-id, and Grade Point Average (GPA).

Since the domain model must be comprehensive, it must represent phenomena belonging both to the static structure of the domain, e.g., objects and relations, as well as the dynamic interactions of the domain, e.g., processes and events (c.f. also [RUMB 91]).

The ***static structure*** of a domain includes objects (entities) and their relationships. In the object oriented approaches, objects of the domain with similar characteristics are grouped into object-classes [GELL 91]. Objects relate to other objects in various ways, e.g., by generalization, specialization, aggregation, or association. We view these relationships themselves as phenomena belonging to the static structure of the domain.

The ***dynamic interactions*** of the domain include behavior patterns of phenomena. The object-oriented methods specify operations that can be applied to instances of a given object-class [GELL 91], but we also want to represent processes that may involve more than one object, relationship, or activity. Using processes, it is possible to represent the methods and techniques used to solve problems in the domain.

A process is a set of activities operating on or executed by various phenomena in the domain, the results of these activities, and their sequencing. An example of a process in the university domain is registration. In this process, a student selects courses, receives an approval from his advisor, registers, and is billed. We also propose representing events and states transitions in the domain model as part of the dynamic structure. An example of an event is a failure in an exam. An example of a state transition could be a faculty changing rank from assistant to associate professor.

A general model will also include a variety of other kinds of qualitative and quantitative information and statistical information such as averages and maximums. It might include rationales and constraints.

5.2.1.2 Different Perceptions

A domain is perceived differently by entities which have different relationships, roles, or concerns with the domain [THIM 92]. For example, a student and a registrar have different perceptions of the university domain. These differing perceptions arise from the different relationships of the perceivers to the domain and may include: different groups of elements; the same elements under different names or with different attributes and

roles. To achieve universality and comprehensiveness, we propose building the domain model by integrating multiple domain perceptions.

First, entities with a significant perception of the domain are identified. These entities may influence the domain or be influenced by it. For example, in the university domain, we might identify faculty, registrar, board of education, student, and staff, as entities which have significant perceptions. After identifying these entities, it is necessary to build a perception for each of them. Thus, a perception is a representation of the domain as perceived by an entity who has a significant role in or concern with the domain. Phenomena are represented in a perception as perception-elements. For example, perception-elements for a faculty's perception of the university domain might be student, course, department. All the perception-elements for a specific phenomenon, perceived by different significant perceivers, will finally be merged into one integrated element in the domain model. For example, the registrar's student-perception-element, the faculty's student-perception-element, and the student's student-perception-element are integrated into the final student-element in the domain model.

Figure 5.2 illustrates the integration of several perceptions into a domain model. A domain with phenomena X, Y, Z, U is perceived by some significant Perceivers. Perception-1 of perceiver-1 includes perception-elements X'_1, Y'_1, Z'_1, U'_1. Perception-2 of perceiver-2 includes perception-elements X'_2, Y'_2, and U'_2. The domain model includes elements X', Y', Z', U' where element X' integrates both X'_1 and X'_2, element Y' integrates both Y'_1 and Y'_2, etc.

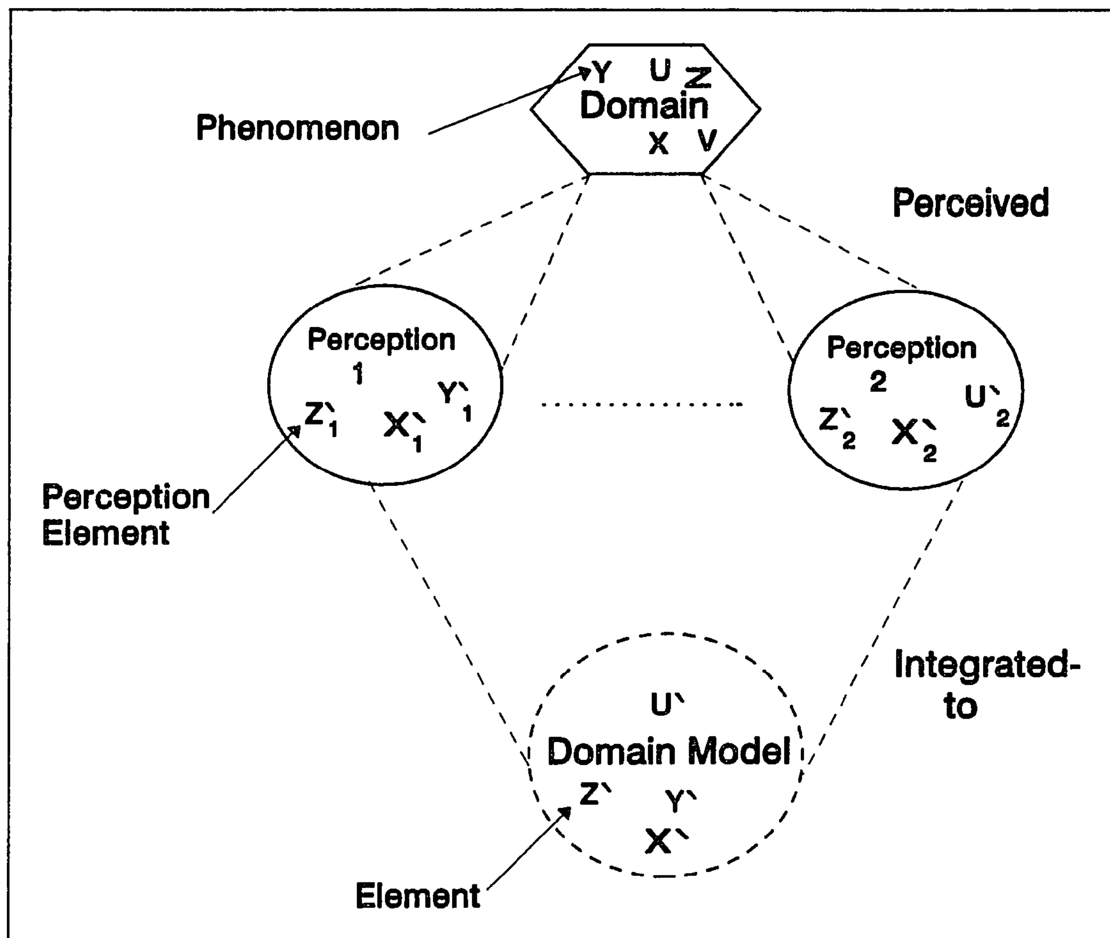

Figure 5.2 A Domain Model as an Integration of Multiple Perceptions

It is important to note that perceptions as described here generalize the view concept in database systems [ELMA 89], [SHET 90], [ULLM 88]. Views in databases are used to specify parts of a database, to create virtual objects from real objects, and to restrict the access of users to different parts of the system [ULLM 88]. We, however, define a perception as a representation of a domain as perceived by an entity with a significant relationship to the domain.

5.2.1.3 Aspects of Phenomena

Any phenomenon has different "aspects": physical, structural, dynamic, static, etc. Physical aspects refer to the physical properties of phenomena: dimensions, weight, composition. Structural aspects pertain to the manner in which a phenomenon is organized, or related to other phenomena, e.g., the components of the phenomena, or membership. Dynamic aspects describe changes of the phenomenon, e.g., the frequency of a change, the originator of a change, etc. An aspect usually deals with a specific set of attributes. Aspects are also discussed in [WIMM 92] who calls an aspect a view.

The significance of aspects is domain specific. For example, in the CAD domain, physical aspects are more important than legal aspects, which on the other hand might be more significant in the banking domain. Since a perceiver is often interested in a subset of domain aspects, the significant aspects for different perceivers may be disjoint or they may overlap. For example, the significant aspects of a faculty perceiver in the university domain might be structural, static, and dynamic aspects, while for the physical plant manager they might be the structural, static, and physical.

5.2.2 Structuring the Model

5.2.2.1 Domain-Schema

The required universality and comprehensiveness of a domain model implies that the domain model must be able to handle a large amount of information. In order to manage this information, support the modeling technique, and uniformize the various perceptions,

we propose using a domain modeling schema. We call this the ***domain-schema***. The domain-schema is used to define the modeling primitives which will be used later to represent the phenomena of the domain as elements. We call the modeling primitives ***element-types***. A similar idea is suggested in [WIMM 92].

A domain-schema consists of element-types used as modeling primitives to represent a group of elements with similar attributes. A group of elements that might be represented by using the same element-type is called an element-class. Possible element-classes are object, relationship, event, process, etc. The object-element-class, for example, includes all object-classes that belong to the domain, where each object-class represents a group of objects in the domain with similar attributes.

In the domain model, all elements that belong to the same element-class are represented using an element-type that defines a possible set of attributes for the elements of the element-class. The element-type acts as a template that is filled-in with actual attributes for each element. Since every phenomenon has multiple aspects, we divide the attributes into groups based on these aspects. We call these groups ***element-aspects***. Thus, an element-type is a union of element-aspects, where each element-aspect includes the attributes of one aspect for a specific element-class.

The domain-schema can be considered as a meta-schema, and its element-classes and element-types as meta-classes and meta-types. Element-types are used to describe classes of elements, e.g., object-classes, processes, not one element that represents a class of instances of the domain with similar attributes, e.g., student or registration, nor instances of the domain, i.e., J. Smith or the CIS department. They do not describe the

attributes of a specific element, e.g., student or faculty; they describe the attributes of an element-class, e.g., object-class, relation, event. Beyond the element-types such as objects and relations, our schema might also include other element-types, e.g., processes, constraints, or special domain-dependent element-types. The attributes of the element-class might be considered as meta-attributes since they are used to describe a set of possible attributes of the elements that belong to the same element-class.

It is important to differentiate the domain schema and schemas of databases. Schemas of databases describe the structure of the database and represent elements of the problem space itself, e.g., student, faculty, department, etc. Domain schemas defines the modeling primitives to be used for modeling the domain: objects, relationships, events, etc.

5.2.2.2 Using the Domain-Schema

The domain-schema specifies a set of modeling primitives. It provides flexible guidelines and a checklist for domain analysts. The element attributes are optional and attributes can be added when required. The domain schema simplifies perception integration, since the element-types, aspects, and attributes provide a structured, organized basis for integration.

5.2.2.3 Dimensions

[RUMB 91] suggests modeling a system from three viewpoints: the object model, the dynamic model, and the functional model. We also recommend dividing the domain model and its elements into orthogonal and interrelated parts considering each part as a

dimension of the domain model. In order to implement this idea, we specify domain-schema dimensions as groups of inter-related element-types. Each group is used for modeling a dimension of the domain model. The number of dimensions and their content depend on both the modeling approach and the domain. For example, a model based on the Entity-Relationship (ER) approach includes only a data dimension with the entities and relationship as the element-types.

Dimensions, aspects, database views, and perceptions are different. The aspects in a domain schema deal with attributes of phenomena and group them into sets. The dimensions of the domain model, are groups of interrelated phenomena used to simplify modeling by dividing the model into interrelated parts. Views in databases are used to define virtual objects and restrict user access to parts of the data; this is close to the perception concept in our approach. Perceptions are used to model the domain from a specific point of view and include a sub-set of the phenomena and aspects of the domain.

5.2.2.4 Element-Types

An element-type is defined in the schema by a set of attributes divided into aspects and represented by a frame-template (see Table 5.2). Each frame includes actual aspects and their attributes. Composite attributes consisting of other attributes are also allowed and are drawn as split cells, e.g., attribute 21. Multi-valued attributes, which may appear more than once, are designated by a star (*); attributes that appear at least once are designated by a plus (+).

Defining a domain-schema requires identifying element-classes and then defining their element-types with appropriate sets of attributes. The number and kind of element-classes and the content of their element-types depend on the modeling approach, the application domain, and the significant aspects. Similar templates, but with a restricted set of element-types and no explicit division of the attributes into aspects appear in [BOOC 91].

Table 5.2 A Template for Element-Types

<table>
<tr><th colspan="6">Element-Type</th></tr>
<tr><th>Aspect 1</th><th colspan="2">Aspect 2</th><th>Aspect 3</th><th>..</th><th>Aspect N</th></tr>
<tr><td rowspan="3">Attr. 11:
Type 11</td><td rowspan="3">Composite-
Attribute
21</td><td>Attr. 211:
Type 211</td><td rowspan="3">Multi-valued
attribute31 *
:
Type 31</td><td rowspan="3"></td><td rowspan="3">Multi-
valued
Attr. N1
(Not
Empty)
+:
Type N1</td></tr>
<tr><td>Attr. 212:
Type 212</td></tr>
<tr><td>Attr. 213:
Type 213</td></tr>
<tr><td>Attr. 12:
Type 12</td><td colspan="2">Attribute 22:
Type 22</td><td></td><td></td><td>Attr. N2:
Type N2</td></tr>
<tr><td>Attr. 13:
Type 13</td><td colspan="2"></td><td></td><td></td><td>.
.</td></tr>
<tr><td></td><td colspan="2"></td><td></td><td></td><td>Attr. Nk:
Type Nk</td></tr>
</table>

Table 5.3 is an example of an object-element-type. This element-type might be used for representation of objects in a domain model. If the actual aspects in the analyzed domain are physical, structural, static, dynamic, legal, and logical, the template includes only attributes of these aspects. The physical aspect includes physical characteristics of

objects, e.g., "dimension", "weight", "color", etc. The structural aspect includes the "generalizes", "specializes", "aggregates" attributes to enable inheritance and aggregation of objects into composite objects. An object can be a generalization of several objects and therefore the "generalizes" attribute is a multi-valued attribute. Objects have their life cycle. It is possible to describe the objects' life cycle by state-diagrams [SHLA 92]. These diagrams include the various states an object might have and the transitions between them. Accordingly, the static aspect might include a state attribute that represents the actual state of the object. The dynamic attributes of an object might include a reference to a state transition diagram that describes the transitions between the various states in which an object can be. "Method" is a multiple attribute that represents the methods that can be applied on instances of the object class. Similarly, the other aspects include a list of relevant attributes.

It is important to understand that this is an example only of an object-element-type. A process-element-type or another element-type will use other set of attributes. Furthermore, since, schemas are domain dependent, it is possible that for other domains the object-element-types will have different aspects and sets of attributes. Like the object-element-type, a domain schema might include relationship-element type, process-element-type, event-element-type, etc.

Table 5.3 An Example of an Object-Element-type

Object					
Physical	**Structural**	**Static**	**Dynamic**	**Legal**	**Logical**
Dimension *: Numeric	Generalizes *: Object	State: State	State-diagram: Diagram	Status: Text	ID: Identifier
Weight: Numeric	Specializes *: Object		Method *: Method		purpose: Text
Velocity: Numeric	Aggregates *: Object				Value: Numeric
Color: Text	Part-of *: Object				Status: Text
Material: Text					Role: Text
Tempera-ture: Numeric					

5.2.2.5 Modeling with Element-Types

The various phenomena of the domain are represented using the appropriate element-type. For each aspect, the relevant attributes are specified. Each attribute is now described by a split cell. The upper part of each cell includes the element-type's attribute. The lower part includes the actual attribute of the element. For example, the dimension attribute of an object-class might include only height and width. Irrelevant or unused attributes are designated by (--).

Using the object-element-type of Table 5.3, it is possible to uniformly represent the various object-classes in a domain. Each object-class in the domain is defined by using

the template. A representation of a Building object-element in a university domain appears in Table 5.4. In this case, the Weight, Velocity, and Temperature attributes of the physical aspect, the Generalizes of the structural aspects, and other attributes are not used and therefore are designated by "--". The Method attribute includes the Assign method that assigns a Building to a Department. The Aggregates attribute includes all the object-class-elements that are aggregated by a Building: Floor, Hall, Room, and Elevator. Similarly, other attributes might be examined and specialized according to the actual element.

Table 5.4 An Example of a Building-Object-Element

Building					
Physical	**Structural**	**Static**	**Dynamic**	**Legal**	**logical**
Dimension *: Numeric	Generalizes*: Object	State : State	State-Diagram: state-diagram	Status: status	ID: Alpha-numeric
Height,Width, Length	(--)	(--)	(--)	(Approved, Restricted)	Building-ID
Weight: Numeric	Specializes*: objects		Method *		purpose*: Enumerate
(--)	(--)		Assign		(Teaching, Administration, Sports,Storage, Utilities)
Velocity: Numeric	Aggregates*: object				Value: Numeric
(--)	Floor, Elevator, Room				(--)
Color: Enumerate	Part-of *: object				Status
(--)	Campus				(--)
Material					Role: Alphanumeric
(Wood,Blocks)					(--)
Temperature: Numeric					
(--)					

5.2.2.6 The Perception-Schema

The significance of various domain aspects may vary with each perceiver. It is also possible that a perceiver is interested only in a limited set of element-classes. To simplify modeling we suggest using a perception-schema for each perceiver. A perception-schema

is derived from the domain-schema by selecting the perceiver's actual element-classes and restricting the schema to the significant aspects for that perceiver. It is, also, possible to limit the attributes of an element-aspect to include only a subset of attributes of the element-aspect in the perception-element. Thus, a perception-schema is a sub-schema of the domain-schema determined by the set of element-types, the set of element-aspects of each element-type, and the set of attributes in an element-aspect.

Table 5.5 includes an example of perception-object-element-type. This perception-element-type is derived from the object-element-type that appears in Table 5.3 and fits a perceiver whose actual aspects are static, structural and physical. Accordingly, Table 5.6 includes an example of an object-perception-element for a building. The building-perception-element includes only the attributes of the significant aspects.

If each domain aspect belongs only to one perceiver, the perception-schemas are aspect disjoint. However, it is more likely that the same aspect or the same group of aspects appears in more than one perception-schema. Perceivers who are interested in the same aspects and the same set of element-classes can use the same perception-schema and yet build different perceptions.

Table 5.5 Object-Perception-Element-Type

Object-Perception-Type		
Physical	**Structural**	**Static**
Dimension *: Numeric	Generalizes*: Object	State: State
Weight : Numeric	Specializes*: Object	
Velocity: Numeric	Aggregates*: Object	
Color: Text	Part-of *: Object	
Material: Text		
State: Enumerate		
Temperature: Numeric		

Table 5.6 Building-Perception-Element

Building-Perception-Element		
Physical	**Structural**	**Static**
Dimension*: Numeric	Generalizes*: Object	State:
Height, Width, Length	(--)	
Weight	Specializes*:Object	
(--)	(--)	
Velocity	Aggregates *: Object	
(--)	Floor, Elevator, Room	
Color	Part-of *: Object	
(--)	Campus	
Material		
(Wood, Blocks)		
State		
(--)		
Temperature		
(--)		

5.2.2.7 The Structured Perception

Using the perception-schema as a guideline, a model of the domain is built for each perceiver. We propose that domain experts will build these perceptions supported by domain analysts.

The relevant phenomena of the domain for a given perception are identified. Each phenomenon is classified into one of the element-classes. Using the appropriate perception-element-type, the different attributes of the element are specified. The processes that build the perceptions can be done concurrently by different groups. However, since the domain-schema is used for derivation of all perception-schemas, the resulting perceptions will be both structured and coordinated.

5.2.2.8 The Integrated Model

The various perceptions are finally integrated into a domain model. We first determine which perception-elements of different perceptions represent the same phenomenon. Later, all the perception-elements for a specific phenomenon are integrated into a unified element. The perception integration process is based on the element-aspects. When different perceivers are interested in different sets of aspects, the final integrated element is the union of the various element-aspects. When an aspect is relevant to more than one perceiver, attributes in the appropriate element-aspect are compared; conflicts in element-type, names, attributes, roles, etc., are resolved; and a unified element-aspect is derived. When conflicts cannot be resolved, the conflicting versions are all incorporated into the element.

Figure 5.3 illustrates this integration process. A domain with phenomena P_1, P_2, ... , P_m is perceived by N perceivers. A perception for each perceiver is built (Perception 1, ..., Perception N). Each perception consists of perception elements that represent the significant phenomena for the perceiver. PE_{ij} denotes the perception element of perceiver i for phenomenon j. The domain model integrates all the perception-elements for phenomenon j: $\{PE_{ij} \mid i = 1 \text{ to } n$, where phenomenon j is significant to perceiver i$\}$ into an element E_j.

Perception \ Phenomena	P1	P2	P3	Pm
Perception 1	PE_{11}		PE_{13}	PE_{1m}
Perception 2	PE_{21}	PE_{22}		PE_{2m}
Perception 3	PE_{13}		PE_{33}	
Perception N	PE_{N1}			PE_{Nm}
Domain Model (Elements)	E1	E2	E3	Em

Figure 5.3 The Integration of Perception-Elements into Elements

The process is similar to integrating views or schemas of databases [BATI 86], [SHET 88], [SHET 90], [GELL 91], [GELL 92]. However, schema integration [SHET 88], [GELL 91], [GELL 92] is done on static structure elements (objects and their

relations) only, while here integration is done for elements of all dimensions of the model including processes, events, etc.

Figure 5.4 illustrates the relationship between the concepts of domain analysis. A domain-schema consists of dimensions and element-types (denoted by lower case letters) and aspects (denoted by numbers). For simplicity only one dimension is drawn, and not all identifiers are marked. Each element-type (denoted by a dashed ellipse) integrates several element-aspects. The schema includes three element-types: ET-a, ET-b, and ET-d. X_{ij} denotes an element-aspect where i identifies the element-type and j the aspect.

Based on the domain-schema, perception-schemas with Perception-Element-Types (PET) are derived. Perception-schemas include only subsets of the element-types and element-aspects. For example, perception-schema-Y includes only two perception-element-types, PET_{Ya} and PET_{Yd}, corresponding to ET-a and ET-d with element-aspects X_{a1}, X_{a2}, X_{d1}, and X_{d4} that fit the actual aspects of the perception, i.e., aspect-1, aspect-2, and aspect-4. Similarly, Perception-Z includes two perceptions-element-types PET_{Za} and PET_{Zb} that correspond to ET-a and ET-b with X_{a1}, X_{a3}, and X_{b4} as element-aspects.

Perceptions with perception-elements (PEs, denoted as filled ellipses) are built using the perception-schemas. Perception-Y includes three perception-elements of PET_{Ya} (PE_{Ya-i}, PE_{Ya-j}, and PE_{Ya-k}) and two perception-elements of PET_{Yd} (PE_{Yd-l} and PE_{Yd-m}). Perception-Z includes three perception-elements of PET_{Za} (PE_{Za-i}, PE_{Za-j}, and PE_{Za-k}) and one perception-element of PET_{Yb} (PE_{Zb-n}). These perceptions are finally integrated into a domain model that consists of three elements of ET-a (E_{a-i}, E_{a-j}, E_{a-k}), one element of ET-b (E_{b-n}), and two elements of ET-d (E_{d-l}, E_{d-m}).

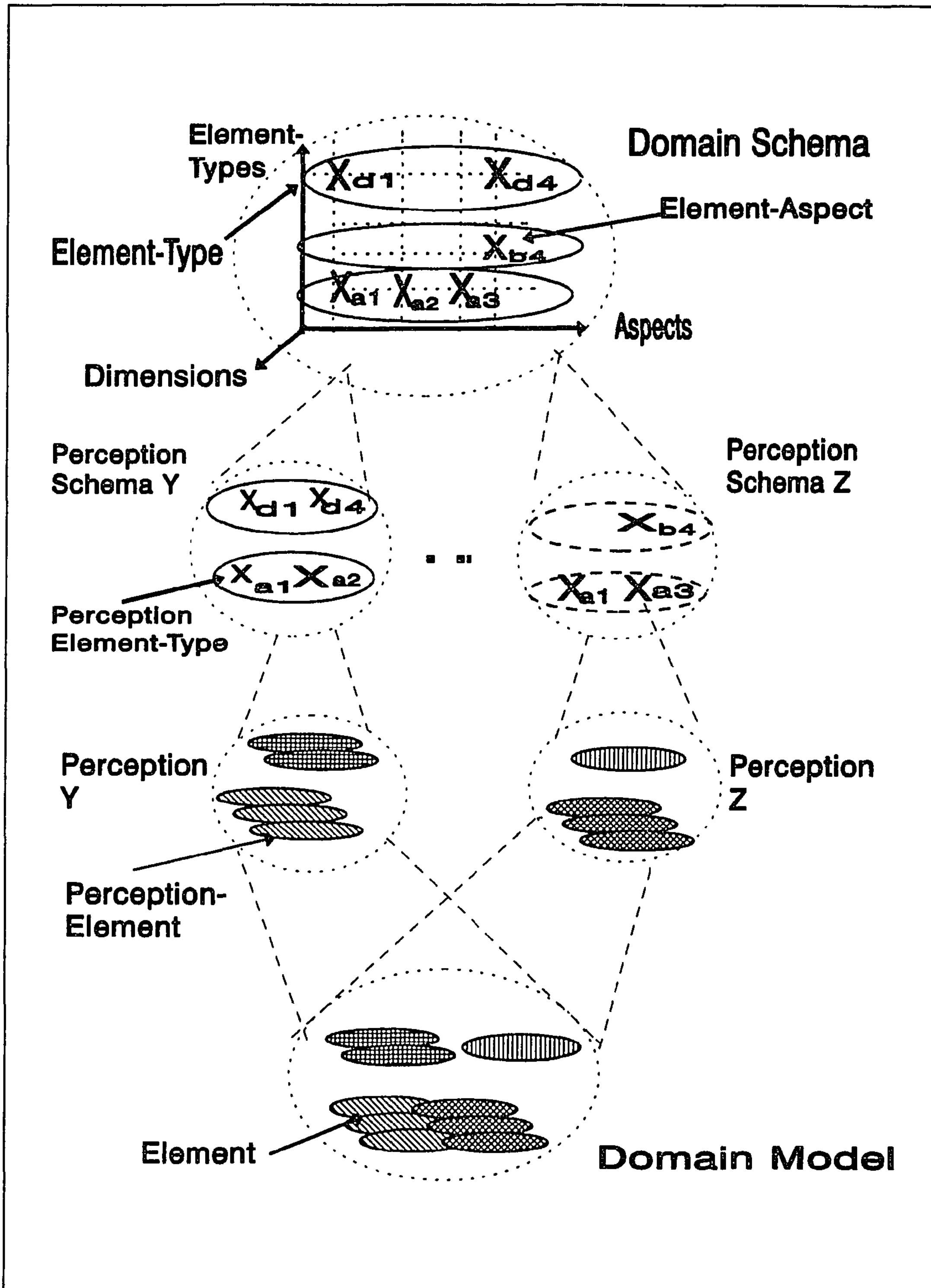

Figure 5.4 The Components of Domain Analysis

5.2.3 Definitions of Domain-Analysis Concepts

This section summarizes the main concepts of MegSDF's domain modeling. The definitions are given in a top down fashion for ease of understanding.

Application Domain

An application domain (domain) D is a comprehensive, internally coherent, relatively self-contained field or area of action, business, research, etc., supported by software systems. An application domain D consists of phenomena $\{P_1, P_2, ..., P_n\}$. For example, a university, banking, or military vessels could be considered as application domains.

Phenomenon

A phenomenon P in an application domain is a concept that abstractly represents instances of a thing, activity, relations, constraints in the domain. For example, students, registration, or acceptance policy are phenomena in the university domain.

Element

An element E is a representation of a phenomenon P of a domain D in a domain model M. The element represents the characteristics of the phenomenon as a set of attributes. For example, a university domain model might include student, faculty, graduation, and registration elements.

Domain Model

A domain model M is a universal, general, comprehensive, non-constructive representation of an application domain D. The model consists of a set of elements $\{E_1, E_2, ..., E_n\}$ which represents the various phenomena $\{P_1, P_2, ..., P_n\}$ of the domain.

Element-Class

An Element-Class C is a set of elements with certain significant similarities. For example, the elements student, faculty, and department belong to the object-element-class, while the elements graduation and registration belong to the process-element-class.

Dimension

A dimension is a group of interrelated elements, typically belonging to a restricted set of element-classes, used to describe the domain from a significant viewpoint. The dimensions are interdependent because the same element may appear in more than one dimension. The actual dimensions of the model depend on the modeling approach used. For example, a domain model based on the [RUMB 91] approach might include structural, dynamic, and functional dimensions, while a domain model based on the ER approach would include only the data dimension.

Attribute

An attribute A is characteristic of an Element-Class C that defines a mapping A from all elements of C into a set of values V, i.e., A: C -> V where V might be a set of integers, real numbers, characters, etc. Each element in C is mapped to either a value or a set of values in V. An attribute is defined by its name and type. The type defines the set V into which the element-class is mapped. Examples of attributes are name, weight, status, object, event, method, etc.

Aspect

Aspects are used to group and organize the attributes of elements in a domain model. Possible aspects are physical, logical, legal and structural. The aspects that are significant to a domain model depend on the application domain.

Element-Aspect

An element-aspect is a set of attributes $F=\{A_1, A_2, .., A_l\}$ used to describe the characteristics of an element-class with respect to a specific aspect. For example, the physical-element-aspect of the object-element-class may include: weight, dimension, color, position, etc.

Element-Type

An element-type is an organized set of possible attributes of an element-class. It is used to describe elements with similar characteristics. The set is organized into element-aspects. An element-type is the union of all element-aspects possible for a specific element-class in a domain. For example, the element-type for the process-element-class includes static and dynamic element-aspects.

Domain-Schema-Dimension

A domain-schema-dimension is a set of element-types used to describe a dimension of the domain model. For example, the data dimension of a domain schema includes object and relation element-types.

Domain-Schema

A domain-schema S is a set of element-types used as modeling primitives. It is dependent on both the domain and the modeling approach, and is organized into domain-schema-

dimensions, in order to simplify the modeling process. For example, a domain schema based on the ER approach includes entity-element-class and relationship-element-class in the data dimension.

Perceiver

A perceiver is either an entity that has any concern with, essential influence on, or which is influenced by the domain. A perceiver has a distinct perception of the domain. Typically, a perceiver is interested only in a subset of the domain phenomena and their aspects.

Perception

A perception is a set of perception-elements representing a subset of the phenomena and the aspects of a domain D as perceived from the viewpoint of a given perceiver.

Perception-Element-Type

A Perception-Element-Type (PET) is a set of element-aspects for a specific element-class for a specific perception. Thus, a PET is an element-type with a restricted set of aspects and attributes. A perception-element-type is derived from an Element-Type (ET) depending on the actual aspects the perceiver is interested in.

Perception-Schema

A Perception-Schema PS is a set of Perception-Element-Types $\{PET_1, PET_2, .., PET_k\}$ that addresses the concerns of a specific perceiver in the domain. A Perception-schema is derived from the domain-schema by specifying the relevant perception-element-types and aspects for the perceiver. A perception-schema is used by the perceiver to define a perception.

Perception-Element

A perception-element is a representation of a phenomenon P in a domain as perceived by a perceiver V. A perception element is modeled as an instantiation of a corresponding Perception-Element-Type PET. PE=Instantiation of (PET).

Using these definitions, we can now redefine the terms element and domain model.

Domain-Model

A domain model M integrates perceptions $\{P_1, P_2, ..., P_n\}$ into a set of elements $M=\{E_1, E_2, ..., E_l\}$.

Element

An element E is the representation of a domain phenomenon P in a domain model M. It integrates the appropriate element-perceptions of the phenomenon.

5.3 The Domain Analysis Process

The domain analysis task is defined using the format described in chapter 4.1.

5.3.1 Purpose

The purpose of the domain analysis process is to provide a universal, general, comprehensive, non-constructive model of the domain.

5.3.2 Interfaces

Inputs

• Domain Data - Information regarding the domain in which the Mega-System is intended to operate.

• Customers/Users requirements - Requirements of the Customers/Users of the systems.

• Feedback - Feedback from the system and Mega-System architecture design tasks including recommendations for improvements and corrections to the domain model.

• Modeling Approaches - Commonly available modeling approaches that might be used as a basis for the domain modeling.

Control Inputs

• Management Control - The schedule and milestones to the domain analysis task assigned by the meta-management task.

Outputs

• Domain Model - A domain model of the application domain, defined in section 5.2.

• Feedback - Feedback from the domain analysis task to the meta-management task.

5.3.3 Processing

Based on the domain identification, a domain schema is defined and significant perceivers are identified. For each perceiver, a perception-schema is derived and then used in building his perception. All perceptions, finally, are integrated into a domain model. Figure 5.5 illustrates this process. The following algorithm summarizes the previous discussion.

1. Identify the domain

2. Define a domain-schema

3. Identify significant perceptions of the domain.

4. For each perceiver

    4.1 Derive a perception-schema

    4.2 Build a perception of the domain

5. Integrate the various perceptions.

The algorithm presumes that verification, validation, and quality assurance are done as part of every task or sub-task to ensure the model accurately describes the application domain.

5.3.4 Timing

Since domains evolve, domain analysis must be a continuous activity. To maintain the effectiveness and usability of the model, essential changes in the domain as well as feedback from the various projects should be evaluated and reflected in the domain model, as required. The process should be active as long as the Mega-System is being developed and maintained.

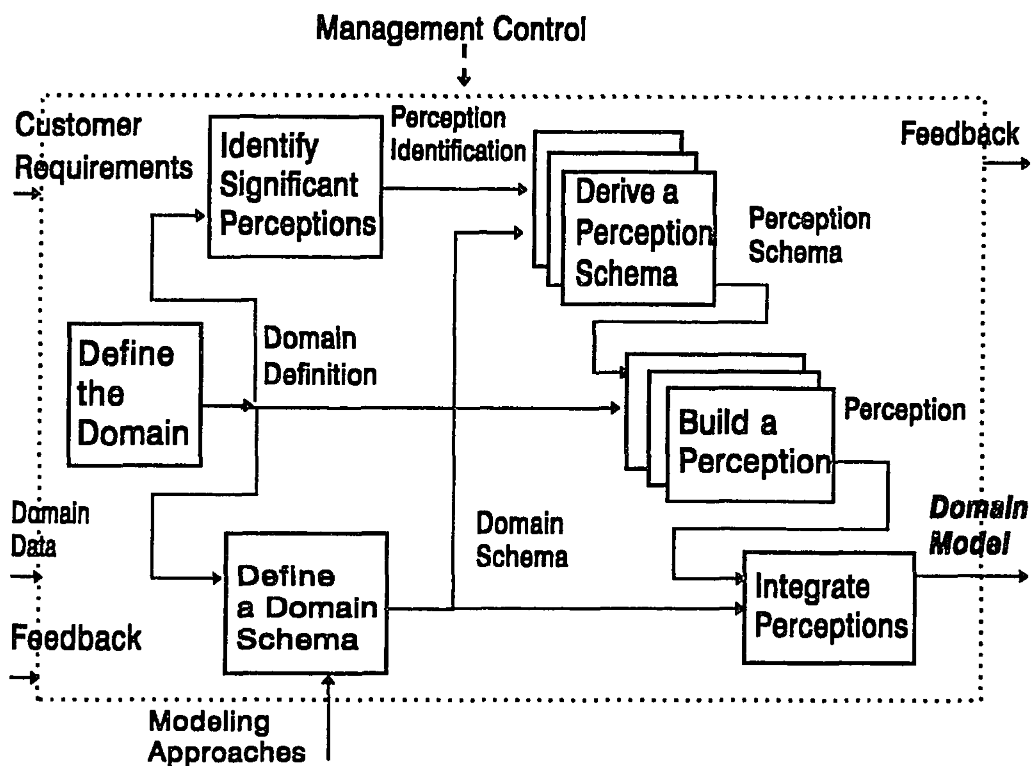

Figure 5.5 A Process Diagram for Domain Analysis

5.3.5 Sub-Tasks

5.3.5.1 Identify the Domain

The first sub-task is to identify the domain to be modeled. Application domains are interrelated, so it is necessary to specify what domain is being modeled and how general the model will be. This identification includes preliminary definition of the domain boundary, which will then be further refined and detailed by the other tasks of the process.

5.3.5.2 Define a Domain-Schema

This task defines the domain-schema for modeling the domain. After choosing a suitable modeling approach that fits the needs of the domain, the dimensions and element-classes are defined. An element-type is then specified for each element-class.

We do not restrict this process to a specific method or modeling approach. Schemas and element-types are intended to organize and coordinate the modeling process. It is possible to define a schema that fits the modeling method and the analyzed domain. Any approach to modeling, e.g., the ER or the object-oriented approach, can be enhanced by the domain-schema and be used to model the domain if appropriate. The definition of the domain schema requires the following activities:

1. Specify relevant aspects for the domain
2. Choose an appropriate modeling approach for the domain
3. Specify dimensions for the modeling
4. For each dimension
    - 4.1 Specify element-classes
    - 4.2 For each element-class define an element-type as follows:
        - 4.2.1 For each relevant aspect
            - 4.2.1.1 Define an element-aspect

5.3.5.3 Identify Significant Perceptions

This task includes identification of the significant perceptions of the domain and the perceivers that might best represent these perceptions. Perceivers are entities, either inside

or outside the domain, who have any concern with, influence on, or which are influenced by the domain. Since a domain might be perceived in many ways, it is essential to identify the significant perceptions.

5.3.5.4 Derive a Perception-Schema

To simplify the process of building a perception, a perception-schema is defined for each perceiver. A perception-schema is a sub-schema of the domain-schema that includes a subset of the element-types and is restricted to a subset of the significant aspects. If required, a perception-schema might include only a subset of the attributes of the element-aspects. The perception-schema is later used to build the perception. Deriving a perception requires:

1. Specify the relevant aspects for the perceiver.

2. Specify relevant element-classes.

2. For each relevant element-class

    2.1 Specify relevant element-aspects

    2.2 For each relevant element-aspect

        2.2.1 Specify relevant attributes

5.3.5.5 Building a Perception

In this task a model of the domain as perceived by the perceiver is built. We propose building the perception by using the perception-schema and its perception-element-types. The first step in this process is identification of relevant phenomena for the perception,

i.e., the various objects and their relations, behavior patterns in the domain, and constraints.

The next step includes classification of the phenomena to one of the element-classes and addition of more detailed information, i.e., specification of the various attributes of the different element-aspects for the perception-elements. In this step it is possible to add qualified and quantified information regarding the various attributes.

This process should be iterative and involve domain experts. Verification and validation to ensure that the perception appropriately describes the application domain from the viewpoint of the perceiver are essential.

In summary:

1. Identify and classify perception-elements
2. Represent each perception-element using the appropriate perception-element-type.

5.3.5.6 Integrate Perceptions

A domain model is built as an integration of the various perceptions. Each perception consists of a set of perception-elements. In order to integrate these sets, we have to distinguish the various elements that constitute the domain. Then we have to compare the various perception-elements that represent a specific phenomenon; and resolve contradictions and differences in names, structures, and semantics. This task includes detailed definition of the content and boundaries of the domain based on coherence and relationships between elements. Thus, integration of perceptions requires the following activities:

1. Distinguish the various elements of the domain
2. Identify perception-elements that represent the same phenomenon
3. For each element
    - 3.1 For each relevant aspect
        - 3.1.1 If only one element-aspect exists
            - 3.1.1.1 Use it with no change
        - 3.1.2 Else (If more than one element-aspect exists)
            - 3.1.2.1 Compare element-aspects
            - 3.1.2.2 If attributes fit
                - 3.1.2.2.1 Use the element-aspect
            - 3.1.2.3 Else (Attributes do not fit)
                - 3.1.2.3.1 Try to resolve conflicts
                - 3.1.2.3.2 If conflicts remain unsolved
                    - 3.1.2.3.2.1 Include the various versions as different versions of the element-aspect.

5.4 Comparison with Existing Methods

Section 5.4.1 compares MegSDF's approach with modeling approaches used in system analysis. Section 5.4.2 compares it with existing domain-analysis approaches.

5.4.1 Comparison with System Analysis Approaches

MegSDF's approach can be considered as a generalization of system analysis approaches. Existing methods for system analysis, e.g., the object-oriented approaches of [BOOC 91], [RUMB 91], [MONA 92], the Dual-Model [GELL 91], [GELL 91a], the Structured Analysis approach [DeMA 78], and the ER data-modeling approach [ELMA 89], [KIM 90], typically model a single system and a specific instance of the domain. They capture only partial knowledge of the domain, e.g., only its static structure. Domain analysis in MegSDF, on the other hand, defines a universal, general, and comprehensive model common to all systems in the domain.

System analysis approaches generally use a restricted set of predefined modeling primitives. They do not use a meta-schema to describe the possible attributes of their modeling primitives, except for Booch's approach, which does includes a fixed set of predefined templates for element-types but does not divide their attributes into aspect.

[RUMB 91] uses data, dynamics, and functional models to represent different orthogonal and cross-linked parts (dimensions) of the model. Booch's approach includes a data dimension and part of a functional dimension but does not explicitly recognize dimensions or parts. The structured analysis approach [DeMA 78] deals primarily with the functional part. The dual model [GELL 91] deals with the data dimension and includes methods applicable to instances of object-classes only. The dual model divides the data model into semantic and structural parts. It describes the semantic and structural attributes of objects separately in object-class and object-type hierarchies. The ER approach includes only a data dimension.

The MegSDF approach is more open and flexible. It allows defining different dimensions and modeling primitives with templates that define element-types with their possible attributes organized by aspects. It is compatible with various modeling approaches.

None of the modeling approaches includes integration of different perceptions as a means of providing a comprehensive model. MegSDF domain analysis identifies significant perceivers and integrates their perceptions.

We recommend using object-oriented modeling for domain analysis. To achieve a comprehensive model, we need to augment the object-oriented model with information about the dynamic interactions of the domain. Table 5.7 summarizes this discussion.

Table 5.7 A Comparison of MegSDF Approach with Modeling Approaches

Character-istics	**MegSDF**	**Object-Oriented [RUMB 91]**	**Object-Oriented [BOOC 91]**
Type of System	Systems of systems and generic systems	System	System
Model Type	Universal, general, comprehensive, domain model	An instance of a domain	An instance of a domain
Modeling Schema	Domain-Schema, dimensions, aspects, element-types	No Schema for the modeling approach	Element-types
Dimensions	Dimensions group interrelated elements and divide the model into manageable, orthogonal, and interrelated parts. Number of dimensions depends on the domain and the modeling approach.	Uses data, dynamic, and functional dimensions	Data dimension with some functionality
Element-Types	Uses element-types. No restriction on number or kind of element-types.	None	Uses templates for a restricted set of element-classes.
Aspects	Uses aspects to group attributes of elements	None	None
Perceptions	Integrates multiple perceptions	None	None

Table 5.7 - A Comparison of MegSDF Approach with Modeling Approaches (Continued)

Character-istics	**Dual-Model [GELL 91]**	**Structured Analysis**	**ER Modeling**
Type of System	Database Systems and their integration	System	System
Model Type	A specific instance of a domain	A specific instance for a limited part of the domain	A specific instance for a limited part of the domain
Modeling Schema	No schema for the modeling approach	No schema for the modeling approach	No Schema for the modeling approach
Dimensions	Semantic and structural	Functional only	Data only
Element-Types	None	None	None
Aspects	Each element is described from semantic and structural aspects only	None	None
Perceptions	Does not include multiple perceptions for modeling *	Does not include perceptions for modeling	Does not include multiple perceptions for modeling *

*Views in the context of database are used.

5.4.2 Comparison with other Domain Analysis Approaches

Existing domain analysis approaches are primarily intended for software reuse for families of systems only [NEIG 81], [PRIE 91a], while domain analysis in MegSDF is intended

for developing and integrating Mega-Systems, i.e., both systems of systems and generic systems (families of systems).

[WIMM 92] uses domain analysis to represent domain knowledge. In our approach and Wimmer's, the domain model is used as a common knowledge basis for all projects developing systems in the domain. [PRIE 91a] uses a domain model only to identify common objects used in software systems in the domain for further reuse. [ARAN 91] suggests including conceptual and constructive parts, e.g., plans to transform specifications to code, in contrast to our approach and Wimmer's, which include only conceptual modeling.

Existing methods for domain analysis rely on knowledge representation and acquisition methods, requirements specification, object-oriented or hypertext methods [PRIE 91a]. Our approach, in contrast, is not based on a specific method and is compatible with different modeling methods. Wimmer's approach is based on ontological concepts.

We have proposed structuring the model, using domain-schema, element-types, and aspects, as also suggested by [WIMM 92]. However, we additionally allow the use of dimensions to organize interrelated elements into groups and model separate parts of the domain. To ensure flexibility and generality the domain-schema is not fixed and other element-classes can be added. Similar domain-schemas are not explicitly used in domain analysis for reuse.

MegSDF's domain model is built by integrating different perceptions into a unified model. This integration is facilitated by the domain-schema. No other approach for

domain analysis proposes building different perceptions as a part of the modeling process. Table 5.8 summarizes this discussion.

Table 5.8 A Comparison of MegSDF with other Domain Analysis Approaches

Characteristics	**Domain Analysis in MegSDF**	**Domain Analysis for Reuse**	**Domain Analysis of Wimmer**
Objective	Support development of Mega-System	Reuse	Construct domain knowledge
Applicable to	Mega-Systems: systems of systems and generic systems (families of systems)	Family of systems	Systems in a domain
Usage	Common understanding basis for the various projects developing systems in the domain	Identification of common objects used in software systems in the domain	Tool for modeling applications
Model Type	Conceptual, non-constructive	Conceptual, constructive	Conceptual, non-constructive
Modeling Approach	Any approach	Knowledge representation, system analysis or hypertext	Ontological concepts
Schema	Based on domain-schema, dimensions, aspects, and element-types	None	Schema, aspects (called views), and object-schema
Perceptions	Multiple perceptions are integrated into one model	None	None

5.5 An Example for Domain Analysis in the Insurance Domain

This section includes a simplified example of domain analysis for the insurance domain with an emphasis on MegSDF concepts. An extended example for domain analysis according to MegSDF can be found in [AGAN 93].

Insurance is a system that enables a person, business, or organization to transfer loss exposure to an insurance company which indemnifies the insured for covered losses and provides for the sharing of the costs of losses among all insured [SMIT 87].

The objective of our domain analysis is to build a domain model as an integration of significant perceptions of the domain. This requires the identification of the significant perceptions. In the insurance domain, we identify the insurance company (which we call the insurer), the agent, and the insured as the significant perceivers.

The ***Insurer*** is a company or a person that contracts to indemnify another in the event of loss or damage. The ***Insured*** is a person, business, or organization who purchases insurance to cover himself against losses. Insurance companies usually market their products by agents. The ***agent*** serves the insured and represents the insurer. Other significant perceivers of the domain, e.g., the actuary who computes insurance rates, government regulators, and claims adjustors are omitted in this simple example.

In the following sections we describe these significant perceptions and their integration. For simplicity, we identify only elements of the domain model and avoid the details of each element. Each perception is built using multiple dimensions. We select the

static (object) and the functional dimensions based on the object oriented approach of [RUMB 91]. In a real example, the dimensions depend on the chosen modeling approach and the actual domain. We look first at the static dimension for each perception then the functional. In practice, each perception, with its multiple dimensions, is built separately; during the integration phase the appropriate dimensions of each perception are integrated.

For each dimension, the different perception-elements are mapped into actual domain phenomena. Resolution of conflicts and definition of a unified model are then demonstrated for the selected dimensions and perceptions.

5.5.1 The Static Dimension

The static dimension is illustrated by object diagrams consisting of objects (drawn as rectangles) and their relations (drawn as lines or arrows). Generalizations are designated by the $\wedge$ sign. Perception descriptions are typed using bold italics to denote an object and underlined bold italics to denote a relationship. A line denotes a one-to-one relation, an arrow denotes a one-to-many relation, and a double arrow denotes a many-to-many relation.

5.5.1.1 The Static Dimension of the Insurer Perception

The ***Insurer*** ***Issues*** ***Policies*** and is **Represented-by** ***Agents***. The Insured ***Purchase*** Policies ***Sold*** by Agents. The Insurer specialized to ***Life, Health, Property*** and ***Liability***. An Insurer is ***Reinsured*** by a ***Reinsurer.*** The Insurer ***Indemnifies*** a ***Loss*** ***Covered*** by a Policy.

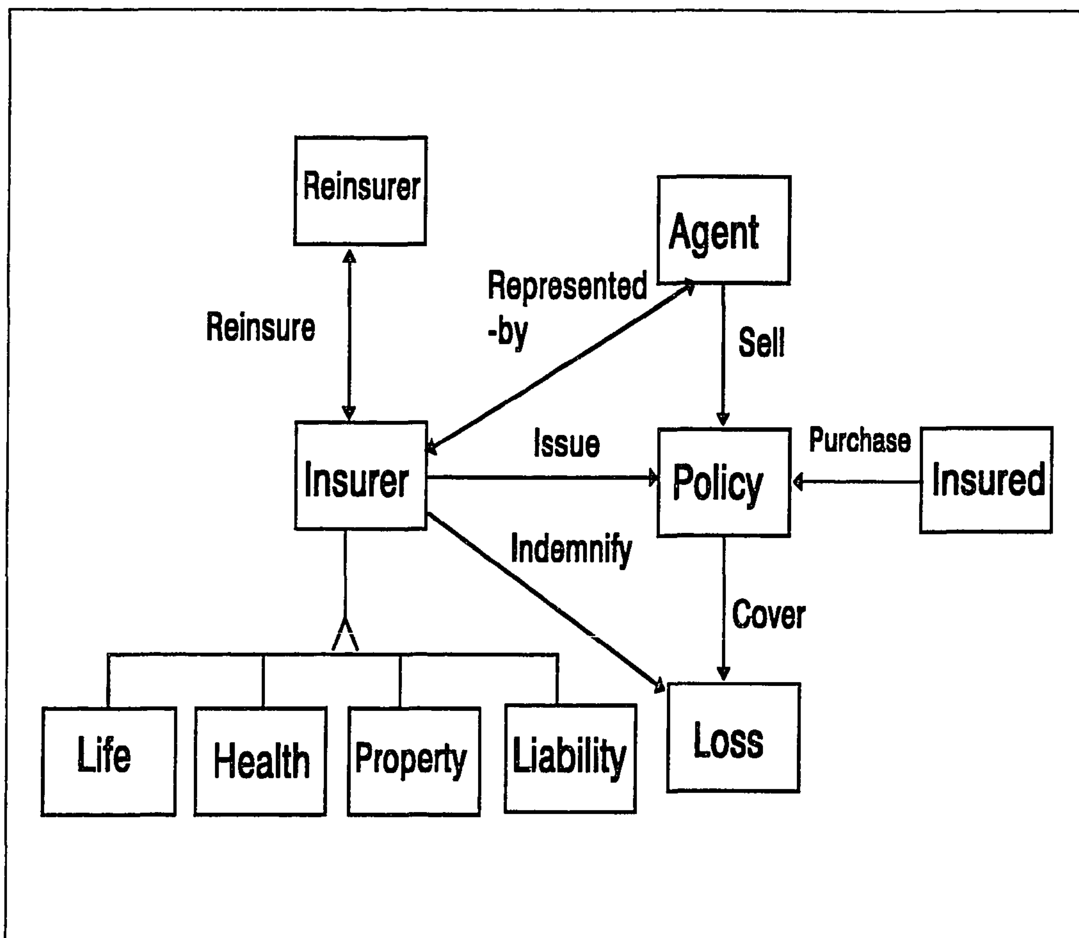

Figure 5.6 The Static Dimension of the Insurer Perception

5.5.1.2 The Static Dimension of the Insured Perception

The ***Insured*** ***buy*** ***Policies*** from an ***Agent***. Policies are ***Issued*** by the ***Insurer***. The Insurer is ***Represented-by*** Agents. Policies ***Cover*** ***Insurance-Items*** ***Owned*** by the Insured. Insurance-Items can ***have*** ***Losses***. Insurance-Item generalizes ***Car***, ***Life***, and ***Building***. The Insurer ***Compensates*** Losses of an Insurance-Item.

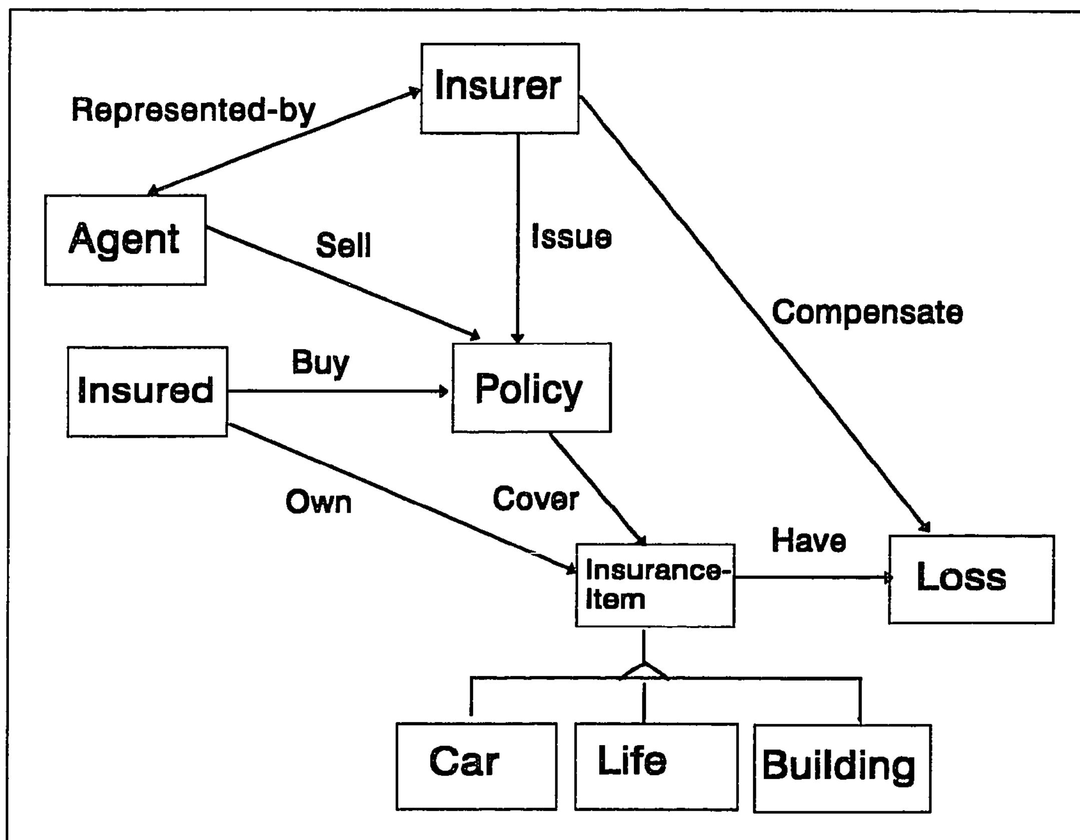

Figure 5.7 The Static Dimension of the Insured Perception

5.5.1.3 The Static Dimension of the Agent Perception

The ***Agent*** ***Represents*** the ***Insurers*** and ***serves*** ***Clients.*** Clients ***buy*** ***Policies***. The Agent has ***Private, Business,*** and ***Group*** Clients. The Agent ***Sells*** Policies ***Issued*** by an Insurer. Policies **cover** ***Losses*** ***Indemnified*** by the Insurer. Policies specialized to ***Life, Property,*** and ***Health***. Property Policies specialized to ***Building, Motor Vehicle,*** and ***Property in Transmit.***

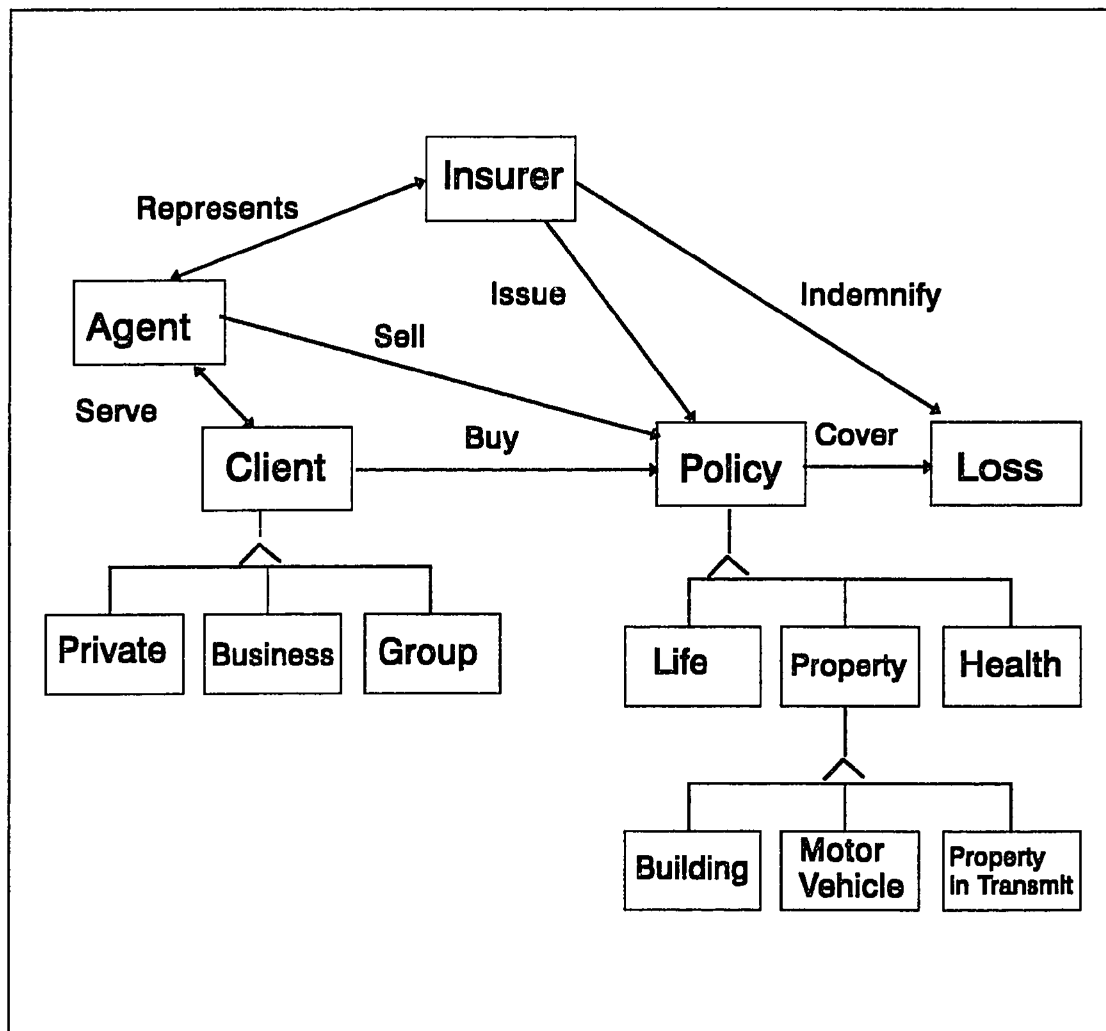

Figure 5.8 The Static Dimension of the Agent Perception

5.5.1.4 The Integrated Static Dimension

We use object and relationship tables to identify which perception-elements belong to the same phenomenon. The object table (Table 5.9) maps the appropriate perceptions-elements of each perception to domain objects.

Upon examining the objects in the different perception, we find:

- Objects that appear in all perceptions with the same name, e.g., ***Policy, Insurer,*** and ***Loss***. These elements will be included in the domain model using the same name.
- Objects having the same role but with different names, e.g., ***Insured***. This is called ***Client*** in the agent perception. We use the insured in the domain model.
- Objects that appear in only one perception, e.g., the ***Reinsurer*** in the insurer perception and the ***Insurance-item*** in the insured perception. Both elements are added to the domain model.
- Specializations that do not appear in every perception. We prefer to see all these specialized objects in the domain model. Thus, we include the specialized **insured** types, i.e., ***group, private,*** and ***business***, the specialized ***insurer*** types, the specialized insurance items, and the specialized ***policies***.

Table 5.9 Mapping Perceptions' Objects to Domain Model Objects

Insurer Perception	**Insured Perception**	**Agent Perception**	**Domain Model Objects**
Insurer • Life • Health • Property • Liability	Insurer	Insurer	Insurer • Life • Health • Property • Liability
Agent	Agent	Agent	Agent
Insured	Insured	Client • Private • Business • Group	Insured • Private • Business • Group
Policy	Policy	Policy • Life • Property • Health	Policy • Life • Property • Health
Loss	Loss	Loss	Loss
Reinsurer			Reinsurer
	Insurance Item • Car • Life • Building		Insurance Item • Car • Life • Building

Upon Examining the relationships in the multiple perceptions, we find that:

• Some relationships appear in all perception with the same names, e.g., ***Issue*** and ***Sell***. These relationships will be represented in the domain model under the same name.

• Some relationships appear with different names, e.g., ***Indemnify*** is also called ***Compensate***, and ***Purchase*** is also called ***Buy***. We select a name for these relationships and use it in the domain model.

• Several relationships do not appear in all perceptions, e.g., ***Reinsure*** in the insurer perception, ***Serve*** in the agent perception, ***Own*** in the insured perception. These relationships are all included in the domain model.

• A name of a relationship is used in different perceptions between different pairs of objects, e.g., ***Cover*** appears at both the agent and insurer perceptions between policy and loss and in the insured perception ***Cover*** appears between policy and Insured Item. We choose the insured names. Thus, ***Cover*** will represent the relationship between policy and insurance-item; ***Have*** will represent the relationship between insurance item and loss. We do not use the relationship between policy and loss.

The relationship table (Table 5.10) consists of pairs of objects, the perception names, and the name of the relationship in domain model.

Table 5.10 Mapping Perceptions' Relationships to Domain Model Relationships

Objects	**Insurer Perception**	**Insured Perception**	**Agent Perception**	**Domain Model Relationship**
Insurer-Agent	Represented-by	Represented-by	Represent	Represented-by
Insurer-Policy	Issue	Issue	Issue	Issue
Reinsurer-Insurer	Reinsure			Reinsure
Insurer-Loss	Indemnify	Compensate	Indemnify	Indemnify
Policy-Loss	Cover		Cover	
Policy-Insured	Purchase	Buy	Buy	Purchase
Policy-Insurance-Item		Cover		Cover
Agent-Policy	Sell	Sell	Sell	Sell
Insurance-Item-Loss		Have		Have
Insured-Insurance-Item		Own		Own
Agent-Insured			Serve	Serve

Figure 5.9 illustrates the integrated static dimension.

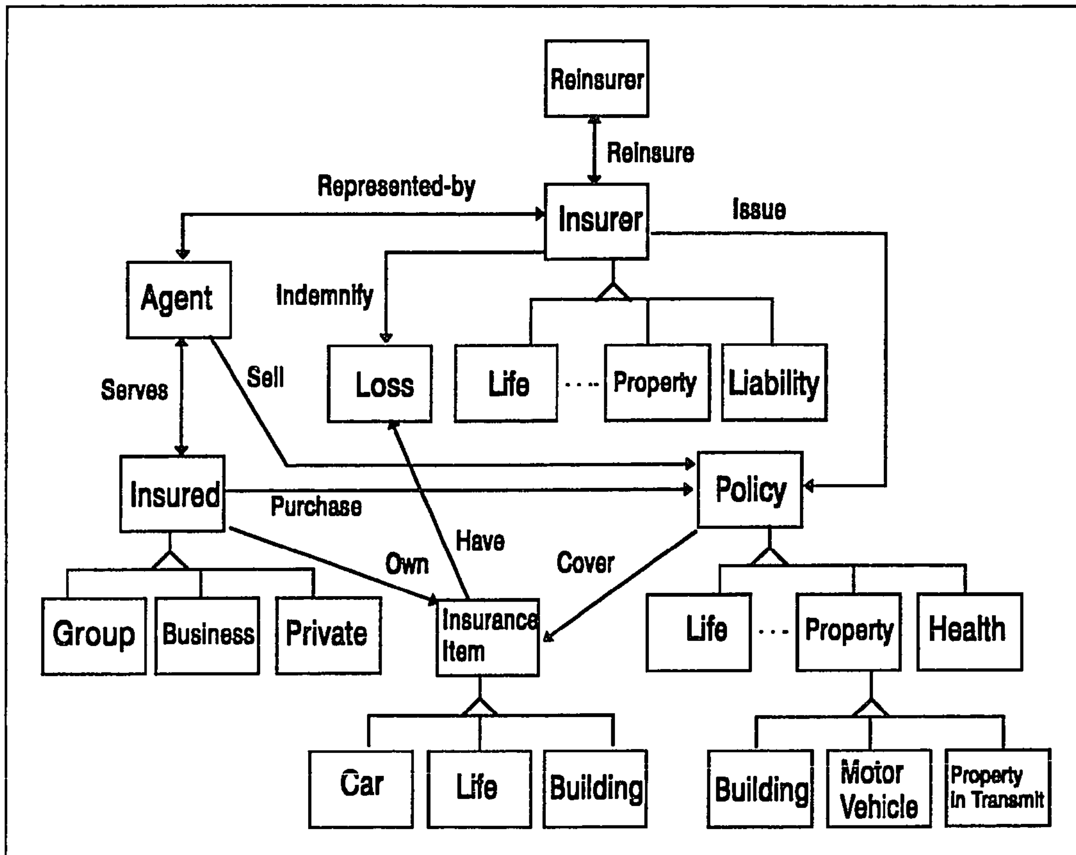

Figure 5.9 The Integrated Static Dimension

5.5.2 The Functional Dimension

The functional dimension includes processes (drawn as bubbles), data and control flows (drawn as solid and dashed arrows), and data stores (drawn as double lines). Sources or terminators are drawn as squares. We use the process of issuing a policy to illustrate the integration of the functional dimension.

5.5.2.1 The Functional Dimension of the Insurer Perception

In the insurer perception, the Insured ***Fill-in an Application*** for insurance of an insurance-item, provide ***Insurance-Item*** and ***Insured*** information, and ***Agree*** to the insurance terms. In the ***Underwrite a Policy*** a ***Policy*** is prepared for approval based on the ***Insurance-Rates*** computed in the ***Compute Insurance Rates*** by the actuary. These rates are computed according to ***Statistical Tables***. After ***Approving the Policy*** it is issued to the insured.

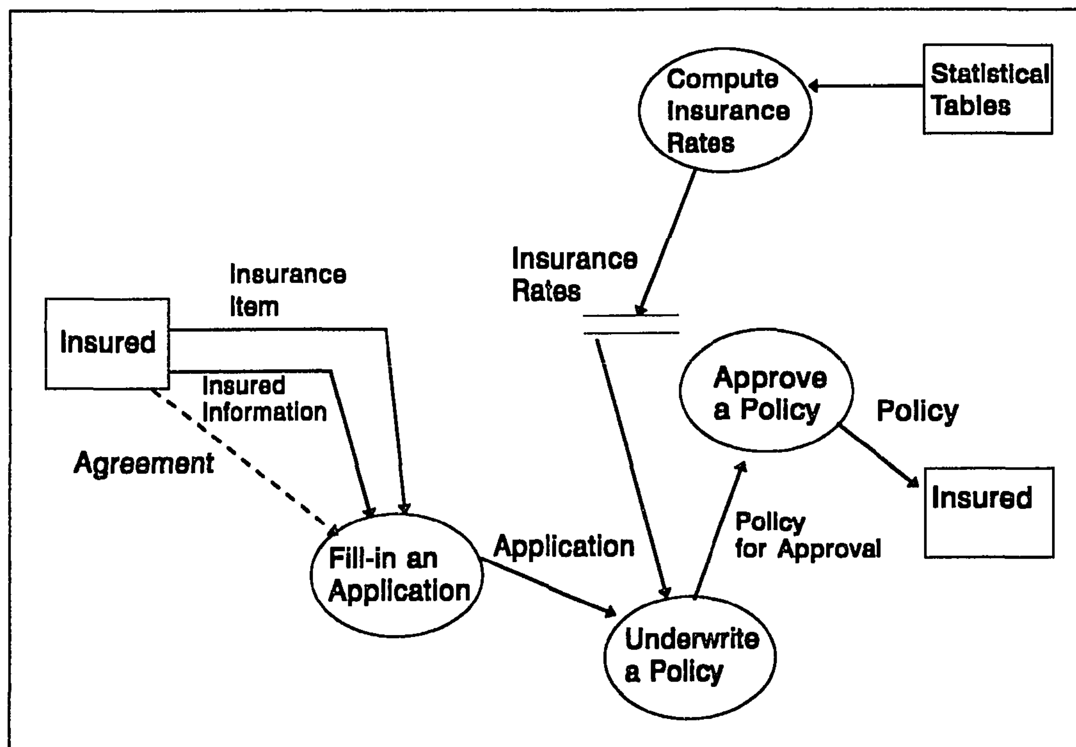

Figure 5.10 The Functional Dimension of the Insurer

5.5.2.2 The Functional Dimension of the Insured Perception

In the insured perception, an insured asks for quotes from different agents. The agents ***Prepare Quotes*** according to the ***Insured Item*** information. After several iterations, the insured ***agree*** and ***Fill-in an Application*** for insurance. Based on this ***Application***, a ***Policy is Underwritten*** and issued to the insured.

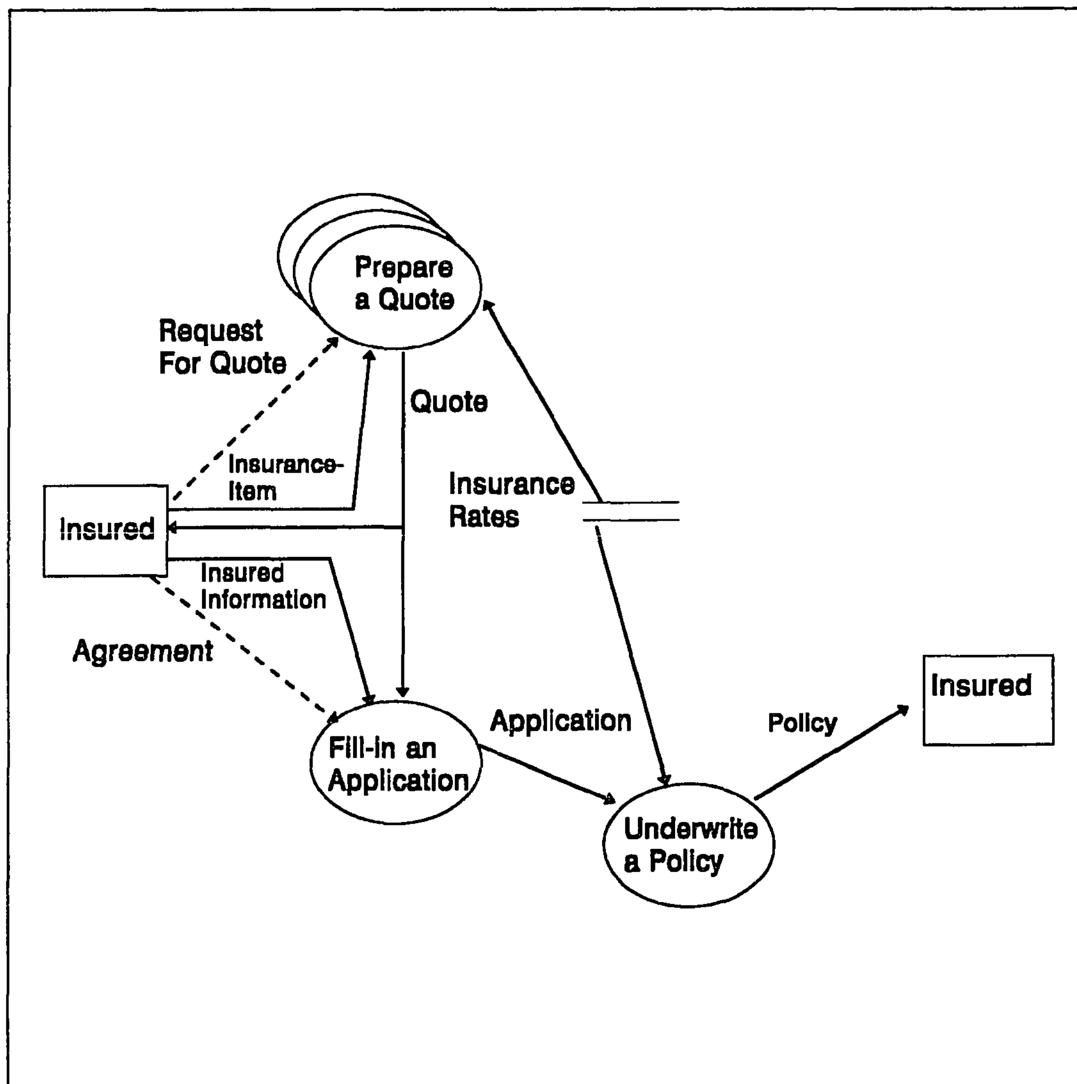

Figure 5.11 The Functional Dimension of the Insured Perception

5.5.2.3 The Functional Dimension of the Agent Perception

According to the agent perception, insured ***ask for quote.*** The agent ***Prepares a Quote.*** If the insured ***Agree*** to the quote, the agent and the insured ***Fill-in an Application.*** **A** ***Policy is Underwritten*** and is passed for ***Approval.*** The approved ***Policy*** is issued to the insured.

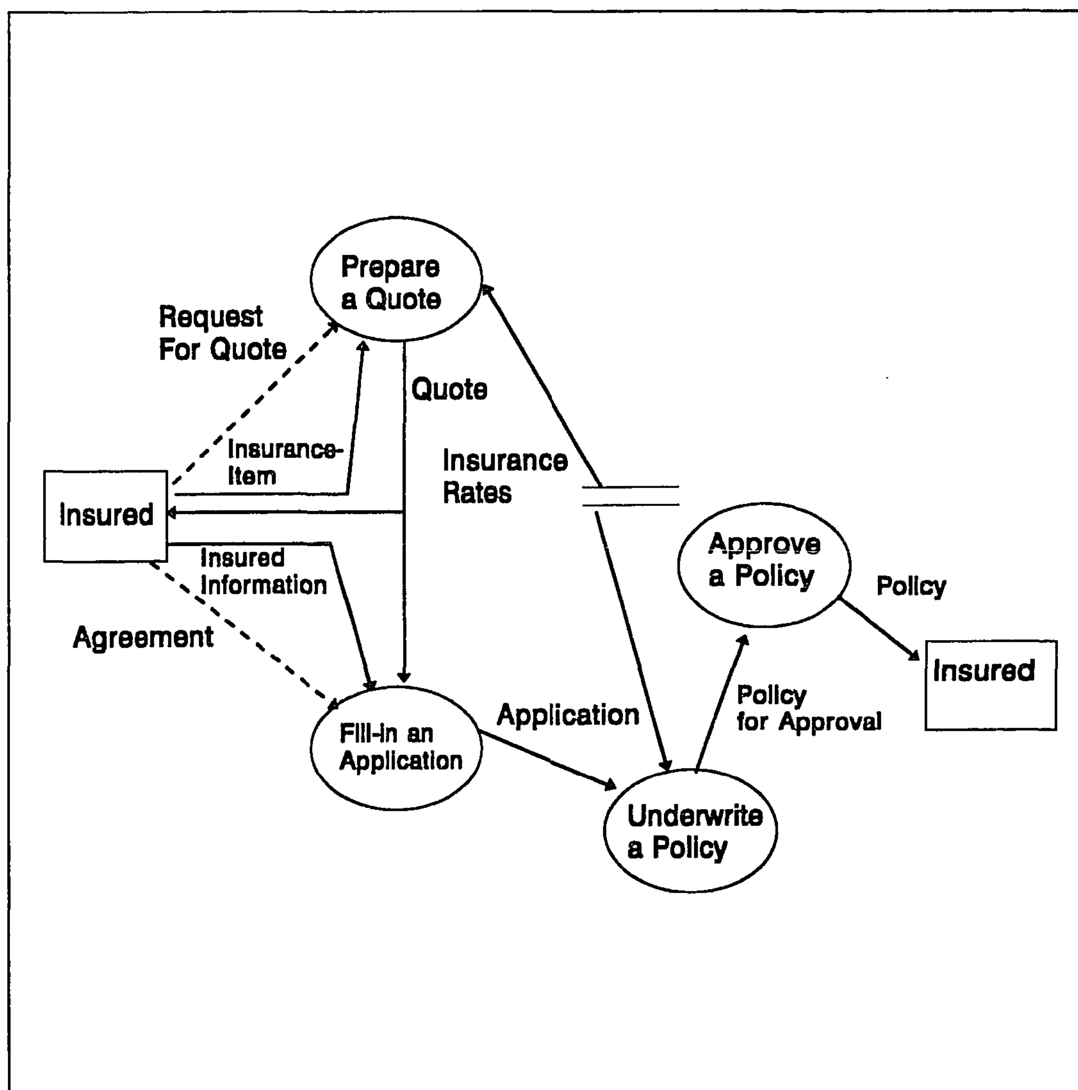

Figure 5.12 The Agent Perception Functional Dimension

5.5.2.4 The Integrated Functional Dimension

Similar to the integration of the static dimension, we use tables for mapping the perception elements into actual domain phenomena.

Table 5.11 lists the process elements. Examining the process elements we find:

• Processes that appear in each perception, e.g., ***Fill-in an Application***, ***Underwrite a Policy***. These processes are included in the domain model.

• Processes that appear only in some perceptions, e.g., ***Approve a Policy***, ***Compute Insurance Rates***, and ***Prepare a Quote***. These processes are included in the domain model, too.

• The ***Prepare a Quote*** process appears in one perception as a single process and in another perception as a multiple process. In this case we decide to represent it as a multiple process. The difference is caused since the insured can asks different agents to prepare quotes and only then to select one offer.

Table 5.11 Mapping Perceptions' Processes to Domain Model Processes

Insurer Perception	**Insured Perception**	**Agent Perception**	**Domain Model Processes**
Fill-in an Application	Fill-in an Application	Fill-in an Application	Fill-in an Application
Underwrite a Policy	Underwrite a Policy	Underwrite a Policy	Underwrite a Policy
Compute Insurance Rates			Compute Insurance Rates
Approve a Policy		Approve a Policy	Approve a Policy
	Prepare a Quote (multiple process)	Prepare a Quote	Prepare a Quote

Table 5.12 includes mapping of the control and data flows, the sources and terminators and the data stores of the perceptions into appropriate domain model elements.

Table 5.12 Mapping Flows, Sources, Terminators, and Data Stores

Insurer Perception	Insured Perception	Agent Perception	Domain Model
Insured	Insured	Insured	Insured
Insurance-Item	Insurance-Item	Insurance-Item	Insurance-Item
Insured-Information	Insured-Information	Insured-Information	Insured-Information
Agreement	Agreement	Agreement	Agreement
Application	Application	Application	Application
Policy for Approval		Policy for Approval	Policy for Approval
Statistics Tables			Statistics Tables
Insurance Rate	Insurance Rate	Insurance Rate	Insurance Rate
	Request for Quote	Request for Quote	Request for Quote
	Quote	Quote	Quote

Figure 5.13 represents the integrated functional dimension.

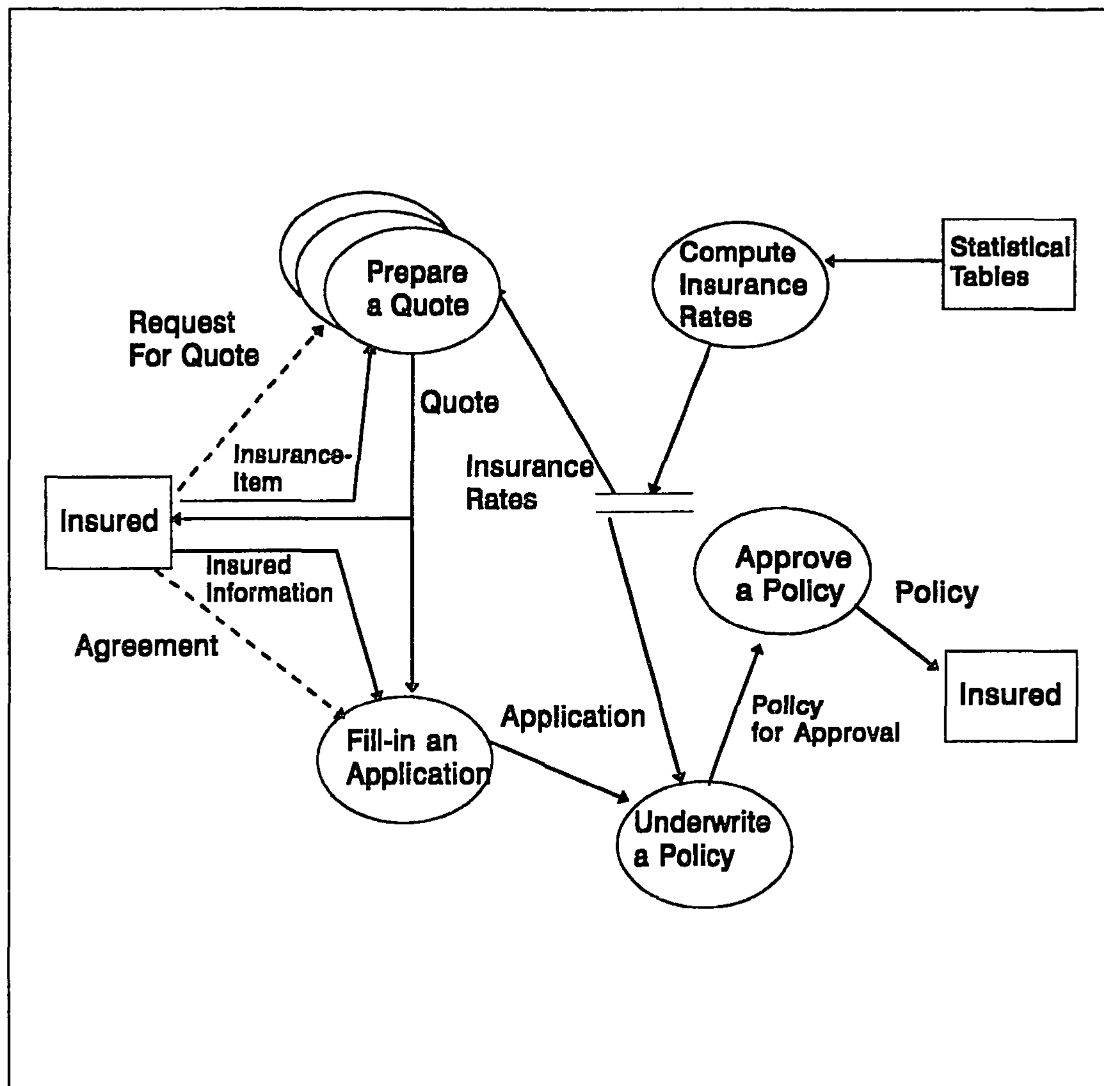

Figure 5.13 The Integrated Functional Dimension

CHAPTER 6

MEGA-SYSTEM ARCHITECTURE DESIGN

The Mega-System Architecture design task defines the strategy for the development of the Mega-System as a whole. It provides concepts to be used in the design and implementation phases of the constituent systems, defines requirements for the infrastructure, and specifies the overall structure of the Mega-System. The Mega-System architecture is used as a bridge between the domain model and the implementation and enabling technologies. We divide the Mega-System Architecture into ***Conceptual Architecture*** and ***Application architecture.***

The Conceptual Architecture defines design and implementation concepts and the requirements for the infrastructure. It generalizes the ideas of software system architectures. However, for Mega-Systems, the conceptual architecture is a necessity to ensure uniformity of the system not only over time but also in an environment that includes multiple developer groups and different projects. It specifies concepts for implementation and promotes reuse of components.

For manageability, we suggest dividing the concepts of the conceptual architecture into ***views*** where each view includes a set of interrelated concepts. Possible views are: structural, communication, control, data, environment. We discuss concepts for each of these views, but we observe that both the views and their contents will be domain dependent.

The Application Architecture specifies the system boundaries within the domain, main components of the Mega-System, and interfaces. The application architecture is an instantiation of the conceptual architecture. Application architecture design is similar to traditional software design, but works on a larger scale and is based on a conceptual architecture.

The process of Mega-System architecture design is continuous. Any change in the domain model as well in the enabling technologies should be evaluated and reflected in the architecture.

This chapter discusses the Mega-System architecture design task. Section 6.1 describes the role of the Mega-System architecture design in MegSDF and its required characteristics. Section 6.2 describes the underlying concepts for MegSDF's Mega-System architecture. A process for Mega-System architecture design is defined in section 6.3. Section 6.4 describes existing software architectures and relate them to MegSDF's concepts.

6.1 Requirements for Mega-System Architecture Design

6.1.1 The Role of Mega-System Architectures

The design of a Mega-System architecture is one of the Mega-System tasks. It defines a global strategy for developing the Mega-System. It includes guidelines for design and implementation which are to be common to and adhered to by all systems in the domain.

It defines the structure, boundaries, constituents, and interfaces of the Mega-System. It also maps the domain model to the implementation and enabling technologies.

Mega-System architecture design generalizes and extends traditional system design in two respects. It is intended for systems of larger scope and complexity, i.e., systems of systems and families of systems, and it specifies design concepts to be used by the entire system. The latter feature is either lacking or not thoroughly realized in traditional systems design.

Related ideas have been suggested previously and even been used in some projects. For example, Lawson [LAWS 92a] proposes defining a philosophy of system development. [SHAW 89] suggests higher levels of abstractions for software architectures. Perry et al. [PERR 92] recommend defining software architectures for large scale systems. Garlan [GARL 93] suggests development of a scientific basis for software architecture to enable new systems to be built, compared, and analyzed in rigorous ways. Project Ship-2000 [SS2000a, b] uses an architecture as a fundamental tool. The OSCA[1] architecture [OSCA 92] defines a conceptual architecture to be used in developing interoperatable systems for Bellcore Client Companies. We elaborate on these ideas and recommend Mega-System architecture design as an essential task in the development of any Mega-System.

Mega-System architecture supports the "pre-planned" approach. We claim it will enable efficient integration of systems. It may be considered as a meta-design, above the design of the constituent systems that finally constitute the Mega-System.

---

[1] OSCA is a trademark of Bellcore, inc.

Mega-System architecture design addresses difficulties in software development described in chapter 1, aiming at problems caused by the neglect of general, long term objectives; coordination and communication problems; neglect of the overall view of the system; the existence of multiple and unstable requirements; the existence of heterogeneous and non-standardized environments; and the need to bridge various technologies and to incorporate new technologies over time. These difficulties are listed in Table 6.1 as an inverted sub-table of the problem list (Table 1.1).

The Mega-System architecture deals with long-term goals and objectives. The architecture is a tool for engineering coordination between the various groups developing constituent systems of the Mega-System. The Mega-System architecture is intended to ensure the uniformity and consistency of the Mega-System. The architecture is also the specifications or requirements list for the infrastructure. In this respect, it must ensure that different environments are integrated, that the various enabling technologies are efficiently bridged, and that emerging technologies can be incorporated with minimal effort.

The common design principles recommended by a Mega-System architecture will enhance productivity by enabling the reuse of design concepts. Moreover, common design concepts will improve the traditional reusability of elements, i.e., programs, modules, etc., developed according to these concepts. A Mega-System architecture reduces the complexity that arises from using different approaches for the design and implementation of various components [HERB 89a], [PERR 92]. The conceptual uniformity imposed by a Mega-System architecture also improves the quality of the system.

Table 6.1 Difficulties and Problems Addressed by the Mega-System Architecture

Difficulties	Caused By	Aspect	Problems
Additional efforts are required for integration of systems	More than one system	Engineering	Current methods do not fit development of more than one system, with multiple and unstable requirements
The overall view of the system is neglected	More than one group of developers		
Multiple requirements	More than one customer		
Engineering solutions are required to close technology gap	Heterogeneous environment		
Unstable requirements	Long life cycle		
General objectives are neglected	More than one system	Management	There is no clear distinction between general, long-term objectives and local, short-term objectives
Coordination and communication problems on a larger scale	More than one developer		
No standardization of tools	Heterogeneous environment		
Long terms objectives are neglected	Long life cycle		
Heterogeneous environment	More than one system	Technology	There is a need to bridge the various technologies and efficiently incorporate emerging technologies as a common domain-wide solution
Each development group has to struggle independently with Heterogeneity and dynamic environments	More than one developer		
Bridging different technologies and incorporation of new technologies is required	Heterogeneous environment		
Customization to user environment	More than one customer		
Dynamic environment requires incorporation of new technologies	Longer life cycle		

A Mega-System based on a Mega-System architecture is intended to be planned and conceptualized rather than being erratic or random. The architecture conserves the structure and consistency of the Mega-System over time and despite the underlying Mega-Systems characteristics: long life cycles, dynamic requirements, and multiple groups of developers, any of which might destroy the integrity of the original structure.

The Mega-System architecture is used by all projects in the domain during the development and maintenance of a Mega-System. It serves as a guideline in the design phase of each project. The architecture is used by the Mega-System synthesis task as a general structure of the system. The architecture is influenced by the domain model, and is used as an essential input for the infrastructure acquisition task. Feedbacks from the system and the Mega-System synthesis tasks are used to improve the current architecture. The relationship of the Mega-System architecture with other parts of the framework is illustrated in Figure 6.1.

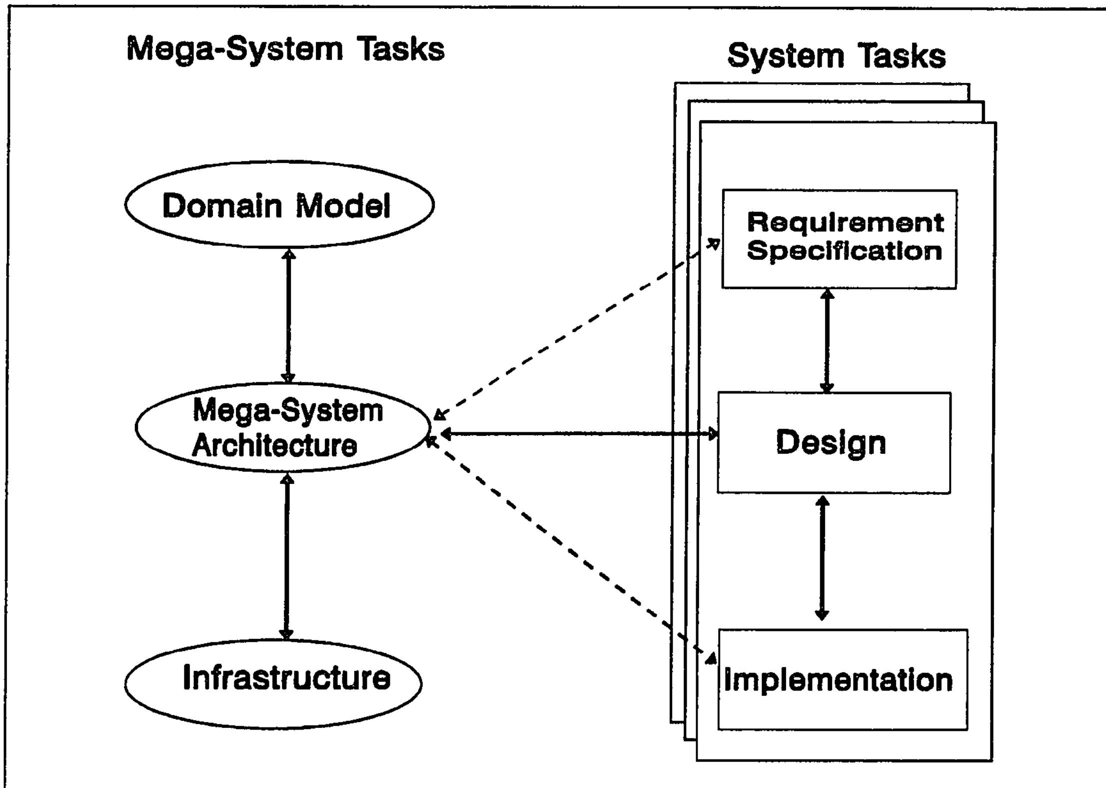

Figure 6.1 The Role of the Mega-System Architecture

Although the Mega-System architecture recommends a guideline and common design principles for the various constituent systems, this does not mandate a specific approach for developing a system. The only restriction is that each delivered system must be compatible with the proposed architecture specification in order to fit into the framework of the Mega-System. Moreover, the Mega-System architecture design task does not deal with the implementation of tools support for these concepts. Such implementation elements are dealt with separately in the infrastructure acquisition task which has to ensure that the concepts of the architecture are supported by the chosen infrastructure. From our viewpoint, tools that support software development by integrating enabling technologies are infrastructures, though they are often called architectures.

6.1.1 Requirements for Mega-System Architectures

MegSDF must be general and applicable to any domain or Mega-System. Therefore, the process of Mega-System architecture design must be applicable to any application domain. Consequently, the process must be flexible and domain independent.

To accomplish these goals, a Mega-System architecture has to assure that the Mega-System it supports is:

- Scalable and integratable,
- Flexible and technology independent,
- Manageable,
- Reliable, and
- Transparent.

The Mega-System architecture must be used for all Mega-Systems developed in the application domain so the architecture must allow configuration both for small instances with limited capabilities, as well as large instances that include extensive capabilities. In the domain of military vessels, for example, the same architecture may be used by a small coastal control ship, by a frigate, or by a submarine, although the features of every system will be different [SS2000a, b]. Thus, the Mega-System must be ***scalable***, i.e., it must be possible to add new constituent systems to the Mega-System or to remove/replace constituent systems with minimal effort. The Mega-System must also be ***integratable*** in the sense that it must be possible to efficiently integrate the Mega-System with other systems.

The characteristics: heterogeneous groups of users and long life cycles, require ***flexibility*** and ***technology independence***. The requirements of heterogeneous groups of users are multiple, not always well defined or known in advance. Moreover, long life cycles increase the possibilities for changes in requirements. Thus, we must recognize that there will be unknown and unexpected requirements. Therefore, the systems must be adaptable and changeable. The heterogeneity of user groups and their requirements increase the need for a Mega-System architecture to be as technology independent as possible. The architecture must fit various hardware configurations, i.e., different platforms and environments. Situations where systems have longer life cycles than the technologies they were originally implemented with, reenforce this necessity. The systems must be prepared for technological evolution.

The Mega-System architecture must ensure the Mega-System is ***manageable***, that is, modular, simple, and divided into well defined parts. Each part of the system must be highly cohesive and the coupling between the parts must be low. Such modularity supports developing a Mega-System by multiple coordinated projects. Each project develops a constituent system by applying an appropriate development approach, but yet complies with the concepts of the whole system.

Reliability is the extent to which a system operates without failure. It includes availability, consistency, security, and fault tolerance [TANE 92]. Availability refers to the time a system is usable. Availability is often increased in distributed systems by replication of servers, data, and resources. These replications enable partial services even when some part of the distributed system fails. However, replications also introduce

consistency and performance degradation problems. Even when some part of the system fails, the consistency of all replicated data is required. Mechanisms to assign appropriate servers to clients are required too. Security deals with protection of data and resources from unauthorized users. Distribution and replication increase the complexity of security mechanisms. Fault tolerance means that the failure of one system should neither degrade nor stop the other systems.

Though distributed systems are often designed to improve reliability, the complexity of these systems may aggravate reliability problems. The various aspects of reliability must be considered by the Mega-System Architecture design task to ensure the overall reliability of the Mega-System, despite the increased complexity.

Transparency deals with the ability to achieve a single system image. We distinguish two levels of transparency: transparency for the developers, and transparency for the user, as suggested by [TANE 92]. Developers' transparency means that the implementation of distributed systems operating in a heterogeneous environment be done in the same way as an implementation of systems operating in a homogeneous environment. Thus, the distributed system is developed on a virtual uni-processor. Developers' transparency includes location, migration, replication, and parallelism transparency [TANE 92]. Developers' transparency is a mechanism for achieving technology independence. On the other hand, transparency for the user masks the physical structure of the system from the user. The Mega-System appears to the users as a large single system that offers multiple services in a user-friendly manner, with a uniform user-interface, and allows efficient interaction between the various parts.

A Mega-System must be more than its parts. It must provide at least the same services as its independent constituent systems, with the same performance and quality. For example, if sharing of data is done by replication of data, this replication should not degrade the performance of the system. But beyond accumulation of services provided by the constituent systems, the Mega-System also provides added values not achievable otherwise: a unified view of all parts of the system, and efficient inter-system cooperation that avoids redundant data and functionalities and eliminates manual interfacing.

6.2 The Mega-System Architecture

A Mega-System architecture is the plan and strategy for the development of the Mega-System as a whole. It includes the concepts for the design and implementation of the system as well as the structure of the system, its boundaries, its various constituents and their interfaces.

6.2.1 Parts of the Mega-System Architecture

We divide the Mega-System architecture into conceptual and application architectures. The conceptual architecture includes definition of design and implementation concepts, e.g., the types of components, the communication approach, etc. The application architecture uses the concepts defined by the conceptual architecture to map the

application domain to an implementation. It includes definitions of system boundaries, specification of the different components of the Mega-System and their interfaces.

The conceptual architecture is general. It may fit multiple domains, but it must fit the given application domain. An application architecture is domain dependent. The application architecture is reused by the systems developed within the domain.

6.2.2 The Conceptual Architecture

A conceptual architecture specifies the concepts to be used in the design and implementation of the constituent systems and in defining the application architecture for the entire Mega-System. It abstracts implementation issues, identifies patterns of processing, and provides common conceptual solutions.

We propose existing Mega-System architectures and infrastructures be reused, or at least evaluated, before selecting a conceptual architecture. Moreover, as suggested by [PERR 92], there is a need to define architectural styles to facilitate reuse of architectures. These styles identify common and general conceptual architectures for major types of applications, e.g., real-time systems and data-processing systems. Typically, such architectural styles would be less restrictive and constrained than an actual conceptual architecture. When designing a conceptual architecture, the concepts of the style will be specialized and refined according to actual domain needs. In the application architecture design task, the application architecture will be specified as an instantiation of the conceptual architecture. The relationships between styles, conceptual architecture, and application architecture are illustrated in Figure 6.2.

The concepts of a conceptual architecture are interdependent. [PERR 92] suggests defining software architectures using three views: data, processing, and connection. The ANSA[2] project suggests using several viewpoints to describe distributed systems or architectures [ANSA 89]. Adopting these ideas, we define a conceptual architecture using multiple views, but we propose that the number of views and their content be domain dependent. Thus, a Mega-System architecture designer is free to decide what views are required and what level of details must be included in the architecture. Together, these views define the conceptual architecture.

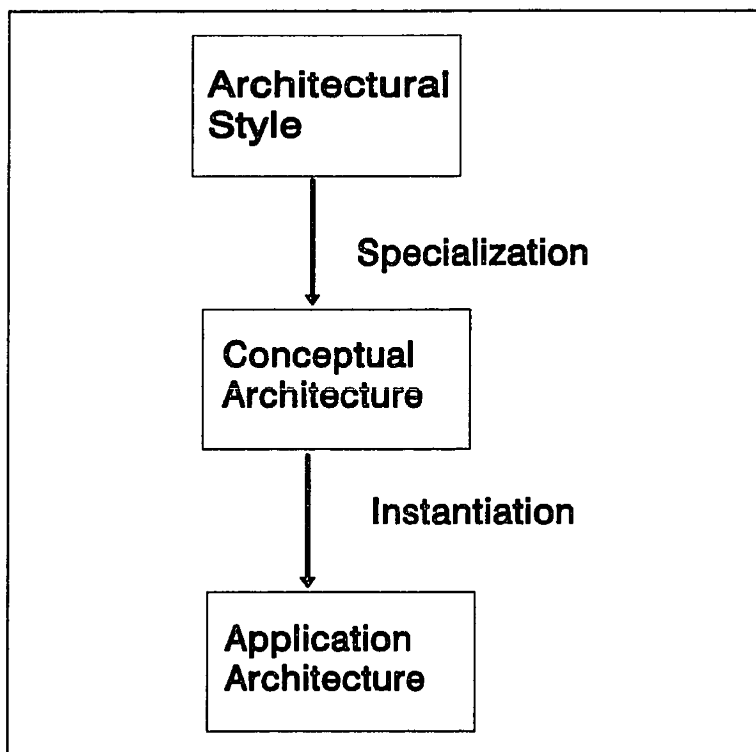

Figure 6.2 Architectural Style, Conceptual and Application Architectures

[2] ANSA is a trademark of Architecture Project Management Limited

Possible views for a conceptual architecture are:

• Structural view - specification of component (Building Block) types, relation between them, and guidelines for decomposition of the systems

• Communication view -a model for communication in the system

• Control view - a model for system-wide control

• Data view - a model for data handling

• Environment view -a model for interfacing with the environment of the system (the outer world) that includes human operators, other systems, and special purpose hardware.

We suggest not specifying a physical view that describes hardware configuration, i.e., processors and communication channels,and geographical organization, i.e., where to locate the various systems, as part of the conceptual architecture design task. These elements belongs to the Mega-System synthesis task.

The idea behind the conceptual architecture is to identify the appropriate views and specify the design concepts relating to these views. We suggest adapting existing international or commercial standards for the concepts of the views. This will promote integratability of the Mega-System with other systems and reduce the effort required for architecture design.

We recommend building a Mega-System as an open distributed system. This generalizes the federation of database systems suggested by Sheth and Larson [SHET 90]. A federation of systems consists of several autonomous systems that share data and control to achieve the required functionality.

The following sections describe possible views and discuss some basic concepts of these views. In each view we specify what approach and concepts will support our goals. However, for any Mega-System, we must define the concepts of each view according to the needs and characteristics of the domain.

6.2.2.1 The Structural View

The purpose of the structural view is to provide a framework for describing the organization of the elements of the Mega-System and their interrelation.

The size and complexity of Mega-Systems preclude their development as huge systems and suggest dividing them into components. Different architectures use different names for these components, e.g., systems, Building Blocks, layers, or computational units. A Mega-System may have components with different sizes and characteristics, e.g., Building Blocks, systems, or clusters. The conceptual architecture must specify the types of components for the entire system and possible classes of components based on processing type or other characteristics. For example, the OSCA architecture distinguishes between data, processing, and user interface Building Blocks [OSCA 92]. The conceptual architecture must specify the relationship between component types and constraints on each class and type of component. A conceptual architecture should also include a guideline for hierarchical decomposition of the system into components and sub-components.

A system is composed of components that provide its required functionality. The conceptual architecture only defines the types and classes of the components of the Mega-

Systems, but does not specify the actual components of the system. Actual components are specified by the application architecture. Actual components are instantiations of the component types identified by the conceptual architecture and must satisfy the constraints and rules specified in the structural view.

A typical structural view includes:

- Definitions of component types,
- Specification of classes of elements based on processing type or other characteristics,
- Specification of constraints on components, and
- Guidelines and rules for decomposition of an application into components.

We divide a Mega-System into loosely coupled Building Blocks (BB). Building Blocks provide services (functionalities) to other Building Blocks or to the users of the system and have well defined interfaces. However, we do not specify types for Building Blocks as done by the OSCA architecture. We allow Building Blocks to have any type of processing. Architecture designers may define types for Building Blocks according to the actual domain needs.

Building Blocks are data capsules which hide implementation details. Generally, Building Blocks are large and can include multiple objects or object hierarchies. Thus, Building Blocks can be considered as meta-objects. Unlike objects in the object-oriented approach, these meta-objects do not exhibit inheritance.

Building Blocks are not typical traditional systems. A Building Block is an "open" version of a traditional system in the sense that they provide services for authorized users,

using well defined interfaces. Building Blocks are designed to enable efficient integration. A traditional system can be implemented by a single or by multiple Building Blocks.

To ensure modularity, low coupling, and high cohesion, Building Blocks are developed without rigid assumptions about other Building Blocks or specific configurations, e.g., implementation aspects or physical addressing. For the same reasons, the services provided by Building Blocks must be used only via the means defined in the communication model (see section 6.2.2.2).

We add the concept of clusters of Building Blocks to the structural view. Clusters group together several Building Blocks for communication, control, or managerial purposes. Clusters enable broadcasting and atomic operations and facilitate organizing development efforts into projects.

We shall, henceforth, refer to the components of Mega-Systems as Building Blocks.

6.2.2.2 The Communication View

The communication view provides a framework for the description of the interconnection between Building Blocks. The Building Blocks of the Mega-System must communicate to provide the required functionality. The hardware allows processors to send messages to other processors; the operating system allows sending messages between processes in different processors. The operating system may also allow virtual circuiting between processes using protocols ensuring a certain level of reliability. However, all these

communication features are low-level and an abstract level for communication is still required [JOSE 89].

To enable extendibility and to ensure uniformity, it is necessary to define a standardized communication model. All the Building Blocks (the components of the system) must communicate exclusively using the same communication model. Manageability requires prohibiting the use of any other possible communication technique between Building Blocks.

It is important to note that the communication view is a conceptual model only and must be free from any implementation influence. The implementation details of the communication view will be dealt with separately in the infrastructure task.

The underlying idea behind a communication model is that Building Blocks must specify interfaces. These interfaces are the "open view" of the Building Blocks, and therefore of the Mega-System as a whole. Communication between Building Blocks must utilize only these interfaces. This guarantees information hiding and eliminates the need to know implementation details.

The communication model is a way of providing developers' transparency. It abstracts implementation and physical distribution details. The communication model supports mechanisms that provide migration, replication and parallelism transparency. It also supports the uni-processor image: processes operating on different processors interact like processes operating on a single processor.

Memory sharing and message passing are the common approaches to communication. In memory sharing, Building Blocks use memory accessible by more than

one Building Block to transfer information and/or control. Examples of shared memory are common databases, abstract (virtual) global memory, or special hardware memories that allow access by more than one processor. In message passing, Building Blocks communicate by means of a communication channel. Examples of communication channels are Remote Procedure Call (RPC), broadcasting, and pipelines.

The communication model must specify communication primitives. It is possible to define these primitives based on the number of receivers and senders, and the type of links (permanent or temporary). Usually a message is sent from one source to one destination. This model is called port-to-port. However, it is often necessary to send the same message to several destinations. This could be done by sending separate messages for each destination, which is inefficient and increases communication load, or in the broadcasting approach, by defining groups of processes and sending a single message to the whole group. The latter approach requires special hardware and/or software mechanisms. Similarly, a receiver may receive a message from a specific sender or from several sources. The connections between the sender and receiver may be permanent, existing as long as the system is operating, or temporary, for a single message or for a specific session.

To achieve location transparency, the communication model must define an addressing method. Physical addresses restrict the use of the system; inhibit using similar resources or services in the event of failure; and prevent migration or reconfiguration for load balancing purposes. The use of logical names, on the other hand, requires translation into physical addresses. Such translation is often done by a server called a "trader". A

trader may become a bottleneck for systems with intensive communication. Moreover, a trader failure may cause system-wide failure.

Communication is an essential element in distributed systems, but may be a source of failure. Routing and message correctness problems are usually handled by communication protocols and the lower levels of the ISO/OSI protocol [ROSE 89]. Communication failures may be caused by the receiver or the sender. A receiver may fail or be busy and so unable to accept the message. In this case it is possible to repeat sending the message a limited number of times or until it is accepted. If a sender fails before its message is processed, it is possible to abort the processing of this message or to continue with no change. A communication model has to specify the policies for handling such failures.

A typical communication view includes definitions for:

- Communication style, e.g., message passing, shared memory,
- Communication primitives, e.g., port to port communication, Broadcasting, etc.,
- Constraints for load balancing, e.g., maximal length of a message,
- Specification of legal communication, e.g., what types of Building Blocks are allowed to communicate, by what primitives, and common message format,
- Specification of a location transparency mechanism, and
- Communication failure handling policy.

Regardless of the communication style, we recommend that the communication primitives include both port-to-port and broadcasting communication. Moreover, we propose that the infrastructure of the system use the same communication model to ensure

integratability of the various parts of the infrastructure. Thus, we suggest an "open infrastructure", in the sense of communication.

6.2.2.3 The Control View

The control view provides a framework for describing interactions between Building Blocks. Autonomy is the state in which a Building Block exists and acts as an independent entity. Building Blocks may be either autonomous or controlled by other Building Blocks. A controlled Building Block acts according to decisions of the controlling Building Block and is considered as non-autonomous. The objectives of systems integration is to interconnect systems that were originally autonomous. It is often required to sacrifice autonomy in order to achieve the required functionality and additional values.

Systems with a set of Building Blocks can be implemented as centralized or fully distributed. In the centralized approach, one of the Building Blocks controls the operation of all the other Building Blocks. In this approach, it is relatively easy to implement activities that span more than one Building Block, but the reliability of the system is decreased since a failure of the controlling Building Block may lead to failure of the entire system. This problem, can be overcome by duplicating the software of the controlling Building Block to allow election of a new controller in the event of failure. Centralized control may also become a bottleneck in systems in which the Building Blocks have many interconnections.

In the fully distributed approach, every system controls itself and communicates with other systems to achieve the required functionality. The implementation of this

approach requires special care with consistency control, and a recovery mechanism is required to assure reliability of the system. A form of distributed organization, called federation, which compromises between the fully autonomous and the centralized approaches for database systems is described by Sheth and Larson [SHET 90].

Sheth and Larson [SHET 90] and Veijalainen et al. [VEIJ 88] define different types of autonomy for distributed database systems: design, communication, execution, and association. A designer of Building Blocks with design autonomy is free to choose his own design for any element of the system, e.g., data content, representation, semantics, constraints, functionality, association, implementation, etc. A Building Block with communication autonomy can decide by itself whether to communicate with other Building Blocks. Execution autonomy allows Building Blocks to execute local operations. Association autonomy means that a Building Block can decide whether and how much of its functionality to share. These notions of autonomy can be generalized to all kinds of systems. The conceptual architecture has to specify what types of autonomy there will be in the system. The meta-management and application architecture design will specify the appropriate autonomy for the Building Blocks.

The control view must define types of control units. Beyond processes that operate as basic primitives, where each process has its own address space and a specific task, it is possible to define other control units, e.g., threads of control, clusters, or groups of processes. Threads are parts of a process that share the same address space, but each thread has its own program counter and status word. Threads are usually designed to

cooperate for a specific task. It is also possible to define groups or clusters of processes for efficient communication and to simplify recovery mechanisms.

The control view must define an invocation strategy and related primitives. The activities of a system may be invoked periodically or be event-driven. In the periodic case, a set of operations is done routinely. In the event-driven case, the operations of the system are activated according to external or internal events.

The concepts of synchronous and asynchronous operations also belong to this view. In a synchronous operation, a Building Block that activates a service in a second Building Block waits for acknowledgement from the server (receiver) and does not continue processing during that time. In the asynchronous approach, on the other hand, the Building Block that activates an operation in another Building Block does not wait for acknowledgement from the server, but continues processing.

Various kinds of parallelism are possible in a distributed environment. Parallelism requires operation ordering primitives, e.g., sequencing, optional, clocked, and parallel actions [HERB 89c]. Atomic transactions and nested transactions of database systems can be extended to atomic operations for any type of systems or operation. Mechanisms for atomic transaction and nesting of transaction and recovery must be adapted to the general case. The control view must describe the policy and approach to atomic operations.

In summary, a typical control view includes definitions for:

• Control approach (centralized, fully distributed)

• Control units (processes, threads, control clusters)

• Invocation approach (periodic loop, event-driven)

• Operation ordering primitives (sequencing, atomic operation, etc.)

We suggest developing Building Blocks with as much autonomy as possible. Since autonomy may lead to inconsistencies of data and since the centralized approach tend to be inefficient, we suggest introducing the concept of ***clusters of Building Blocks***. A cluster behaves like a distributed system with a centralized control that enables efficient communication, atomic operations, and consistency control. The Mega-System consists of several clusters. We recommend identifying clusters of Building Blocks using the domain model, and with due attention paid to other views of the architecture. We also recommend that the architecture support both synchronous and asynchronous operations, since using only synchronous operation restricts concurrency.

6.2.2.4 The Data View

The data view provides the framework for describing data elements and data handling in the Mega-System. Data are an essential part of every system. Indeed, systems of systems are generally formed for efficient sharing of data. However, systems of systems that operate in heterogeneous environments represent data differently, use different database management systems and modeling approaches, and even different semantics. Therefore, sharing data in a Mega-System in a heterogeneous environment requires substantial effort.

To ensure integratability, it is necessary to define a meta-data-model for the Mega-System. The meta-data-model includes definition of a common data-modeling approach, e.g., the ER model, relational model, etc., a common data representation approach, e.g., data types, accuracy, etc. The meta-data-model specifies categories of data, e.g., private,

shared, or replicated, and rules and constraints for handling these categories, e.g., where data is stored, who is responsible for its consistency, and what type of access is allowed and by whom.

The meta-data-model can be used by newly implemented systems. For systems that already exist, or for new systems developed with another data-model, the meta-data-model is used as a connection mechanism. If a constituent system does not use the meta-data model, it is required to provide an interface from the meta-data-model to the used data-model, and vice versa.

The use of common or canonical data-models for representing federated database systems is suggested by [SHET 90]. However, it is important to differentiate between the meta-data-model and a data-model. The meta-data-model includes only specifications for the common data-modeling approach and the categories of data which can be used to represent the domain-wide data model.

Connecting two systems, which use two different data-models, in Mega-Systems that have a meta-data-model, entails interfacing the first data model to the meta-data-model and the meta-data-model to the second data-model. In this case, it is possible to provide a direct interface between the two models, without using the meta-data-model as intermediary. Though, this solution that might be more efficient in the sense of performance, we suggest using a meta-data-model to avoid interfacing any two data-models. Using this approach, if we have N data models we need to provide only 2N interfaces (one for interfacing each model to the meta-data-model and one interfacing the meta-data model to each data-model) instead of N*(N-1) interfaces (for interfacing each

model with the other). Adding a system with a new data-model requires only 2 interfaces instead of 2N interfaces. Similar ideas regarding different approaches for integrating existing system are discussed in [CLAR 92].

For database systems, the data view must specify the database organization, i.e., how data is stored and managed. There are three approaches for organizing databases:

- Common repository,
- Distributed database with centralized control, and
- Distributed database with distributed control.

In the first approach, data is stored in a common repository. All systems use and even communicate via this database. This approach is restrictive and requires heavy adaption of existing systems to the common database. It also eliminates developing a system using another DBMS that might be more efficient for a specific case.

In the "distributed database with centralized control" approach, data is distributed, but managed by a centralized transaction handler. The handler is responsible for replication, consistency control, and atomic transactions (operations). It is easier to implement this approach than the distributed database with distributed control approach. But, the centralized transaction handler becomes a bottleneck for systems with components that are highly connected.

In the "distributed database with distributed control" approach, there is no centralized transaction handler and therefore all issues of concurrency control, redundancy, and transaction handling become more difficult. In this approach, the constituent systems are highly autonomous.

The data view must specify strategies for concurrency control, recovery, security and replication on the data-level. Minimally, the data view will be used to define data handling of domain-wide data-objects (shared information).

A typical data view includes:

• A meta-data-model, i.e., common data modeling approach, data types, data categories, and policies for handling them,

• Data organization for the database if applicable, i.e., common, distributed with a unique server, fully distributed,

• Specification of transactions primitives and mechanisms for atomic and nested transactions,

• Policy for shared data, and

• Redundancy and consistency control.

We suggest using the fully distributed approach and recommend using a meta-data-model to "glue" the various parts. This simplifies integrating any system to the Mega-System, as long as the other systems use the meta-data-model or interfaces from the data-model of the systems to the meta-data-model are provided. We also propose that shared data be handled as a Building Block, as suggested in the OSCA architecture [OSCA 92]. These Building Blocks serve as active data capsules with well defined interfaces and provide data oriented services for the various Building Blocks of the Mega-System.

6.2.2.5 The Environment View

The environment view includes guidelines and rules for developing interfaces between the system and its environment. Typically, the environment of a system includes operators or users. For real-time embedded systems, one can also define a hardware environment consisting of special purpose hardware components that the system interacts with. The environment might include also other software systems.

To achieve the user transparency described in section 2.1 it is desirable for a Mega-System to have a consistent user interface. Consistency eliminates the need to remember unnecessary details and reduces the complexity of usage of the system. Adapting the ideas of the System Application Architecture[3] (SAA) of IBM [MART 91], we suggest dividing the user view into two parts: presentation and user interaction. The presentation part specifies concepts and rules for presentation, i.e., different types of windows, standard layouts of panels and windows, use of icons, color, emphasis, and voice. The user interaction part defines types of interaction, e.g., selection, entering information, help mechanism, and error handling. For each type of interaction it is necessary to specify standard methods, e.g., function keys, pointing by mouse, direct commands. Standardizing these elements increases the productivity of developers and users.

The specification of rules and constraints on hardware interfaces is necessary to ensure flexibility and portability, in the sense that adaptation of the software to different hardware configurations, e.g., different sensors, be minimal.

---

[3] SAA is a trademark of IBM, inc.

The environment view must also define a strategy for dealing with security problems. It must suggest mechanisms and concepts to ensure data and resources are used only by authorized users.

A typical environment view includes:

- Specification of a common user interface,
- Strategy for special purpose hardware and external systems interfaces, and
- Strategy to ensure security in the system.

The actual domain determines which parts of this view are relevant. We suggest including in the user interface both presentation and interaction guidelines and rules. Strategies for interfacing with hardware or external systems and for security are required only when applicable.

6.2.2.6 An Outline for a Conceptual Architecture

Table 6.2 summarizes section 6.2.2.1-6.2.2.5 and includes an outline for a conceptual architecture based on views as discussed in these sections. Both views and their content are domain specific, so this outline is suggestive. An actual conceptual architecture might include different views with different concepts.

This outline can be used as a check list for definition of a conceptual architecture. Section 6.4 uses this outline together with the outline of the application architecture to compare and classify existing architectures.

Table 6.2 An Outline for a Conceptual Architecture

View	Concepts
Structural	Definition of component types
	Specifications of classes of components
	Specifications of constraints on components
	A guideline and rules for decomposition of an application into components
Communication	Communication style
	Communication primitives
	Constraints for load balancing
	Specification of legal communication
	Specification of a location transparency mechanism
	Communication failure handling policy
Control	Control approach
	Control units
	Invocation approach
	Operation ordering primitives
Data	A meta-data-model
	Organization of the database (when applicable)
	Specifications of transactions primitives mechanism for atomic and nesting transactions
	Redundancy and consistency control
Environment	Common user interface - Presentation
	Common user interface - Interaction
	Strategy for special purpose hardware and external systems interfaces
	Strategy to ensure security in the system

6.2.3 Application Architectures

The application architecture specifies the structure of the Mega-System for the actual domain. It is based on the domain model and maps the domain model into the implementation model. It is designed according to the concepts defined in the conceptual architecture and utilizes the infrastructure.

The application architecture is used by the requirements analysis phase of the system tasks to specify the boundary of the actual developed constituent system and its interfaces. The application architecture is also used by the mega-system synthesis task to specify the software configuration (see also section 8.3). Feedback from the system and synthesis tasks is used to improve the application architecture. In turn, feedback from the application architecture design task is used to improve the domain model and the conceptual architecture. The role of the application architecture in the framework and its relationship with the other components of the framework are illustrated in Figure 6.3.

The application architecture specifies the boundaries of the Mega-System within the domain on the basis of the domain model and the conceptual architecture. It specifies which parts will be computerized, which part will not, and the rationale for these decisions. The application architecture also specifies the building blocks that provide the required functionalities. In this respect the application architecture is an instantiation or extension of the conceptual architecture for a specific application domain. Each element of the application architecture is an instantiation of one of the component types of the conceptual architecture and adheres to the constraints and rules the conceptual architecture imposes on this type.

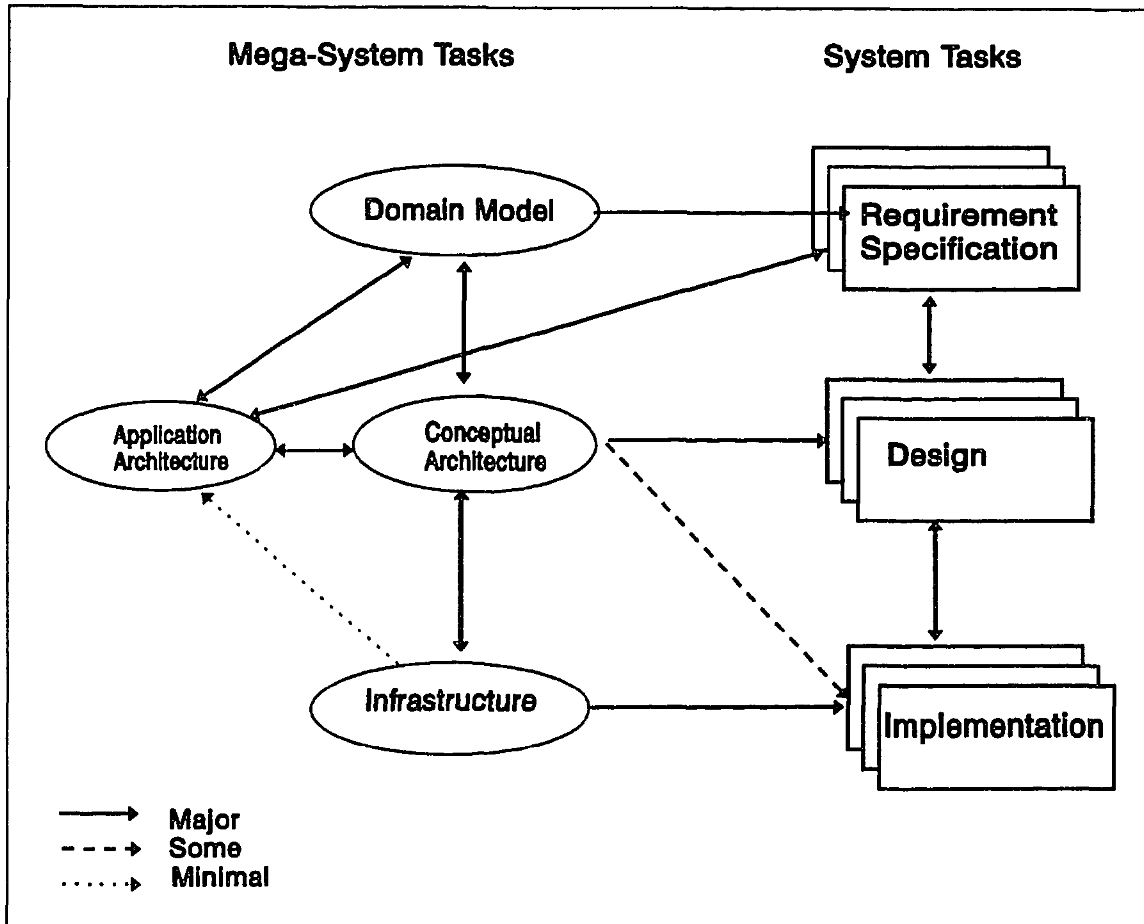

Figure 6.3 Relationship of the Application Architecture to Other MegSDF Elements

Every building block is classified as one of the building block types (when applicable). A building block is specified by the set of services it provides and by their interfaces. One way to form building blocks is by identifying sets of domain elements with high cohesion and low coupling. This reduces the load on the communication channel and minimizes the possibility that communication will become a bottleneck in the system.

In the event that the architecture defines clusters of building blocks, these clusters are similarly defined on the basis of the cohesion and coupling of the building blocks and the constraints imposed by the conceptual architecture.

The interfaces of the building blocks (and clusters) are specified according to the communication and control views. Interfaces with the environment are specified according to the environment view. Shared data, accessed by more than one building block, if not handled as a building block by itself, is identified in the application architecture and designed according to the concepts and constraints of the data view.

In some approaches that utilize architecture concepts, e.g., the Network of Application Machines [LAWS 92a, b], the application architecture includes the implementation of building blocks as generic units. This entails developing a Mega-System by "gluing" these elements, with or without customization.

We recommend a graphical representation to illustrate the building blocks and their interconnections. We also propose that the designers define a data-distribution map and service dictionary. A data distribution map specifies shared-data, their replications, and types of distribution (vertical, horizontal). The service dictionary maps services to building blocks and specifies replication of services.

A typical application architecture includes:

- A list of building blocks and for each building block its classification, list of services, and interface definitions.
- A list of clusters and for each cluster a list of participating building blocks.

• Building block interaction diagram - a diagram that includes the building blocks and the interconnections between them.

• Data distribution map - mapping of shared data into building blocks with definition of distribution and replication.

• Service dictionary - a list of the available services, their interfaces, and their replication approach.

Table 6.3 includes an outline for an application architecture based on these elements.

Table 6.3 An Outline for an Application Architecture

List of building blocks
List of clusters of building blocks
Building block interaction Diagram
Data distribution map
Service Dictionary

6.3 Mega-System Architecture Design Process

We describe the Mega-System architecture design process following the format defined in section 4.1.

6.3.1 Purpose

The purpose of the Mega-System architecture design task is to specify a Mega-System architecture for the Mega-System.

6.3.2 Interfaces

Inputs

• Domain Model - A model of the domain, defined in section 5.2.

• Users/Customers requirements - Requirements of the users/customers for the systems.

• Existing infrastructures and projected technologies - To define a feasible architecture, it is necessary to specify the concepts of the architecture with relation to existing and projected technologies.

• The Chosen Infrastructure - The infrastructure of the domain, defined in chapter 7.

• Feedback - Engineering information from the system tasks (projects) and the Mega-System synthesis tasks including recommendations for improvement and corrections to the current Mega-System architecture.

Control Inputs

• Management Control - The assigned schedule and milestones to the Mega-System architecture design task by the meta-management task.

Mechanism

• Architectural styles - Styles of conceptual architectures for Mega-Systems. These styles are evaluated in order to find the appropriate Mega-System architecture for the domain.

Outputs

• Mega-System Architecture - The architecture of the Mega-System, defined in section 6.2.

• Feedback - Feedback from the Mega-System Architecture design task to the domain analysis and meta-management tasks.

6.3.3 Processing

The Mega-System architecture design task defines a conceptual architecture as a specialization of a chosen architectural style and in accordance with the special characteristics of the application domain. Guided by this conceptual architecture and the domain model, the application architecture is then specified. Feedback from the application architecture is used to improve the conceptual architecture. Feedback from other tasks of the process is used to improve both architectures. Figure 6.4 illustrates the process diagram for the Mega-System Architecture design.

6.3.4 Timing

The Mega-System architecture design is an ongoing process since domains evolve over time. Any essential change in the domain or in enabling technologies and feedback from other tasks must be evaluated and reflected in the Mega-System architecture as required.

Although both conceptual and application architecture design are continuous processes, it is important to observe that the conceptual architecture is more stable than the application architecture. A change in the conceptual architecture means a change in a design or implementation concept. It is infeasible to change concepts frequently, though freezing them is not desirable either. The developers must remain open to new methods and adapt their concepts as required to ensure the competitiveness of their systems. The application architecture, on the other hand, must reflect any essential change in customer needs. It is more dependent on the dynamics of the application domain and so changed more often.

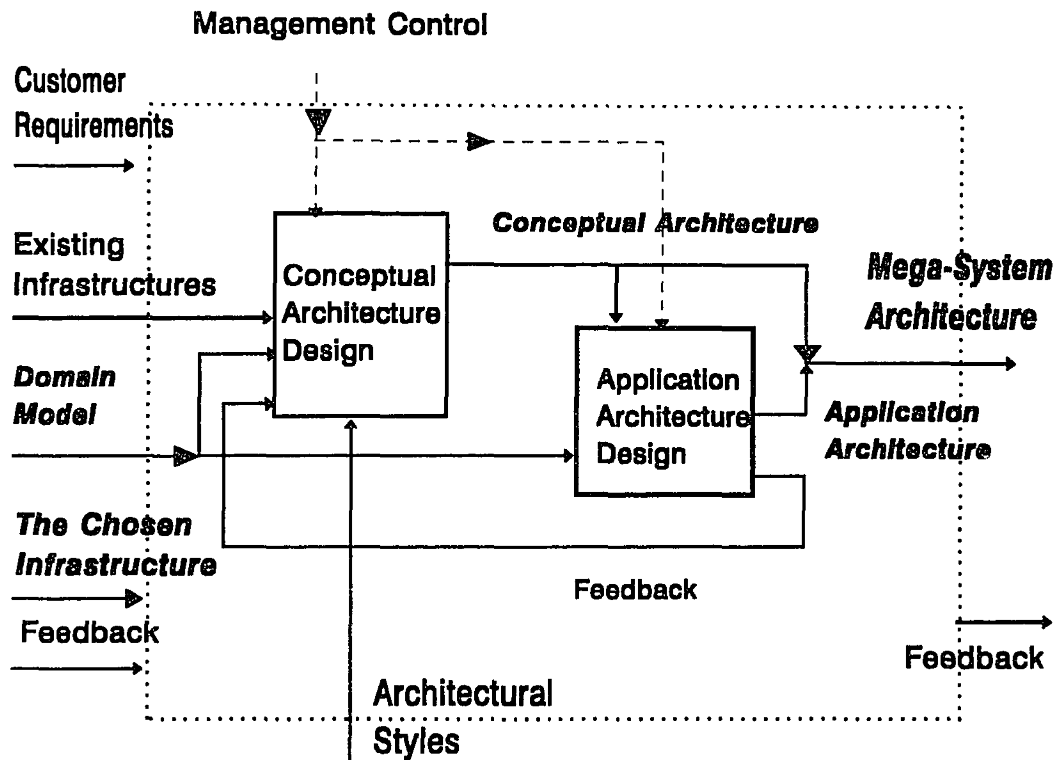

Figure 6.4 Mega-System Architecture Design Process

6.3.5 Sub-Tasks

6.3.5.1 Conceptual Architecture Design

6.3.5.1.1 Purpose

The purpose of the conceptual architecture design task is to specify the underlying concepts for the design and implementation of the whole Mega-System.

6.3.5.1.2 Interfaces

Inputs

• Domain Model - A model of the domain, defined in section 5.2.

• Users/Customers requirements - Requirements of the users/customers for the systems.

• Existing infrastructures and projected technologies - In order to define a feasible architecture it is necessary to specify the concepts of the architecture in relation to existing technologies and projected technologies.

• The Chosen Infrastructure - The infrastructure of the domain, specified in chapter 7.

• Feedback - Engineering information from the system tasks (projects) and the Mega-System synthesis tasks including suggestions for improvement and corrections to the current conceptual architecture.

Control Inputs

• Management Control - The schedule and milestones for the conceptual architecture design task assigned by the meta-management task.

Mechanism

• Architectural styles - Common styles of conceptual architectures for Mega-Systems. These styles are evaluated in order to find the appropriate Mega-System architecture for the domain.

Outputs

• Conceptual Architecture - The concepts and guideline for the Mega-System, defined in section 6.2.2.

• Feedback - Feedback from the task to the other tasks.

6.3.5.1.3 Processing

The process of conceptual architecture design specifies the actual views and the concepts of each view on the basis of the chosen architectural style. Figure 6.5 illustrates the conceptual architecture design process, which is summarized as follows:

1. Choose an appropriate architectural style
2. Define actual views for the architecture
3. For each actual view
   3.1 Specialize the actual view

6.3.5.1.4 Timing

The conceptual architecture design is an ongoing process since domains evolve and change over time. Any essential change in the domain or in enabling technologies, and any feedback from other tasks, should be evaluated and reflected in the conceptual architecture as required.

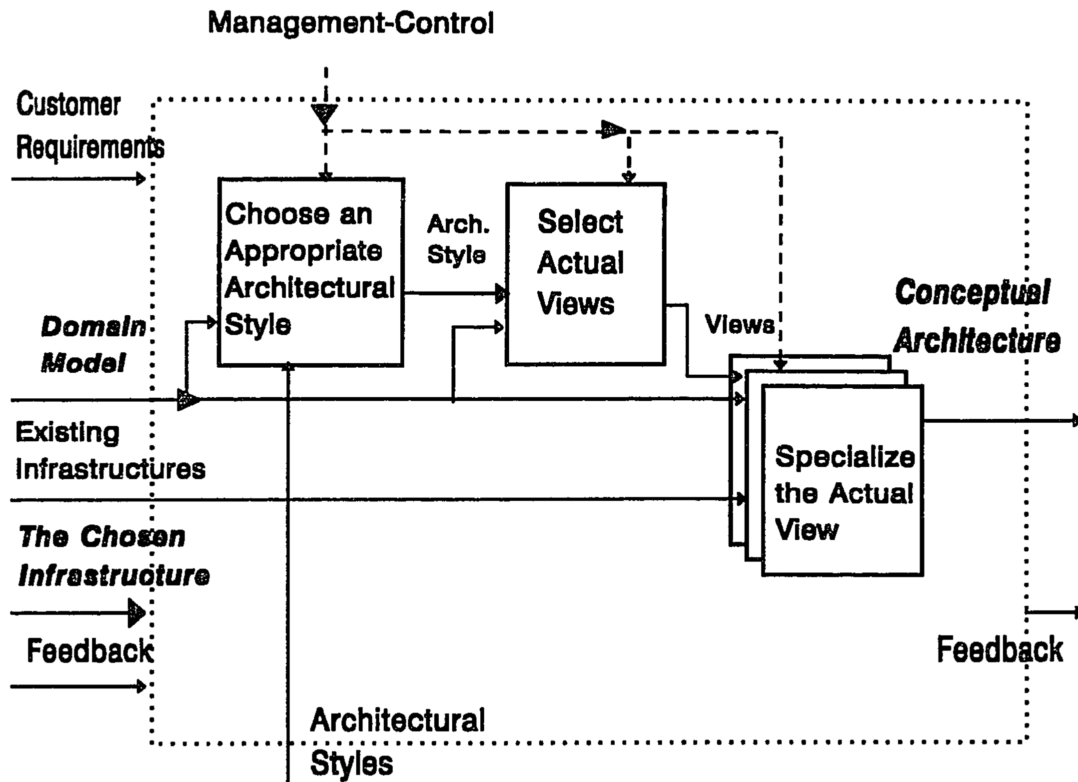

Figure 6.5 Conceptual Architecture Design Process

6.3.5.2 Application Architecture Design

6.3.5.2.1 Purpose

The purpose of the application architecture design task is to specify the application architecture for the Mega-System.

6.3.5.2.2 Interfaces

Inputs

- Domain Model - A model of the domain, defined in section 5.2.

• Users/Customers requirements - Requirements of the users/customers for the systems.

• The Chosen Infrastructure - The chosen infrastructure of the domain, specified in chapter 7.

• Feedback - Engineering information from the system and Mega-System synthesis tasks including recommendations for improvement and corrections to the current application architecture.

Control Inputs

• Management Control - The schedule and milestones for the application architecture design task assigned by the meta-management task.

Circumstance

• Conceptual Architecture - The concepts and guidelines for the Mega-System, defined in section 6.2.2.

Outputs

• Application Architecture - The overall structure of the Mega-System, defined in section 6.2.3.

• Feedback - Feedback from the conceptual architecture design task to the conceptual architecture design, domain analysis, and meta-management tasks.

6.3.5.2.3 Processing

The application architecture design task defines an application architecture which is an instantiation of the conceptual architecture and based on the domain model. Figure 6.6 illustrates the application architecture design process, which is summarized as follows:

1. Specify system boundaries

2. Identify building blocks

3. For each building block

   3.1 Define a building block {Specify BB type, role, BB interfaces and its services}

4. Identify clusters of building blocks (if clusters are defined as part of the architecture).

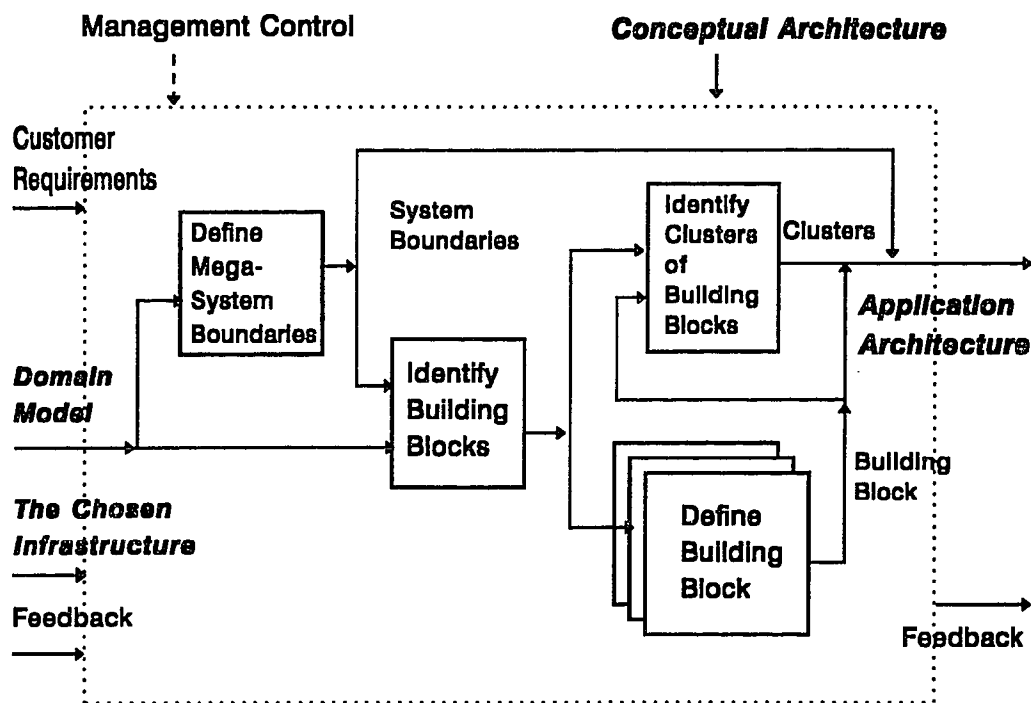

Figure 6.6 Application Architecture Design

6.3.5.2.4 Timing

The application architecture design is an ongoing process since domains evolve over time. Any essential change in the domain, the conceptual architecture, or the chosen infrastructure and any feedback from other tasks should be evaluated and reflected in the application architecture.

6.4 Existing Architectures

Mega-System architecture generalizes systems architectures. This section reviews existing systems and Mega-System architectures. It also describes Mega-System architectures that have been used in two large projects, (ESF [ESF 89] and Ship-2000 [SS2000a, b]). These projects defined a conceptual structure and proposed using a reference model/architecture as the basis for future implementations.

We discuss these architectures by mapping them into the Mega-System conceptual/application architecture dichotomy, and using the views suggested in section 6.2. The discussion of the architecture represents our point of view.

Incidentally, tools that integrate enabling technologies are often called architectures, but from our viewpoint are infrastructures and are not discussed here.

6.4.1 Systems Architectures

The Application Machine of Lawson [LAWS 92a, b] and Best's architecture [BEST 90] are systems architectures. They include a conceptual structure for a system which is an important contribution to system development. Lawson's approach for systems of systems is described separately in section 6.4.2.1.

6.4.1.1 Application Machine

The goal of the lawson's Application Machine [LAWS 92a, b, c] approach is to improve understandability of systems by focusing on essential properties of the applications.

Application Machines can be implemented for any domain. Application Machines have been used in the domain of embedded systems for automobiles, e.g., fuel injection system, brake control, etc.

The idea behind the Application Machines is what we have called conceptual architecture. Though views are not defined explicitly, one can identify elements of the structural, communication, control, and environment views; there is no reference to a data view.

The Structural View

There are two types of components in an Application Machine architecture:

- Application Machine, and
- Application Program.

An Application Machine consists of a set of declarations and Processing Operations (POPS) which have no decision making capabilities. An Application Program consists of a decision making part and invocations of POPS. The Application Program and the Application Machine together provide the required functionality of the system. Lawson suggests decomposing into POPs based on objects and operations. There is no classification of components or any constraint on building blocks.

Table 6.4 Application Machine Structural View Mapping

MegSDF View concept	Corresponding AM concept
Components	POPs and Application Program
Classes of components	Not defined
Constraints for components	Not defined
Guidelines and rules for decomposition	Based on objects and operations

The Communication view

Application Machine components communicate by shared memory. There is no definition of other concepts of the communication view.

Table 6.5 Application Machine Communication View Mapping

MegSDF View Concept	Corresponding AM concept
Communication style	Shared memory
Communication primitives	Not defined
Constraints for load balancing	Not defined
Specification of legal communication	Not defined
Location transparency	Not defined
Failure handling policy	Not defined

The Control View

The control of an Application Machine is done by a loop in the application program that activates the various POPS. Thus, the Application Machine uses a centralized control approach with application programs as control units. Lawson suggests using software circuits that might be considered as operation ordering primitives.

Table 6.6 Application Machine Control View Mapping

MegSDF View Concept	Corresponding AM Concept
Communication style	Centralized
Control units	Application program
Invocation approach	Periodic loop
Operation ordering primitives	Software circuits

The Data View

The Application Machine architecture does not specify data view concepts.

The Environment View

There is no definition of a common user interface. [LAWS 92c] recommends handling sensors by "software circuits" built out of "software components", e.g., sensors (or logical sensors), processors, and actuators. Each software component may activate several POPs. This approach standardizes the way the systems handle sensors (special purpose hardware) in the system environment. No security policy is specified.

Table 6.7 Application Machine Environment View Mapping

MegSDF View Concept	Corresponding AM Concept
Common user interface - presentation	Not defined
Common user interface - interaction	Not defined
Special purpose hardware/ external systems interfaces	Sensors Circuits
Security in the system	Not defined

Application Architecture

Lawson suggests identifying POPs and building a reusable library of POPs. This can be considered as an application architecture design. A system is then developed by implementing an application program and customizing POPs. There are no other elements of the MegSDF application architecture.

Table 6.8 Application Machine Application Architecture Mapping

MegSDF Architecture Element	Corresponding AM Element
List of Building Blocks	Reusable library of POPs
A list of clusters of building blocks	Not defined
Building blocks interaction diagram	Not defined
Data distribution map	Not defined
Service dictionary	Reusable library of POPs

6.4.1.2 Best's Architecture

This section describes Best's architecture [BEST 90] which defines a conceptual structure for large-scale information-processing systems in any application domain. Though Best calls his architecture an "application architecture", it contains elements of both the MegSDF conceptual and application architecture. Best indicates the approach can be used for systems with either a centralized or a distributed database. He claims that systems that are developed according to the suggested architecture can be efficiently integrated into systems of systems by merging appropriate drivers.

According to Best, every data-processing system must include the following superstructure functions:

- Batch transaction updating,
- On-line transaction validation and update,
- Sequential processing facilities,
- Output processing,
- On-line inquiry, and
- Exception data changes

The system must also include the following databases:

- Account/item database - the essential elements of the system,

and two supporting databases:

- Transaction database, and
- Extract database.

The basic organization and flow is illustrated in Figure 6.7. Each super-component is implemented as a driver that provides the super function and routes the processing to the actual application specific sub-program.

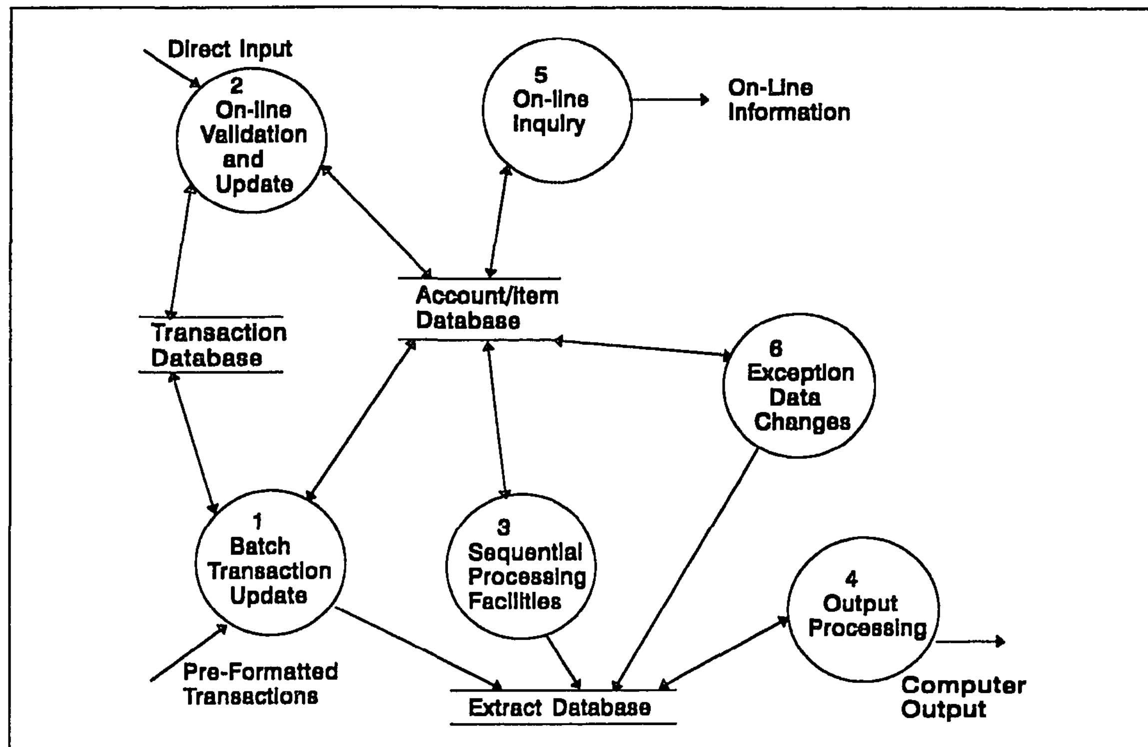

Figure 6.7 Best's Architecture Process Flow
(Copied from [BEST 90])

The Structural View

Best's approach identifies the super-functions and procedures as main components. He suggests dividing the systems according to processing type, which might be considered as classes of components. No other constraints on the components are specified.

Table 6.9 Best's Architecture Structural View Mapping

MegSDF View concept	Corresponding Best's Architecture Concept
Component	Super-components, procedures
Classes of components	Processing type, i.e., online, batch, etc.
Constraints for components	Not defined
Decomposition guidelines and rules	According to the type of processing

The Communication view

The communication in best's architecture is done by sharing databases (sharing memory). There is no definition of other concepts of the communication view.

Table 6.10 Best's Architecture Communication View Mapping

MegSDF View concept	Corresponding Best's Architecture Concept
Communication style	Memory sharing (using databases)
Communication primitives	Not defined
Constraints for load balancing	Not defined
Specification of legal communication	Not defined
Location transparency mechanism	Not defined
Failure handling policy	Not defined

The Control View

The control of the system is done by the super-functions. Each super function can be thought as an autonomous component. Batch super functions are invoked periodically. On-line processing super-functions are event-driven. There is no definition of operation ordering primitives.

Table 6.11 Best's Architecture Control View Mapping

MegSDF View concept	Corresponding Best's Architecture Concept
Control approach	Autonomous components
Control units	Super functions
Invocation approach	Periodic loops and event driven
Operation ordering primitives	Not defined

The Data View

Best's architecture neither defines nor uses a meta-data-model. Best claims it is possible to use the suggested architecture for either distributed or centralized database organizations. He defines databases that must exist in any application. To some extent it is possible to consider Best's suggestions regarding batch, on-line, and exception handlers as data-processing primitives. Similarly, it is possible to consider the integrity control programs as mechanisms for consistency control.

Table 6.12 Best's Architecture Data View Mapping

MegSDF View Concept	Corresponding Best's Architecture Concept
Meta-data-model	Not defined
Database organization	Distributed or centralized
Data processing primitives	Batch, on-line, data-exceptions
Redundancy and consistency control	Integrity control programs

The Environment View

Best describes only the user part of the environment view. He suggests using the "electronic desk" approach. He recommends limiting the details a user sees by showing summaries. The user can then choose which details to explore. Best also suggests supporting both on-line and batch transactions by exception overrides, exception data changes, and efficient help mechanisms and recommends using of security packages for on-line systems.

Table 6.13 Best's Architecture Environment View Mapping

MegSDF View Concept	Corresponding Best's Architecture Concept
Common user interface - presentation	Electronic desk, windows, menus
Common user interface - interaction	Both batch and on-line, exception handling, help mechanisms
Special purpose hardware and external systems interfaces	Not defined
Security in the system	Use of security packages to ensure secure and effective environment

Application Architecture

Best's architecture suggests a kind of "general application architecture" that fits applications in various domains. He specifies the main building blocks and databases. However, these building blocks might be considered as "processing oriented" since they are based on type of processing, e.g., on-line update and batch reports, and not on domain specific objects.

Table 6.14 Best's Architecture Application Architecture Mapping

Architecture Element	Corresponding Best's Architecture Element
List of Building Blocks	Super-functions and main databases
List of clusters	Not defined
Building blocks interaction diagram	See Figure 6.7
Data distribution map	Not defined
Services' Dictionary	Not defined

6.4.2 Mega-System Architectures

This section describes architectures which have been recommended as the basis for developing systems of systems or for system integration, which qualifies them as Mega-System architectures.

6.4.2.1 The OSCA Architecture

The OSCA architecture [OSCA 92], [MILL 90], [DESA 92] was developed by Bellcore. It is designed to promote interoperability and operability of software products systems. It is intended to provide a framework which will allow the systems of Bellcore Client

Companies (BCC), which are distributed over variety of computing environments, to interoperate [OSCA 92].

The OSCA architecture is considered both as a logical and strategic architecture. It consists of detailed guidelines for suppliers of software products, which must be compatible with these guidelines in order to be used by Bellcore Client Companies.

The architecture is oriented towards business software systems operating in heterogeneous environments. These systems, typically, use corporate data, so the architecture is intended to ensure accessibility to this data.

From our viewpoint, the OSCA architecture is a conceptual architecture that can be used in various domains. It specifies different types of building blocks as well as the restrictions and constraints they must adhere to. Though the OSCA architecture is not specified by terms of views, we can map the concepts of the OSCA architecture into the views we have proposed.

The Structural View

Building blocks are the main components of the OSCA architecture. A building block consists of a set of "business aware" functions. It can include sets of computer programs, data schemas, and other related software which process coherent, business aware functions with well defined interfaces. A building block can be deployed as a single unit and is release-independent of other building blocks. Software products that provide business-aware functionality may span more than one building block.

Building blocks support a principle called "concern separation", which is used to separate business-aware functionalities and business independent functionalities. The

business-independent functionalities are combined into the infrastructure; while the business-aware functionalities are subdivided into three layers to ensure "concern separation", the corporate data management layer, the business processing layer, and the human user layer. This division facilitates upgrading technologies, such as database management systems or devices that interact with users, without updating the entire system. Each layer consists of several building blocks, but each building block provides functionality that belongs to one layer only.

A building block provides a set of services defined by interfaces called "contracts". The grouping of contracts into building blocks is for administrative reasons only. A building block that invokes a contract does not care where the contract is installed or what other contracts it is grouped with. Contracts separate clients from implementation and internal details.

The OSCA architecture provides a detailed list of constraints for every class of building blocks. According to [DESA 92], decomposition into building blocks can be based on the object oriented approach and the three layers of functionality.

Table 6.15 The OSCA Architecture Structural View Mapping

MegSDF View Concept	**Corresponding OSCA Architecture Concept**
Component	Building Blocks
Classes of components	Data, processing, and user interface
Constraints for components	A detailed list of constraints
Guidelines and rules for decomposition	Classes of building blocks and object oriented approach

The Communication View

The OSCA architecture building blocks can communicate with any other building block that provides a required service. The building blocks communicate by message passing using the services of the infrastructure. The infrastructure consists of business-independent products that support business functions. Figure 6.8 illustrates the components of the OSCA architecture. The building blocks are location independent and have logical addresses. Building blocks cannot assume the availability of other building blocks and must gracefully accommodate their unavailability.

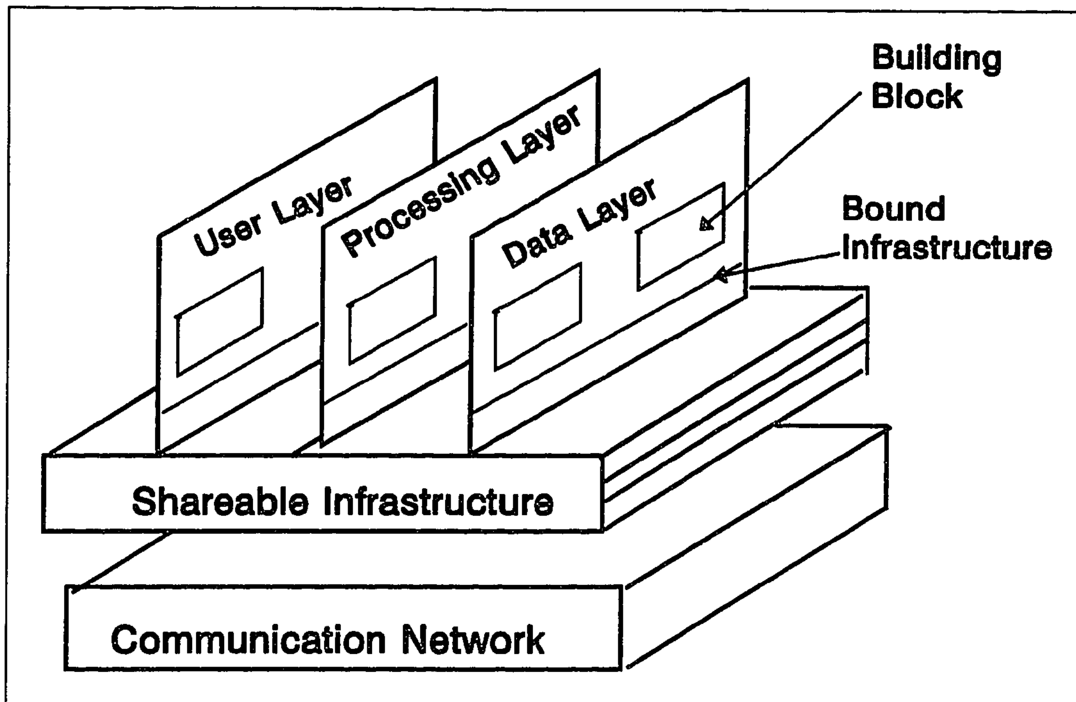

Figure 6.8 The OSCA Architecture

Table 6.16 The OSCA Architecture Communication View Mapping

MegSDF View Concept	Corresponding OSCA Architecture Concept
Communication style	Message passing
Communication primitives	Contracts
Constraints for load balancing	Not defined
Specification of legal communication	Every building block might communicate with any other building block using the contract and the services of the infrastructure
Location transparency mechanism	By means of the infrastructure
Failure handling policy	Gracefully accommodation of non-availability

The Control View

From our viewpoint, the OSCA architecture building blocks form a distributed system in which each building block controls itself. Each building block must have a recovery mechanism. For actions that span more than one building block, the OSCA architecture uses the concept of a logical building block. A logical building block is composed of more than one building block and acts as a recoverable domain.

Table 6.17 The OSCA Architecture Control View Mapping

MegSDF View Concept	Corresponding OSCA Architecture
Control approach	Fully distributed system (Autonomous building blocks)
Control units	Building Block, logical building block
Invocation approach	Event driven, client server
Operation ordering primitives	Recoverable domain for atomic operations

The Data Model

The OSCA architecture does not specify a meta-data-model for systems that use the architecture and does not recommend using a common modeling approach or common representation. However, the OSCA architecture does specify different data categories as well as rules for their handling. It distinguishes between "corporate" versus "private" data. Corporate data is used or created by the corporation to conduct its business. It is shared across business processes and partitioned into portions each of which is stewarded by a data layer building block. Corporate data is a corporate resource and is not the sole property of any single organization or software product.

Private data, on the other hand, is owned by a building block, not available for general retrieval or updating. It is allowed in any kind of building block. Private data may be redundant data or working data.

The OSCA architecture also allows "shared redundant data" in order to meet performance and availability requirements and to allow alternative views. This data is housed in a data layer building block and available only for retrieval purposes. A building block that stewards the data is responsible for supporting all redundant copies.

The OSCA architecture defines cooperative stewardship in cases where corporate data is stewarded by multiple building blocks. These building blocks form a single logical building block. The OSCA architecture specifies recoverable domains and transaction managers that are responsible for consistency control. The steward building block is responsible for recovery and consistency control for all replications.

Table 6.18 The OSCA Architecture Data View Mapping

MegSDF View Concept	Corresponding Concept in the Architecture
A meta-data-model	Data categories only
Database organization	Fully distributed databases
Data-processing primitives	Atomic actions by logical building blocks
Redundancy and consistency control	Recoverable domains, transactions managers. The steward building blocks are responsible for consistency of all replications.

The Environment View

The OSCA architecture does not explicitly specify an environment view in our sense, but it does specify a special layer of building blocks for user-interfacing, as well as a list of constraints and rules for this type of building block. The OSCA architecture also explains how to interact with systems outside the architecture.

The objectives of the user-interface layer of building blocks are to ensure concern separation and minimize dependency of the processing building blocks on special user-interfacing devices. With this organization, technology upgrades effect only the user interface building blocks.

The OSCA architecture recommends supporting multi-tasking (windowing) in order to allow users to work on several tasks simultaneously. Moreover, it recommends specializing building blocks for different users with different roles. A building block is required to accept the least expected input from the user and respond intelligently. The OSCA architecture also recommends providing customization features, and, as a minimum, consistency in presentation, though it does not specify a common user interface.

Interaction with external systems is bi-directional. If an external system requires service from a building block, it must use a building block contract. The external system must have a contract manager or use a transway (special software that provides building block and contract capabilities). Similarly, a building block that requires the services of an external system must send its request either through a transway or directly to an external system that has a contract manager.

The building blocks must adhere to security constraints. Besides limiting access to contracts only, the identity of the invoking user and its building block is passed through to any other building block. Sensitive data must be appropriately protected and building blocks might need to re-authenticate the identity of the invoking user or building block.

Table 6.19 The OSCA Architecture Environment View Mapping

MegSDF View Concept	Corresponding Concept in the Architecture
Common user interface - presentation	Consistency is required
Common user interface - interaction	Multi-tasking, specialized user-interface blocks for different users
Special purpose hardware and external systems interfaces	Policy for integration with external systems based on additional contract manager or transway
Strategy to ensure security in the system	Identity of invoking user and building block should be transferred. Sensitive data should be protected appropriately re-authentication.

Application Architecture in the OSCA Architecture

The OSCA architecture does not explicitly define an application architecture. However, it does suggest that the division into building blocks be based on low coupling and high cohesion of functions. It recommends that stewarding data building blocks (building blocks that manage the corporate data) be determined based on the information model of the Bellcore Client Companies.

6.4.2.2 A Network of Application Machines

This section describes the underlying concepts of a network of Application Machines as suggested by Lawson [LAWS 92a, b, c]. Just as the Application Machine, the network of Application Machines is proposed as a way to improve the understandability of systems by focusing upon the essential properties of an application. The network architecture is intended for problems with a larger scope.

A network of Application Machines architecture might be used in various domains. Networks of Application Machines are mainly used in the domain of embedded systems for automobiles, e.g., fuel injection system, break control, etc.

The ideas of a network of Application Machines can be considered as a conceptual architecture. The following paragraphs map the ideas of the network of Application Machines to MegSDF views.

The Structural View

The components of the network of Application Machines are the Application Machines described in section 6.4.1.1. The decomposition into Application Machine is based on objects and operations. There is no specification of classes of components or of constraints.

Table 6.20 Network of AM Structural View Mapping

MegSDF View Concept	Corresponding AM Architecture Concept
Component	Application Machines
Classes of components	Not defined
Constraints for components	Not defined
Guidelines and rules for decomposition	Based on objects and operations

The Communication View

The communication is handled by the router which sends messages to the various Application Machines. This model can be implemented by using the client server approach [LAWS 92a]. There is no specification of the other concepts of the communication view.

Table 6.21 Network of AM Communication View Mapping

MegSDF View Concept	Corresponding Network of AM Architecture Concept
Communication style	Message passing
Communication primitives	Not defined
Constraints for load balancing	Not defined
Specification of legal communication	Not defined
Location transparency mechanism	Not defined
Communication failure handling policy	Not defined

The Control View

A router synchronizes the various Application Machines and communicates with the outside world. The router deals with global situations of the entire system, while the Application Machines deal with local situations. The router itself can be an Application Machine. The router acts as a master in a master-slave architecture. The control approach is, thus, a distributed system with a centralized controller. The router uses periodic invocation. The software circuits may be considered as operations ordering primitives.

Table 6.22 Network of AM Control View Mapping

MegSDF View Concept	**Corresponding Network of AM Architecture Concept**
Control approach	Distributed system with a centralized control
Control units	Application programs
Invocation approach	Periodic loop
Operation ordering primitives	Software circuits

The data model

A common database serves the various Application Machines. The router can be responsible for redundancy and consistency control. There is no specification of a meta-data-model or definition of transaction primitives.

Table 6.23 Network of AM Data View Mapping

MegSDF View Concept	Corresponding Network of AM Architecture Concept
A meta-data-model	Not defined
Database organization	Common database
Transactions primitives	Not defined
Redundancy and consistency control	By the router

The Environment View

The environment view includes specifications only for sensor handling (see section 6.4.1.1).

Table 6.24 Network of AM Environment View Mapping

MegSDF View Concept	Corresponding Network of AM Architecture Concept
Common user interface - presentation	Not defined
Common user interface - interaction	Not defined
Special purpose hardware and external systems interfaces	Specification of concepts for sensor and logical sensors
Security	Not defined

The Application Architecture

According to Lawson, Application Machines are generic systems. They are developed as part of the application architecture and used by systems developed in the domain as

building blocks. The Application Machines do not undergo major change other than customization or parameterization.

Table 6.25 Network of AM Application Architecture Mapping

Architecture Element	Existence in the AM Architecture
A List of building blocks	Reusable library of Application Machines
A list of clusters of building blocks	Not defined
Building blocks interaction diagram	Not defined
Data distribution map	Not defined
Services' dictionary	Not defined

6.4.2.3 The CAN-Kingdom Architecture

This section describes the underlying concepts of the CAN-kingdom Architecture as suggested by Fredriksson [FRED 92a, b]. The goals of the CAN-Kingdom approach are:

• To support a machine development philosophy characterized by understandability, safety, simplicity, and effectiveness

• To ensure independence for module designers

• To enable efficient integration of third party modules

The domain of this architecture is stationary or mobile machine systems, e.g., spinning machines, weaving or knitting machines, saw mills, robots, cranes, excavators. The architecture fits mainly master-slave control systems and is based on the Controller Area Network (CAN) - real-time parallel processors systems.

Conceptual Architecture

Fredriksson uses the image of a kingdom to describe his architecture. The architecture emphasizes communication concepts, using post-offices, letters, envelops, etc., as reference metaphors. For systems, as with kingdoms, there is a need to specify "governing rules" and develop systems that operate according to these rules.

The Structural View

The entire system corresponds to kingdom. A kingdom has a Capitol and cities. The Capitol is the master of the system. The cites are nodes of the system and provide its services. The cities are connected by the CAN bus.

From our viewpoint, the components of the system are the cities, which provide the functionality of the system. Additionally, a system includes a special type of component, the Capitol, which controls the entire system.

Table 6.26 CAN-Kingdom Structural View Mapping

MegSDF View Concept	Corresponding CAN-Kingdom concept
Component	Cities (Nodes)
Classes of Components	Regular cities and a Capitol
Constraints for components	Not defined
Guidelines and rules for decomposition	Not defined

The Communication View

The CAN-kingdom architecture is based on a well defined protocol for communication that supports message passing. Each message is sent to all nodes. Each node identifies and

processes its messages. Some messages are used to configure the system at start-up time; other messages are used to transfer information to nodes. The structure of the messages is predefined. Every node receives and transmits messages identically, but different types of messages are defined according to the special requirements of the various nodes. The protocol defines a detailed message structure.

CAN-kingdom distinguishes between deterministic messages whose sequence and frequency are known in advance, and stochastic messages, which are event-driven. It also distinguishes between messages with deadlines and messages that are not time critical. Fredriksson recommends early processing of data and transmitting only essential results to avoid overloading the communication channel. Thus, CAN-Kingdom support distributed processing with a centralized control.

Table 6.27 CAN-Kingdom Communication View Mapping

MegSDF View Concept	**Corresponding CAN-Kingdom Architecture Concept**
Communication style	Message passing
Communication primitives	Every node receives all messages but processes only messages sent to itself. Deterministic/stochastic messages. Time critical/non-critical messages.
Constraints for load balancing	Transfer only essential data. Process data as early as possible and send only results and processed data.
Specification of legal communication	Predefined Protocol
Location transparency mechanism	Not defined
Failure handling policy	Not defined

The Control View

The architecture uses the master-slave (distributed system with centralized control) approach. The Capitol is the system master, the cities are the slaves.

Table 6.28 CAN-Kingdom Control View Mapping

MegSDF View Concept	Corresponding CAN-Kingdom Architecture Concept
Control approach	Distributed system with centralized control
Control units	Cities
Invocation approach	Not defined
Operation ordering primitives	Not defined

The Data View

There is no definition of any meta-data-model other than detailed definition of messages structure. Fredriksson recommends using different messages (called forms) for interfacing between different data representation methods.

The Environment View

The CAN-Kingdom architecture does not specify an environment view.

Application Architecture

The kingdom designer defines the functionality of each city and specifies its actual parameters by defining the system configuration at start-up. In this approach the same "city" can be re-used in different ways, depending on the needs of the kingdom. The

allocation of functionalities to cities can be considered as application architecture design. Fredriksson recommends using a graphical notation to represent interaction between cities.

Table 6.29 CAN-Kingdom Application Architecture Mapping

Architecture Element	Existence in the CAN-Kingdom Architecture
List of Building Blocks	The cities
List of clusters	Not defined
BB interaction diagram	Graphical representation of city interactions
Data distribution map	Not defined
Service dictionary	Not defined

6.4.2.4 The Advanced Networked Systems Architecture (ANSA)

The Advanced Networked Systems Architecture (ANSA) [HERB 91a, b, c], [ANSA 89] focuses on Information Technology (IT) that spans several domains. The goals of the ANSA project are:

- To propose an architecture for networked computer systems,
- To support distributed applications, and
- To promote the acceptance of the results of the project as an industry-wide standard.

ANSA is intended to enable integration of application systems from multiple vendors by using a distributed application platform that is independent of communications, operating systems, and the computer instructions set. It aims at an architecture which will provide the simplest set of concepts necessary to build distributed systems.

The ANSA architecture identifies five viewpoints for distributed processing: an enterprise model, an information model, a computational model, an engineering model, and a technology model. The viewpoints are interrelated but emphasize different aspects of the system. ANSA's viewpoints correspond to the essential elements of MegSDF, and not to the views of MegSDF's conceptual architecture. The enterprise and the information model can be mapped to the domain model. The ANSA computational model might be considered as a partial conceptual architecture. The concepts of the computational model are on a lower level and closer to technologies. The engineering and technology views can be mapped to the infrastructure.

The ANSA project concentrates on the computational and engineering viewpoints. These viewpoints are independent of both the application domain and the technology trends. Moreover, these viewpoints provide an environment for the specification of interfaces between the applications and the hardware and software that support them. The ANSA computational model identifies the functions (services) that must be available to programmers and the constraints on program structure necessary to enable distribution.

A federation of ANSA systems is built from systems, each running multiple applications. The individual system applications are linked together by a trader and configuration manager. Federation is achieved by linking together the traders of the various systems. The applications are considered as components that provide or utilize services. A precise specification of the interactions between components is necessary to enable independent development. ANSA suggests using an Interface Definition Language (IDL) for interface specification. Interface specification requires action, data, and property specification. An action is invoked only through an interface.

According to ANSA, different applications require different types of distribution and therefore different types of transparency mechanisms. On the basis of this idea, ANSA provides selective transparency in which a programmer specifies the required transparency when declaring an interface between applications. ANSA supports access, location, concurrence, failure, replication, and migration transparency.

The Structural View

The individual ANSA applications and systems can be thought of as components of the architecture. There is no definition of classes of components, constraints on components, or guidelines and rules for decomposition into systems and applications.

Table 6.30 ANSA Structural View Mapping

MegSDF View Concept	**Corresponding ANSA Concept**
Component	Applications (components), systems
Classes of components	Not defined
Constraints for components	Not defined
Guidelines and rules for decomposition	Not defined

The Communication View

The ANSA architecture uses the client-server approach. A trader supports the interaction between components and their applications. A service is accessible to other applications only after its server exports an interface reference to the trader. A client can retrieve interface references from the trader by import operations. A server can export several interfaces and a client can import a number of interfaces. The trader enables late binding and location transparency.

By using the server group concept (see the control view) ANSA also supports broadcasting. ANSA recommends retransmission and supports error-codes to handle communication failures.

Table 6.31 ANSA Communication View Mapping

MegSDF View Concept	Corresponding ANSA Concept
Communication style	Message passing
Communication primitives	Port-to-port and broadcasting
Constraints for load balancing	Not defined
Specification of legal communication	By Interface reference only
Location transparency mechanism	Trader and selective transparency
Failure handling policy	Retransmitting, error codes

Control View

The ANSA architecture supports fully distributed processing. The components interact using the client-server model. ANSA supports both synchronous and asynchronous interaction to ensure maximum concurrency. It specifies operation ordering primitives as sequential, parallel, or atomic operation; optional invocation of operations; and operations tied to external clocks. The attributes of the invocations are defined in the interfaces for the operations.

ANSA supports the concept of server groups. One can define functionally distributed, coordinated replica, and parallel replica server groups. In a functionally distributed group, each server performs some part of the requested service. In a coordinated replica one server receives the message and performs the required action while all other servers stand by. In the parallel replica group, all members perform the

same service. Each group has a coordinator which accepts requests from clients and distributes them to the members of the group.

Table 6.32 ANSA Control View Mapping

MegSDF View Concept	Corresponding ANSA Concept
Control approach	Fully distributed
Control units	Components, servers groups
Invocation approach	Client-server
Operation ordering primitives	Sequencing, serial, optional, clock based, and parallel operation

The Data View

ANSA suggests using an Interface Definition Language (IDL) as a tool that overcomes problems rooted in the heterogeneity of the environment. IDL can be considered as a meta-data-model. ANSA supports distributed systems with distributed databases. Data is stored in objects and accessed via interfaces. ANSA does not specify redundancy or consistency control mechanisms.

Table 6.33 ANSA Data View Mapping

MegSDF View Concept	Corresponding ANSA Concept
Meta-data-model	Interface Definition Language
Database organization	Distributed systems
Transactions primitives	Atomic operations
Redundancy and consistency control	Not specified

The Environment View

The only element of the environment view that ANSA specifies is the inclusion of security attributes in the interfaces.

Application Architecture

ANSA is intended to develop a platform to support systems integration of information technologies which spans many application domains; Therefore, it does not and cannot specify an application architecture, which by definition must be domain specific.

6.4.3 Examples of Projects with Software Architectures

This section describes two architectures that have been defined and used in projects for the development of systems of systems.

6.4.3.1 Ship-2000

Ship-2000 [SS2000a, b] is a project for the development of a family of integrated systems (a generic system of systems in MegSDF's terminology). The application domain of the Ship-2000 project are naval vessel systems including Naval Command, Control, and Communication (C^3)/Weapon Control Systems.

Understanding the various problems involved in development of such systems led the developers to define an architecture for the system. From our viewpoint, Ship-2000 specifies a Mega-System architecture, but the elements of the conceptual and application architecture are intermingled and not always clearly distinguished into views. The

following describes the role of a Mega-System Architecture in the Ship-2000 project and maps the Ship-2000 architecture into our concepts and views.

Conceptual Architecture

Ship-2000 distinguishes between execution and static views of the system. These views correspond to the structural and control views in MegSDF.

The Structural View

Ship-2000 systems are built from Computer Software Components (CSC). The CSCs are organized into a hierarchy of:

- Functional Areas (FA),
- System Function Groups (SFG), and
- System Functions (SF).

The uppermost layer consists of components called Functional Areas. Each Functional Area is divided into a number of intermediate components, called System Function Groups. The main role of a System Function Group is project management. It is similar to the system task in MegSDF. A System Function Group is divided into System Functions. There are, usually, one to twenty System Functions in a System Function Group. A System Function corresponds to one or a few programs (which are described in the control view).

Ship-2000 also specifies another classification of System Functions based on the level of generality of the components. It includes the following layers:

- Product dependent - for a specific system of a customer,
- Customer - for one customer for several systems,

• Equipment - a specific hardware,

• Ship Systems - special functions for ships,

• Systems Independent - fits other types of system, and

• Fundamental/base system - distributed computing environment.

To reduce dependency of Systems Functions, the architecture allows only downward dependency, i.e., elements may only use services of a lower level only. Thus, a Customer's System Function can use services of an Equipment or Ship System Function.

Table 6.34 Ship-2000 Structural View Mapping

MegSDF View Concept	Corresponding Ship-2000 Concept
Component	Functional Areas (FA), System function groups, and System Functions (SF)
Classes of components	Generality classification: Product, Customer, Equipment, etc.
Constraints for components	Using the generality classification, only downward dependency is allowed.
Guidelines and rules for decomposition	Product or functionality based

The Communication View

The hardware components of Ship-2000 are connected by a Local Area Network (LAN) which enables different communication approaches. The programs of Ship-2000 are connected by Inter Program Communication (IPC). The IPC is supported by Ada runtime system, OS, and hardware.

Messages are sent by procedure call and stored in a queue. A receiver empties its queues at its own pace. For efficiency, logical names are exchanged for physical names by a "name server" using a runtime built database where entries are created when programs register themselves to the network.

IPC provides the following communication primitives:

• Multicast - Only one message is sent; all receiving programs receive it in parallel. There is no indication of how many processors or nodes read the message. Multicast is used for high volume and conserves the network bandwidth.

• Singlecast - The sender names the receiver.

• Virtual Channel - A safer version of singlecast. It can be used for long messages. The virtual channel performs blocking, sequencing, and deblocking.

A fundamental rule reduces communication flow by requiring messages to be transferred only once. The architecture does not specify what constitutes legal communication or a communication failure policy.

Table 6.35 Ship-2000 Communication View Mapping

MegSDF View Concept	Corresponding Ship-2000 Concept
Communication style	Message passing using queues
Communication primitives	Singlecast, broadcast, virtual channel
Constraints for load balancing	High rate messages are transferred only once
Legal communication	Not defined
Location transparency mechanism	Name server that uses a runtime built database to substitute logical names with physical addresses
Failure handling policy	Not defined

The Control View

Ship-2000 execution view includes Ada programs that communicate by exchanging messages. A configuration consists of several nodes. Each node includes several processors. Each processor can run programs. Programs not linked to special hardware can migrate. Multiple instances of a specific program might be installed in the same configuration.

Programs behave as free-running entities. Each program performs a single task and is generally single-threaded. Interfacing with the message passing mechanism is implemented by an Ada generic task called whenever a message arrives. Other tasks are used inside programs when parallel processing is appropriate.

The architecture also specifies events for reporting abnormal technical states in the system. Hardware events indicate malfunctions that require repair by a technician and are

generated by background on-line test programs. Software events indicate coding or configuration errors and are not repaired by customer personnel.

The architecture specifies a specific function that starts and reconfigures the system. When a node starts, a local agent identifies itself to the controlling program. The controlling program sends the node a list of programs which are supposed to run on the node. The agent then loads all programs not already loaded.

The architecture supports both event-driven and periodic processing.

Table 6.36 Ship-2000 Control View Mapping

MegSDF View Concept	Corresponding Concept in the Architecture
Control approach	Fully distributed
Control units	Processes, threads
Invocation approach	Event driven and periodic loop
Operation ordering primitives	events, start-up procedures

The Data View

Ship-2000 defines concepts for data handling but these concepts are more application oriented than the concepts we recommend for the data-view. The project requires that all data be time-stamped as early as possible. It defines essential data components and constraints for handling them. These concepts can be considered as a meta-data-model.

The essential data components in the system includes:

- Sensors (tracking data),
- Altitude,

• Own Ship Position and velocity, and

• History Recording.

The architecture does not specify either transaction primitives or redundancy and consistency mechanisms.

Table 6.37 Ship-2000 Data View Mapping

MegSDF View Concept	Corresponding Ship-2000 Concept
A meta-data-model	Essential elements definitions and their handling
Data organization	Distributed data
Transactions primitives	Not defined
Redundancy and consistency control	Not defined

The Environment View

Ship-2000 defines concepts that corresponds to the user-interfacing and special purpose hardware elements of MegSDF's environment view. For user interfacing, Ship-2000 uses a Man Machine Interface (MMI) function to provide maximum flexibility for users, especially in environments with different customers and varying levels of expertise. The MMI manager defines a set of MMI objects. Operators can define any form of representation based on the defined MMI objects. The MMI is used to isolate the application from representation details.

The architecture specifies the following interfacing primitives:

• Graphics - to draw complex graphical objects.

• Text - to present and accept new values from operators.

• Alerts - to inform operators that something has happened that merits attention.

• Softkeys - keys drawn on a touch sensitive display device.

• Menus - to organize softkeys.

Ship-2000 is intended for real-time embedded systems. Accordingly, it specifies special purpose hardware concepts. The nodes of the system are synchronized within an accuracy of one millisecond ensured by special hardware and software. To minimize complexity and enable reuse, a common internal representation of sensor data, independent of sensor particulars, is used. The project also specifies rules for handling sensor data.

Table 6.38 Ship-2000 Environment View Mapping

MegSDF View Concept	Corresponding Ship-2000 Concept
Common user interface - presentation	Softkeys approach, flexible interfaces
Common user interface - interaction	MMI with set of interaction primitives
Special purpose hardware and external systems interfaces	Sensor handling Synchronization of the systems with 1 millisecond accuracy
Security in the system	Not defined

Application Architecture

Ship-2000 does not distinguish between a conceptual and an application architecture, but it is possible to identify elements of an application architecture. The project specifies the actual functional areas (FA), the system function groups (SFG), and System Functions (SF). The functional areas include:

- Command, Control and Communication (C^3),
- Weapon/Director,
- Fundamentals, and
- Man Machine Interface (MMI).

These functional areas roughly correspond to the cluster concept of MegSDF. There are about 30 SFG's and 200 SF's [SS2000a, b]. The documentation of the project includes general diagrams for building block interaction. The project does not specify data distribution or a service dictionary.

Table 6.39 Ship-2000 Application Architecture Mapping

MegSDF Element	Corresponding Ship-2000 Element
List of Building Blocks	Systems Functions
Clusters of BB	The list of the Functional areas
BB interaction Diagram	General interaction diagram
Data distribution map	Not defined
Service Dictionary	Not defined

6.4.3.2 ESF - FSE Reference architecture

The Eureka Software Factory (ESF) [ESF 89], [ESF 90], [SCHA 90], [HUBE 90], [ADOM 92] is an ongoing project intended for industrial software production using software factories. In ESF, a Factory Support Environment (FSE) must be able to be configured for specific industries and to evolve with technological innovation. To enable such customization and evolution the ESF uses the ESF-FSE reference architecture which is, in MegSDF terminology, a conceptual architecture for systems of systems.

The goal of the ESF - FSE architecture is to define requirements that must be met by every instance of the ESF. It consists of the ESF standards and the structure which inter-relates these standards. The reference model addresses multiple platforms, market fragmentation, and the need to adapt the systems to various customers. The architecture is a reference model for Factory Support Environments. The application domain for the ESF project is Integrated Computer Aided Software Engineering (CASE) Systems.

ESF's architecture is based on a minimal kernel with "plugable" extensions. It is a communication-oriented architecture with service-oriented building blocks.

The Conceptual Architecture

The FSE architecture is defined using structural, user, and process views. These correspond to the structural, environment, and (to some extent) the control view of the conceptual architecture recommended by MegSDF.

The Structural View

An FSE consists of a set of components connected to a Software Bus (SWB). There are two types of components: Service Components (SCs) and User Interface Components (UIC). Service components, typically, do not have a user interface. Figure 6.9 illustrates the FSE architecture. An FSE consists of a set of tools which are dynamically established and configured through bindings between user interaction components and sets of service components.

The ESF project recommends including a minimal kernel of services required by other components. Service Components which implement a functionality of the minimal kernel mechanisms are called kernel components. The kernel components can be replaced

by other components that provide the same services using different algorithms or languages.

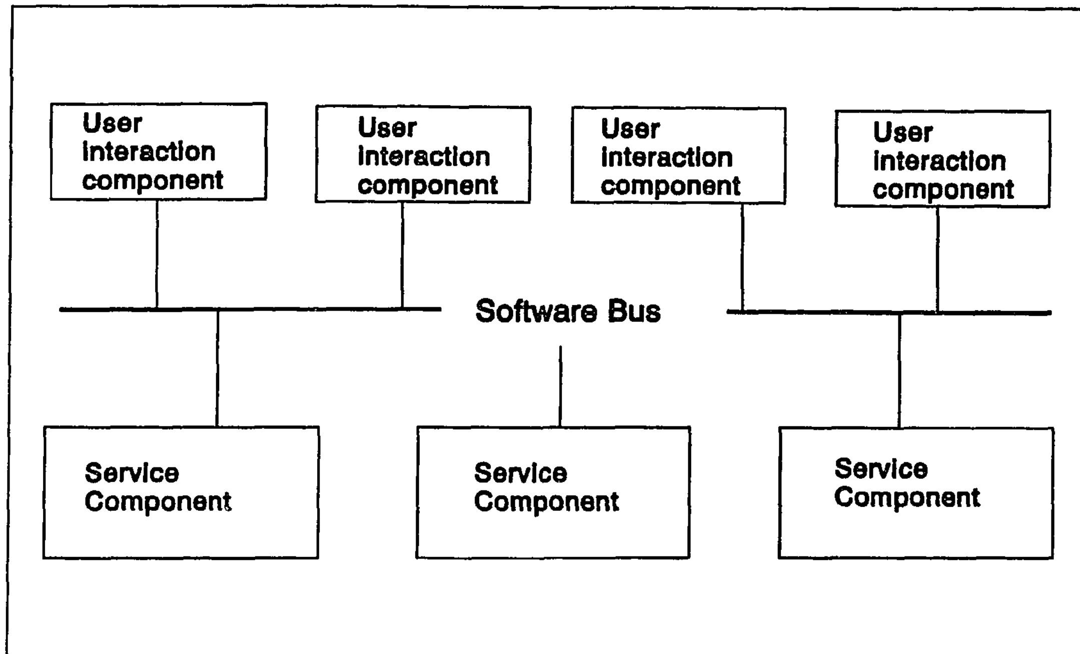

Figure 6.9 The Structural View of the ESF Architecture
(copied from ESF - Project Overview 1990 [ESF 90])

The components and the tools correspond to the component types of the MegSDF structural view. The Service, User Interface, and kernel components correspond to component classes.

Any component can be decomposed into sub-components, which can be integrated by such mechanisms as a common database or communication channel. The reference architecture, however, is not concerned with integration within sub-components.

A service component generally consists of two parts: functionality and a storage system. The storage system can be an Object Management System (OMS), file system, or traditional database. The capabilities of a service component are defined in its interface.

The ESF project proposes specifying a minimal set of capabilities every component must provide, e.g., help mechanisms.

A user interaction component presents information to users and provides editing capabilities. This component also includes code for user interaction logic.

The software bus requires a formal description of components. These descriptions, expressed using the a Component Definition Language (ESF-CDL), include the imported/exported capabilities, transfer syntax, control exchange primitives, and requirements on the actual technical platform. The use of kernel services is not specified in the descriptions.

Conformance criteria for ESF components, corresponding to MegSDF constraints on components, include:

• Use of the SoftWare Bus (SWB) primitives for all inter-component communication

• Specification of interfaces using the ESF-CDL.

• Minimal set of capabilities required to be present in every component.

The ESF does not specify rules for decomposition into components (tools).

Table 6.40 ESF Structural View Mapping

MegSDF View Concept	Corresponding ESF Architecture Concept
Component	Components, tools
Classes of components	Service, user interaction, kernel
Constraints for components	Use SWB primitives, specified by the ESF-CDL; Have the required minimal capabilities
Guidelines and rules for decomposition	Not defined

The Communication View

The ESF architecture is communication oriented. Integration of components is done by a software bus, not by a common database. The software bus is an abstract communication channel. It hides distribution aspects and allows the exchange of data without loss of structural and conceptual information. It supports the components with inter-operations and integration.

The software bus provides two principle services that hide distribution and heterogeneity:

• The plug-in mechanism - for static or dynamic binding of clients to services, and

• A communication mechanism - for exchanging control and data

ESF proposes specifying new standardized transfer syntaxes, as well as standardized means for describing new transfer syntaxes and standardized protocols. ESF does not specify communication primitives, constraints for load balancing, or failure handling policy.

Table 6.41 ESF Communication View Mapping

MegSDF View Concept	Corresponding ESF Architecture Concept
Communication style	Message passing by the SoftWare Bus (SWB)
Communication primitives	Static and dynamic binding
Constraints for load balancing	Not defined
Specification of legal communication	Not defined
Location transparency mechanism	By the SWB based on the plug-in and the communication mechanism
Failure handling policy	Not defined

The Control View

The process view of the ESF-FSE architecture includes concepts corresponding to concepts of the MegSDF control view. One of the essential features of the FSE is the programmable environment. This is supported by a kernel service component called the Factory Process Engine (FPE).

The Factory Process Engine uses process models to customize the FSE according to customer requirements. These models are described as process programs using a special Process Programming Language (PPL). A process program links organization structures, development methods, and tools suitable for supporting the various tasks of the developers. The Component of the Factory Process Engine controls the operations of the other components within the ESF.

Table 6.42 ESF Control View Mapping

MegSDF View Concept	Corresponding ESF Architecture Concept
Control approach	Distributed with a centralized control by the Factory Process Engine
Control units	Components
Invocation approach	Not specified
Operation ordering primitives	The process programs

The Data View

ESF defines a framework that allows different database systems to be accessed through common Data Definition and Data Access languages. ESF also specifies essential requirements on database systems for software engineering. The other concepts of the data view are not defined.

Table 6.43 ESF Data View Mapping

MegSDF View Concept	Corresponding ESF Architecture Concept
A meta-data-model	Data Definition language and Data Access language Essential requirements on database systems for software engineering
Database organization	Not defined
Specifications of transactions primitives	Not defined
Redundancy and consistency control	Not defined

The environment view

ESF mainly addresses the user interaction part of the MegSDF environment view. It specifies a simple paradigm for describing user interaction in the complex environment of a software factory. ESF also developed a prototype for a view server whose task is to synchronize multiple views on shared structures. ESF does not specify a common user interface, but it does specify a conceptual model for the user view of the ESF-FSE Reference Architecture using ER notation. The model includes organization, role, person, tasks, tools, etc., as entities, and their relationships. The process engines can be considered as tools that provide security mechanisms.

Table 6.44 ESF Environment View Mapping

View Concept	Corresponding Concept in the Architecture
Common user interface - presentation	Not defined
Common user interface - interaction	Conceptual model for user environment; view server
Special purpose hardware and external systems interfaces	Not defined
Security in the system	Provided by the process engines

Application Architecture

ESF does not define an application architecture. However it does recommend defining an instance of the FSE by using the structural, user, and process views. Tools can be specified by interconnecting user interaction components with a set of service components. A particular user view consists of all tools and information available to that user.

6.4.4 Classification and Comparison of Existing Architectures

This section summarizes the discussion in the previous sections by comparing the various architectures based on the concepts of Mega-System Architecture defined in section 6.2. Table 6.45 compares architectures for systems; Table 6.46 compares architectures for Mega-systems; Table 6.47 compares architectures for Mega-Systems that have been used in development efforts.

Each architecture is classified according to its application domain, the kind of system it is intended for, and the type of architecture (conceptual, application, etc). The tables map the concepts of each architecture to the views of the conceptual architecture and the elements of the application architecture.

Table 6.45 Systems Architectures

	Application Machine	**Best's Architecture**
Domain	Any domain used in automobiles systems	Data-Processing
Type of systems	Systems	Large scale systems
Classification as an Architecture	Conceptual	Conceptual
Structural view	POPs	Super-structure with drivers and procedures that provide specific functionality
Communi-cation View	Memory sharing	Memory sharing (databases)
Control View	Centralized approach (Application Program)	Autonomous functions
Data View	Not defined	Fits both centralized and distributed databases
Environment View	Sensors circuits	Electronic desks A security package
Application Architecture	Reusable library of POPs	A generic architecture
Remarks		The architecture is function or processing oriented

Table 6.46 Mega-System Architectures

	The OSCA architecture	**Network of Application Machines**	**CAN-Kingdom**	**ANSA**
Domain	Data-Processing	Any domain (used for vehicle systems)	Stationary or mobile machine systems	Information technologies
Type of Systems	Systems of systems	Generic systems of systems	Systems of systems	Systems of Systems
Classification as an Architecture	Conceptual	Conceptual	Conceptual	Conceptual
Structural view	Data, functional, and user interface building blocks	Application machines are the building blocks	Cities with a Capitol like a kingdom	Systems and applications - components
Communication View	By contracts based on infra-structure services	Client-server is a possible implementation	Message passing	Message passing and a trader
Control View	Fully distributed with Autonomous building blocks	A router controls the global operation	The Capitol governs all cities	Fully distributed
Data View	Specifies how to handle corporate data	Common database	Only detailed structure for messages	Interface Definition Language
Environment View	User view + External systems	Some definitions for hardware	Not defined	Not defined
Application Architecture	No explicit	Based on the POPs concepts	Allocation of functions to cities	Not defined
Remarks	It is a detailed conceptual Architecture	Supports a kind of application architecture	Communica-tion based architecture	Supported by an infra-structure

Table 6.47 Projects that Use Architectures

	Ship-2000	**ESF**
Domain	Naval - Military vessels	Integrated Computer Aided Software Engineering (ICASE)
Type of Systems	Generic System of systems	Generic Systems of systems
Classification as an Architecture	Mega-System architecture with elements of conceptual and application architecture	Conceptual architecture for a specific application domain
Structural view	Programs Functional Areas System Functions Groups System Functions	Services and user interaction components interconnected by the Software Bus
Communi-cation View	Message passing using queues based on Inter Program Communication	Software Bus
Control View	Fully distributed	Process Engine
Data View	Identification of essential data components and definition of their handling	Some standards for databases for ICASE
Application Architecture	A list of the FAs, SFGs and the various System Functions	No specification of an Application Architecture. Instances are formed by specifying the components used in the FSE
Remarks	The project does not explicitly distinguish between the conceptual and application architecture	The architecture is communication oriented (unlike previous systems in the same domain that used a common database)

CHAPTER 7

INFRASTRUCTURE ACQUISITION IN MegSDF

The infrastructure acquisition task is responsible for choosing, developing or purchasing, validating, and supporting an infrastructure that integrates the enabling technologies into a unified platform. MegSDF recommends the infrastructure be common to all systems developed in a domain. The infrastructure must address problems caused by the heterogeneous environments in which the systems operate and enable the incorporation of rapidly evolving technologies into the Mega-System. MegSDF recommends (re)using existing infrastructures instead of developing the infrastructure from scratch.

The process of infrastructure acquisition must specify an ***infrastructure model*** that defines the services of the infrastructure on the basis of the conceptual architecture. For manageability, we recommend dividing infrastructure functionalities into ***service groups*** corresponding to the views of the conceptual architecture and an additional group of domain specific services.

Infrastructure acquisition is a continuous task. It must consider both changes in the conceptual architecture and evolution of technologies to ensure the effectiveness of the system.

This chapter describes the infrastructure acquisition task. Section 7.1 describes the role of infrastructure acquisition in MegSDF and its required characteristics. Section 7.2 describes the underlying concepts for an infrastructure. Section 7.3 defines the process of

infrastructure acquisition. Examples of existing infrastructures are described in section 7.4.

7.1 Requirements for Infrastructure Acquisition

7.1.1 The Role of Infrastructure Acquisition

MegSDF recommends the infrastructure acquisition task as one of the activities on the Mega-System level. This task is responsible for providing an effective and operative environment that integrates all enabling technologies that support the operation and facilitate the development of a Mega-System. This section describes the role of the infrastructure in the MegSDF framework.

Infrastructure acquisition addresses the difficulties in software development described in chapter 1, focusing mainly on technology aspects. It addresses: problems caused by the existence of different technologies in heterogeneous and not always standardized environments; the need to bridge different technologies; and the necessity of incorporating over time new and emerging technologies into existing systems. These difficulties are listed below as an inverted sub-table of the problem list (table 1.1).

Systems operate in environments that consist of several technologies, e.g., communication, database management system, user interface, etc. Mega-Systems, typically, operate in heterogeneous environments that may include several types of communications, a number of database management-systems, different tools for user

interfacing, etc. Thus, the infrastructure must support the coexistence of various technologies, and bridge and resolve the differences among them.

Table 7.1 Difficulties and Problems Addressed by the Infrastructure

Difficulties	Caused By	Aspect	Problems
Heterogeneous environment	More than one system	Technology	There is a need to bridge the various technologies and efficiently incorporate emerging technologies as a common domain-wide solution
Each development group has to struggle independently with Heterogeneity and dynamic environments	More than one developer		
Bridging different technologies and incorporation of new technologies is required	Heterogeneous environment		
Customization to user environment	More than one customer		
Dynamic environment requires incorporation of new technologies	Longer life cycle		

Technologies emerge and evolve rapidly. Since Mega-Systems have long life cycles, these technologies must be incorporated to ensure effectiveness. Infrastructure acquisition must evaluate new technologies and efficiently incorporate them into the existing infrastructure.

An infrastructure standardizes the way in which different technologies are used in a domain. It is acquired as a common, unique solution for bridging and handling technologies for the constituent systems. It is intended to provide complex, compound services and commonly needed functionalities for the domain applications that cannot be

found in typical operating systems, communication tools, or database management systems. The infrastructure also promotes portability of systems.

The infrastructure is used by the various developers of systems in the domain. As a common solution it reduces the effort required to deal with technologies, in contrast to solutions where every group develops its own limited solution. It enhances the uniformity and integratability of the systems.

In MegSDF, the infrastructure serves as a platform of unified services primarily during the implementation phases of the various systems tasks (projects). The infrastructure acquisition process uses the conceptual architecture as its main input, implementing the concepts specified in the conceptual architecture and supporting transparency. The conceptual architecture is the bridge between the infrastructure and the domain. It represents the domain needs to the infrastructure. Feedback from the infrastructure acquisition task is used to improve the conceptual architecture. Existing infrastructures and projected technologies are used as inputs for the Mega-System Architecture design task and guarantee the conceptual architecture will be feasible. In this role the infrastructure represents the technology aspects. The relationship of the infrastructure to the other elements of MegSDF is illustrated in Figure 7.1.

It is important to differentiate between MegSDF's infrastructure and the type of infrastructure that is proposed as part of domain analysis for reuse [ARAN 91]. MegSDF's infrastructure integrates enabling technologies that support the operation and facilitate the implementation of the Mega-Systems into a common solution used by all developers of the Mega-System. It may include communication, database, user interfaces and CASE

tools. The infrastructure as recommended in [ARAN 91] is solely to support the reuse process, i.e., facilitate classification, storage, and retrieval of reusable components. Thus, the infrastructure for reuse might be one of the CASE tools integrated into MegSDF's infrastructure.

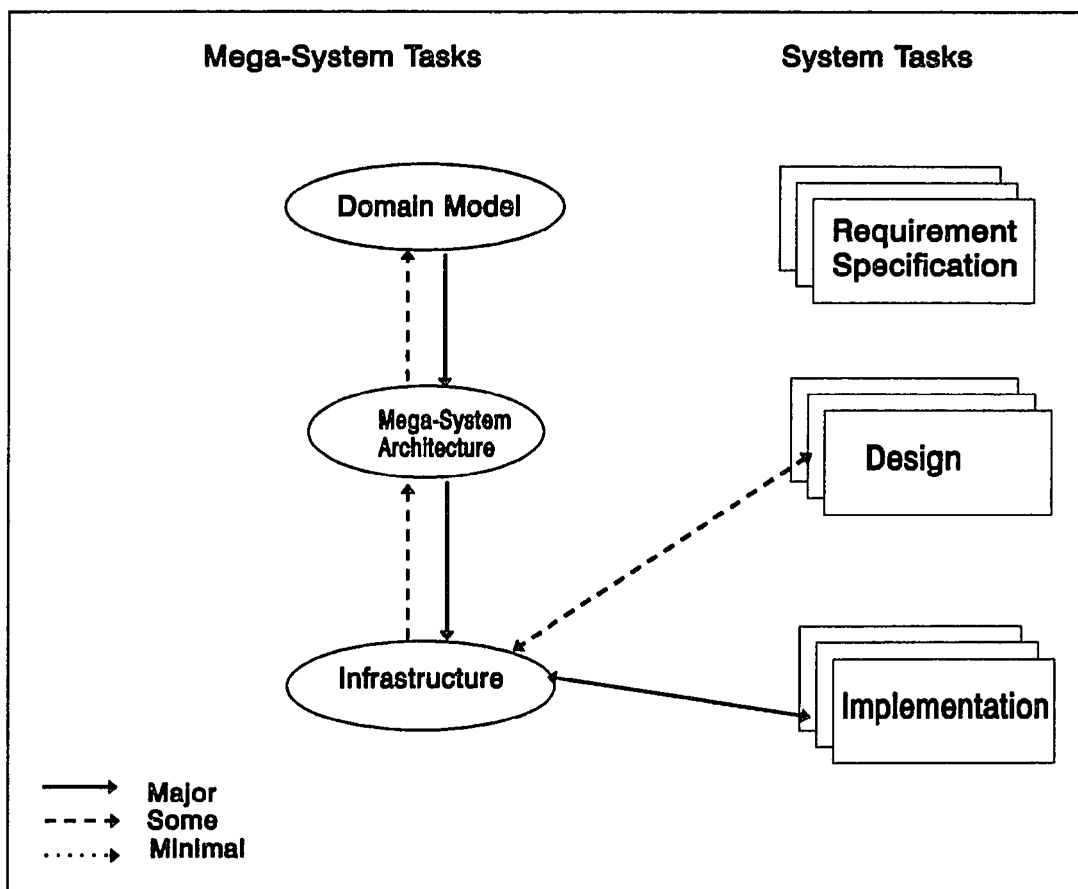

Figure 7.1 Relation of Infrastructure to other MegSDF Components

Though MegSDF infrastructure serves as a common basis for implementation of Mega-Systems in the domain, we recommend that the infrastructure not be developed by the developers as part of MegSDF process itself. An infrastructure really belongs to the technology aspect and should be developed by technology developers as an integrated set

of tools that can be used for different application domains. The Mega-System's developers should focus on developing the application, not developing technology.

7.1.2 Requirements for an Infrastructure

An infrastructure integrates enabling technologies for the development and execution of Mega-Systems in a domain. Its goal is to ensure that the systems developed in the infrastructure environment are open, in the sense that they are integratable, extendable and scalable.

To provide these characteristics an infrastructure should meet domain needs and be:

- Open,
- Service preservative,
- Reliable,
- Efficient, and
- Easy to use.

The infrastructure must meet the domain needs. Although using the same infrastructure in different domains is possible, the infrastructure must support the necessities of the domain represented by the conceptual architecture (see also section 6.2.2). The infrastructure should also include domain specific utilities and tools used by systems in the domain and not supported by an enabling technology. The process engine of ESF [ESF 90] for the CASE domain, and special indexing mechanisms in the library domain, are examples of such tools and utilities.

The infrastructure must also be ***open*** in the sense that it can incorporate new technologies to the infrastructure simply and with minimal effort. It should also be easy to integrate the infrastructure with other infrastructures or expand the infrastructure to other hardware platforms.

An infrastructure should be ***service preservative***. This means that new versions of the infrastructure must support services that were provided by earlier versions. This is important since it would be inefficient to modify all systems whenever the infrastructure was changed (see also [OSCA 92]).

Since the infrastructure is an active part of the Mega-System and enables the operation of the systems in the domain it must be ***reliable***. A failure of the infrastructure degrades the operation and limits the availability of the entire Mega-System. The infrastructure should provide services that will ensure that the system will remain consistent and secure, e.g., atomic operations (transactions) and mechanisms for protecting resources and information.

The infrastructure must execute ***efficiently*** to compensate for the negative effects of using the infrastructure services instead of local programming solutions. Thus, additional execution time and memory space required for using the infrastructure at operation time should be minimal.

Since the infrastructure is used by developers located in different sites, it should be ***easy to use***, simple to understand, and well documented. The infrastructure should also save development efforts in the implementation phases. It must be supported by development and debugging tools that improve developer's transparency and productivity.

We recommend that the infrastructure be implemented according to the conceptual architecture and by using the infrastructure services in a bootstrapping fashion. For example, a distributed database should use the communication channel of the infrastructure. The communication channel, on the other hand, will use the concepts of the data view. This helps ensure system uniformity and avoids conflicts that would be caused by implementing multiple solutions.

7.2 An Infrastructure

7.2.1 MegSDF Infrastructure

The infrastructure in MegSDF implements the conceptual architecture specified by the Mega-System Architecture task. While the conceptual architecture defines strategies and concepts for implementation and is technology independent, the infrastructure integrates the various technologies that implement and support these concepts.

We recommend implementing the infrastructure by a service-based approach, that is, the infrastructure is defined as a set of services where each service corresponds to a capability of the infrastructure that provides common, business independent functionality (as defined by OSCA [OSCA 92]) for the systems developed in the domain, e.g., message transfer and window presentation. It is important to distinguish between operating system services and infrastructure. The functionalities of the infrastructure services are more

sophisticated than those provided by typical operating systems [NIST 91]. An operating system service is usually seen through the "filter" of the infrastructure.

A concept of the conceptual architecture might be implemented by several services and a service might support several concepts. Services will be used by the programmers and provide the means for implementing systems according to the conceptual architecture.

The services form a layer between programs and actual technologies, abstracting implementation details. This layer makes it possible to port the systems and to use the same software with different environments and platforms, provided the services are supported by the environment. Under a service based approach, the infrastructure can be extended simply by adding new services. Furthermore, infrastructures can be integrated by mapping between the services of the infrastructures and using adaptors when required.

We identify three types of infrastructure services:

- Application Services - Services that are used by the programmers within the application to perform required functionality, e.g., message passing, and presentation of a window.
- Background and Administration services - Services that support the operation of the system but are not used by the programmers within their applications. These services might monitor, control, and be an active part of the operation of the system. Examples are administration services, services that support the consistency and integrity of the systems, or services that measure communication load.
- Tools services - Services provided by CASE tools to support the development of systems according to the conceptual architecture. Examples of tool services are compilers that provide developer's transparency, design tools that suggest design constructs, e.g.,

communication primitives, or analysis tools that support a strategy for decomposition of systems.

A concept might be supported by all types of services or be implemented as an application or background service depending on the implementation strategy and the relation between the CASE-tools and the infrastructure. For example, a trader might be implemented as a background service and be used implicitly by all programs for inter-communication; or it could be implemented as an application service where programs explicitly declare the services they provide or want to use and obtain interface references from a trader in order to communicate.

The definition of the infrastructure should include a mapping between the concepts in the various views of the conceptual architecture and the services of the infrastructure. This mapping can be considered as a model of the infrastructure.

The services themselves are implemented by various technologies. Therefore, we propose that the infrastructure definition also includes a mapping between services and enabling technologies. One approach is to realize the services of the infrastructure as a library of subroutines. Programmers invoke these services by subroutine calls within their programs. An alternative approach is to develop a language that includes all services as built-in primitives. We recommend using the first approach since it does not restrict the use of the infrastructure to one language and is more extendable.

The selection of an appropriate infrastructure should be based on international or commercial standards, e.g., ISO/OSI [ROSE 89], SAA [MART 91], etc. The use of standards reduces wasted effort in developing solutions that already exist; improves the

extendibility of the system; guarantees the support of these products; and improves the competitiveness of the Mega-System.

7.2.2 An Infrastructure Model

Our model of an infrastructure is based on the outline for a conceptual architecture in chapter 6.2. We group related services into service groups corresponding to the views of the conceptual architecture: structural, communication, control, data, and environment service groups. We add a domain dependent service group for services that do not fit any of the preceding categories. Our model is limited in scope and provides a check list. The crucial point is that the infrastructure model should correspond to the conceptual architecture.

7.2.2.1 The Structural Service Group

The structural view of the conceptual architecture in section 6.2.2.1 defines component types and classes, constraints for components, and a guideline for decomposition. The infrastructure can support these concepts by application, background, and tool services. Tool services can facilitate defining components using templates and pre-compilers, as done by ANSAware[4] [ANSA 92a], [ANSA 92b]. The other concepts are design guidelines and do not require support by application services. However, it is possible to support these concepts either by tool services that statically enforce constraints for the component, or by background services that dynamically enforce constraints. The structural service group

[4] ANSAware is a trademark of Architecture Project Management

should also include registration and configuration management services for the components. Table 7.2 maps the structural view concepts into the corresponding services.

Table 7.2 Mapping of the Structural View Concepts into Services

Concept	Service
Definition of component types	Support of the various components, e.g., templates and pre-compilers; Registration and configuration management
Specifications of classes of elements	Design constructs in design tools
Specifications of constraints for components	Dynamic or static verification mechanisms in design tools and background services
Guidelines and rules for decomposition of an application into components	Verification mechanisms in design and analysis tools and background services

7.2.2.2 The Communication Service Group

The communication view of the conceptual architecture, specified in section 6.2.2.2, defines communication style, communication primitives, constraints for load balancing, specifications of legal communication, a location transparency mechanism, and communication failure handling. The services provided by the communication view depend on the actual communication style. Different types of message passing services will support the communication primitives, enable interconnections of elements, and realize the communication failure policy as application services. These services can be supported by background services, e.g., the trader of ANSAware [ANSA 92a] that provides location transparency services. The communication view concepts can also be

supported by tools that provide design constructs and static verification mechanisms. Table 7.3 maps the communication view concepts into the corresponding services.

Table 7.3 Mapping of the Communication View Concepts into Services

Concept	**Service**
Communication primitives	Message passing, e.g., broadcasting, virtual channels by application and background services; design constructs in tools.
Constraints for load balancing	Policy enforcement; capacity measurement.
Specification of legal communication	Verification services by tools and background services, e.g., traders
Specification of a location transparency mechanism	Location transparency mechanisms, e.g., logical to physical address trading, etc., by background services
Communication failure handling policy	Message passing services that implement time out, error corrections, etc.

7.2.2.3 The Control Group

The control view of the conceptual architecture, specified in section 6.2.2.3, defines the control approach, the control units, the invocation approach, and operation ordering primitives. The infrastructure should support the control and invocation approaches with its services. It should support the various types of control units, e.g., threads, processes, clusters, and operation ordering primitives, e.g., atomic operations and clocked operations, by application services. For example, the ANSAware supports processes provides a special coroutine package for operating systems that do not have multi-processing [ANSA 92a], [ANSA 92b]. These concepts can also be supported by tool services that include

threads and processes, synchronous and asynchronous invocation, and events as design constructs. Table 7.4 maps the control view concepts into their corresponding services.

Table 7.4 Mapping of the Control View Concepts into Services

Concept	Service
Control units	Support creation, suspension, termination, etc. for threads, processes and clusters
Invocation approach	Support for synchronous and asynchronous processing
Operation ordering primitives	Atomic operations, clocked-operations, etc.

7.2.2.4 The Data Service Group

The data view of the conceptual architecture, specified in section 6.2.2.4, defines a meta-data-model, specification of transaction primitives, and redundancy and consistency control. The infrastructure should support the meta-data model by providing interfaces to and from the meta-data model by both application and CASE tools services. It should also support schemas definition. The infrastructure should support the organization of the data, i.e., a common or distributed database, by providing appropriate services, e.g., servers for distribution, and common database services, e.g., recovery, backups, on-line queries, data compression, encryption, etc. The infrastructure can support atomic transactions by both application and background services, e.g., recovery mechanisms. It can also provide background replication management services. Table 7.5 maps data view concepts into their corresponding services.

Table 7.5 Mapping of the Data View Concepts into Services

Concept	Service
A meta-data-model	Interfacing services, schemas definitions
Organization of the database	Distributed transaction handlers, backup, on-line query, etc.
Specifications of transactions primitives	Atomic transaction
Redundancy and consistency control	Replication management, recovery mechanisms

7.2.2.5 The Environment Service Group

The environment view of the conceptual architecture, specified in section 6.2.2.5, defines a common user interface, special purpose hardware and other systems interfaces, and a security strategy. Accordingly, the infrastructure must support user interfacing by providing presentation and interaction services, e.g., multi-windows, pulldowns, soft-keys, alarms, scroll-bars, emphasis, selection by cursor, typing letters, or mnemonics, mouse, interaction, etc., as suggested by SAA [MART 91]. Interfacing with hardware and other software systems might be supported by encapsulating services that translate external interactions to interactions supported by the system as suggested by [OSCA 92]. The infrastructure should support the security strategy by providing tools to define security privileges for users. It must also provide services for security enforcement as suggested by [NIST 91] and re-authentication of the user and the invoking building blocks for restricted services as suggested by [OSCA 92]. Table 7.6 maps the environment concepts into the corresponding infrastructure services.

Table 7.6 Mapping of the Environment View Concepts to Services

Concept	Service
User interfacing presentation and interaction	Multi-windows, pulldowns, soft-keys, alarms, scroll-bars, emphasis, selection by cursor, typing letters, or mnemonics, mouse, interaction, etc
Strategy for special purpose hardware and external systems interfaces	Special hardware interfacing services, and encapsulating services for interaction with external systems
Strategy to ensure security in the system	Tools to define security privilege and services for security enforcement, and re-authentication

7.2.2.6 Domain Specific Service Group

This group does not correspond to a specific view of the conceptual architecture. We suggest including in this group services that are domain specific and do not fall into any of the categories of the previous groups, e.g., the process engine services of the ESF [ESF 90] which support the various tools of the Factory Support Environment. This group might include application, background, and CASE-tools services.

7.3 The Infrastructure Acquisition Process

The infrastructure acquisition process is defined using the format introduced in section 4.1.

7.3.1 Purpose

The purpose of the infrastructure acquisition process is to choose, develop or purchase, validate, and maintain an infrastructure for the Mega-System.

7.3.2 Interfaces

Inputs

- Conceptual Architecture - The conceptual architecture, defined in section 6.2.2.
- Existing infrastructures and projected technologies - Infrastructures that integrate extant enabling technologies and which are expected to facilitate integrating prospective technologies.
- Customers/users requirements - Requirements of the customers/users for the systems.
- Feedback - Engineering information from the system tasks (projects) and the Mega-System synthesis tasks, including recommendations for improvements and corrections to the current infrastructure.

Control Input

- Management Control - The schedule to the task assigned by the meta-management task.

Circumstance Inputs

- International and commercial standards - Standards developed by international/commercial organizations to uniformize systems and tools used for their implementation.

Outputs

- Infrastructure - The chosen infrastructure of the domain, described in section 7.2.

• Feedback - Feedback from the task to the Mega-System architecture design task and the meta-management task.

7.3.3 Processing

A model of the infrastructure, consisting of groups and the necessary services, based on the conceptual architecture, is first defined. Existing infrastructures and projected technologies are then evaluated. If an appropriate infrastructure is found, it is recommended as the chosen infrastructure. Otherwise, either an existing infrastructure is used as a base and additional services are developed and integrated to it, or a new infrastructure is developed. The chosen infrastructure must be verified and validated against the model and the conceptual architecture to ensure it provides the required services. Figure 7.2 illustrates the infrastructure acquisition process.

7.3.4 Timing

Infrastructure acquisition and adaptation is an ongoing process which must be active as long as the Mega-System is developed and maintained. Domains evolve over time and new technologies emerge, so, it is necessary to consider both changes in the conceptual architecture and new technologies to maintain the effectiveness of the infrastructure. Based on these changes, appropriate technologies should be incorporated to the infrastructure.

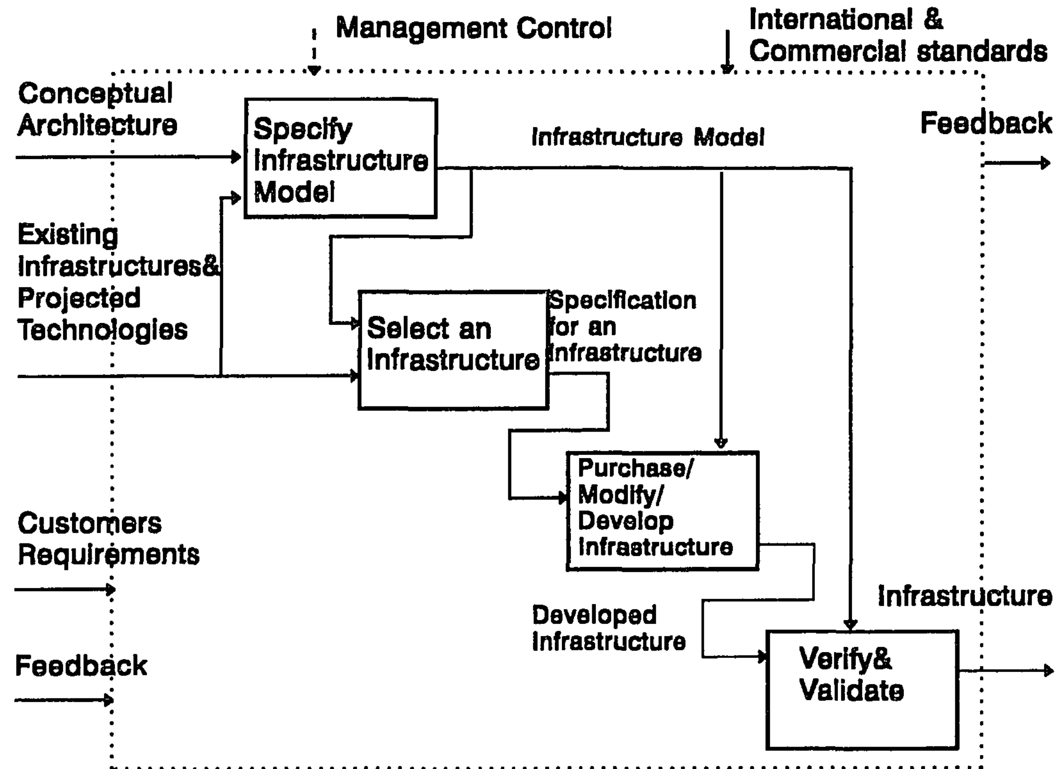

Figure 7.2 The Infrastructure Acquisition Process

7.4 Examples of Existing Infrastructures

This section discusses an infrastructure model and examples of existing infrastructures. Most the existing infrastructures implement only a limited set of services, belonging to some of the groups we specified in the outline of an infrastructure model. These

infrastructures are general and can be used for different application domains. We discuss the infrastructures with an emphasis on elements related to MegSDF.

7.4.1 The NIST Reference Model

The Reference Model for Frameworks of Software Engineering Environments was developed by the National Institute of Standards and Technology (NIST) and the European Computer Manufactures Association (ECMA) [NIST 91]. It is intended to provide a reference model for describing Software Engineering Environment (SEE) and for comparing Existing SEEs or components of SEEs. The model includes only specifications, not implementations.

7.4.1.1 NIST's Reference Model Concepts

In the NIST model, a SEE consists of several tools for developing software and a framework to support these tools. Tools are used by software engineers in different phases of the life cycle of systems. A framework, according to the NIST model, consists of a fixed set of infrastructure capabilities which provide support for objects, processes, and user interfaces, and facilitates the developing tools. The framework can also facilitate porting software development environments across a variety of hardware configurations and operating systems. The SEE tools use services of the framework and other tools. Framework components can use services provided by other components of the framework.

The framework is divided into functional elements called services. Interrelated services are grouped as following:

• Object Management includes the definition, storage, maintenance, management and access of object entities and the relationships among them, e.g., data transaction, archive and backup services.

• Process Management supports the definition of a process model for the development life cycle, enactment of a process, control and resource management, e.g., process definition, process enactment services.

• Communication Services provide a standard communication mechanism which can be used for inter-tool and inter-service communication, e.g., message passing.

• User Interface Services support the interaction of the users with the various tools, e.g., sessions, application interfaces, user assistance services.

• Tool services that support tools by additional functionality, e.g., editing, compiling, testing, analyzing.

• Policy enforcement services that support security, integrity monitoring, and configuration management.

• Framework administration and configuration management services that support management of the SEE and self-configuration-control.

[NIST 91] includes a detailed list of services. Each service is defined for different dimensions, e.g., conceptual, operations, rules. Figure 7.3 illustrates the reference architecture and its various parts and service groups.

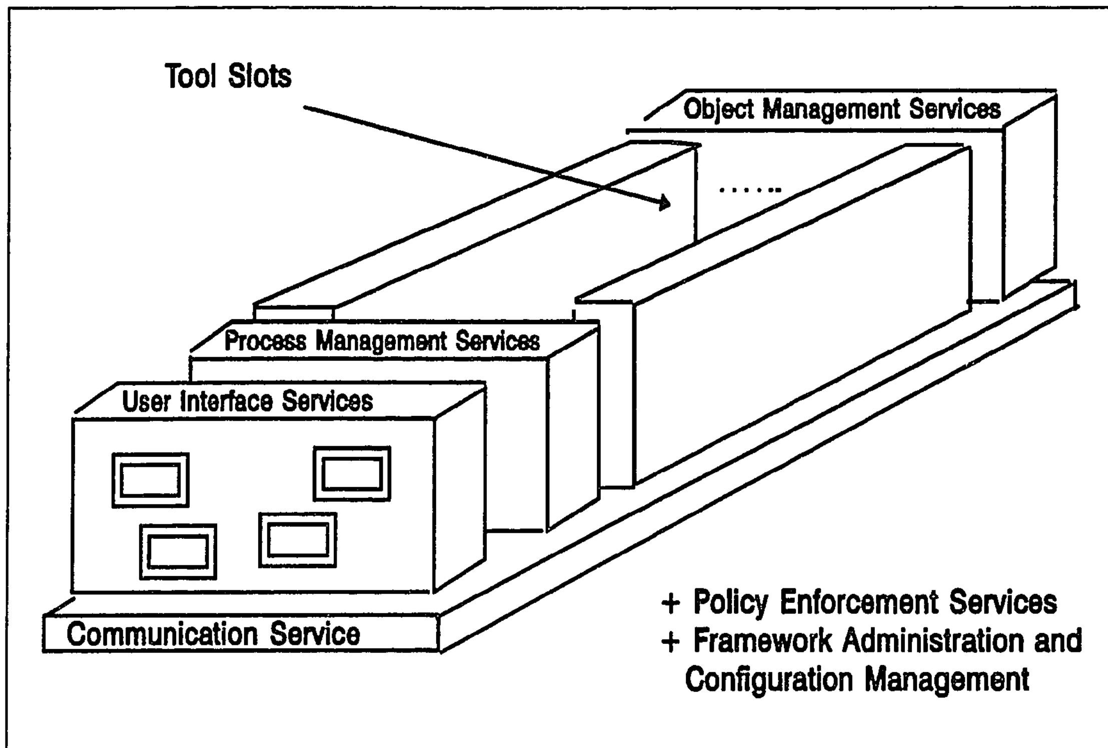

Figure 7.3 The NIST Reference Model (Copied from [NIST 91])

7.4.1.2 Mapping NIST's Reference Model to MegSDF Concepts

Software Engineering Environments (SEE) integrate tools that support the development life cycle and are used by heterogeneous groups of users. In MegSDF terminology, SEEs are Mega-Systems of the systems of systems kind. The NIST's reference model can be considered as a model for the infrastructure for these Mega-Systems. Thus, from our viewpoint, the NIST Reference model is a comprehensive model of an infrastructure that can be used for various domains. The NIST reference model seems to have been developed as a post-facto attempt to standardize existing SEEs rather than as a domain model.

The object management and communication service groups correspond to the data and communication service groups of MegSDF. The user interface service group corresponds to the user interface part of the MegSDF environment service group. The process management, policy enforcement, and tool service groups are domain dependent. NIST's model does not explicitly specify a group of control services. However, parts of the object and process management service groups provide services that belong to the control group, e.g., atomic transactions and process enactment. The NIST's Model does not explicitly specify structural services. However, the Framework Administration does provide services that can be considered as structural services, e.g., tool and resource registration. Table 7.7 compares NIST's Reference Model service groups to MegSDF groups.

Table 7.7 Comparison of MegSDF Views and NIST Service Groups

MegSDF Service Group	**NIST Service Groups**
Structure	Framework Administration and Configuration
Communication	Communication
Control	Parts of Object Management, e.g., data transactions and parts of Process Management, e.g., process enactment
Data	Object Management
Environment	User Interface
Domain Specific and development tools	Process Management Policy Enforcement Tool Services

7.4.2 ANSAware

This section describes ANSAware [ANSA 92a], [ANSA 92b] which, in MegSDF terminology, is an infrastructure that implements only part of the ANSA architecture (see section 6.4.2.4) focusing on the ANSA engineering and computational views. ANSAware supports multi-vendor environments.

7.4.2.1 ANSAware Concepts

ANSAware operates on UNIX, VMS, and MS-DOS. It provides a uniform view of a multi-vendor world, allowing systems builders to link together distributed components into network-wide applications.

ANSAware consists of a suite of software for building open distributed processing systems providing a basic platform as well as software development support, e.g., program generators and system management applications. It operates within a host to provide a unified platform. ANSAware is a service based infrastructure that supports service based applications. It supports an object-based style using the client/server approach.

ANSAware divides its engineering model into nodes, where a node may be a single computer, a process or virtual machine, or a network of computers managed by a distributed operating system. The resources of each node are managed by a nucleus which assigns them to capsules.

The capsule is the unit of autonomous operation within ANSAware. Each capsule represents a separate address space. In a multi-tasking environment, a capsule is a process. A capsule consists of several engineering objects. Each engineering object is composed

of several computational objects which are bound together at compile time and interact via local procedure calls. A computational object may have several interfaces, each offering the same or different sets of operations. A compiled computational object is an engineering object. A service is a program composed of several computational objects. A program can be compiled as a single unit or its computational object can be compiled separately; in either case the result is a set of engineering objects.

An engineering object is the smallest unit in ANSAware which is distributed, activated, deactivated, and migrated. The programmer decides how many engineering objects are merged into one capsule. Engineering objects interact with one another through the nucleus. Transparency services are added to a capsule. These services manage the nucleus-provided resources in a capsule and communicate with transparency services in other capsules to provide the required transparency. An engineering capsule may have several transparency services, and one transparency service may depend upon another. Figure 7.4 represents the relationships between ANSAware elements.

The current release of ANSAware supports access and location transparency services. Access transparency masks differences in data representation. Location transparency translates interface reference (logical address) to address resolution (physical address). Future releases will support other transparencies.

The nucleus includes a service definition for the protocol required for communication between nuclei. The protocol is based on three service layers: session, execution, and message passing. The nucleus provides services called tasks, threads, eventcounts, sequencers, sockets, plugs, channels, sessions, and interface references.

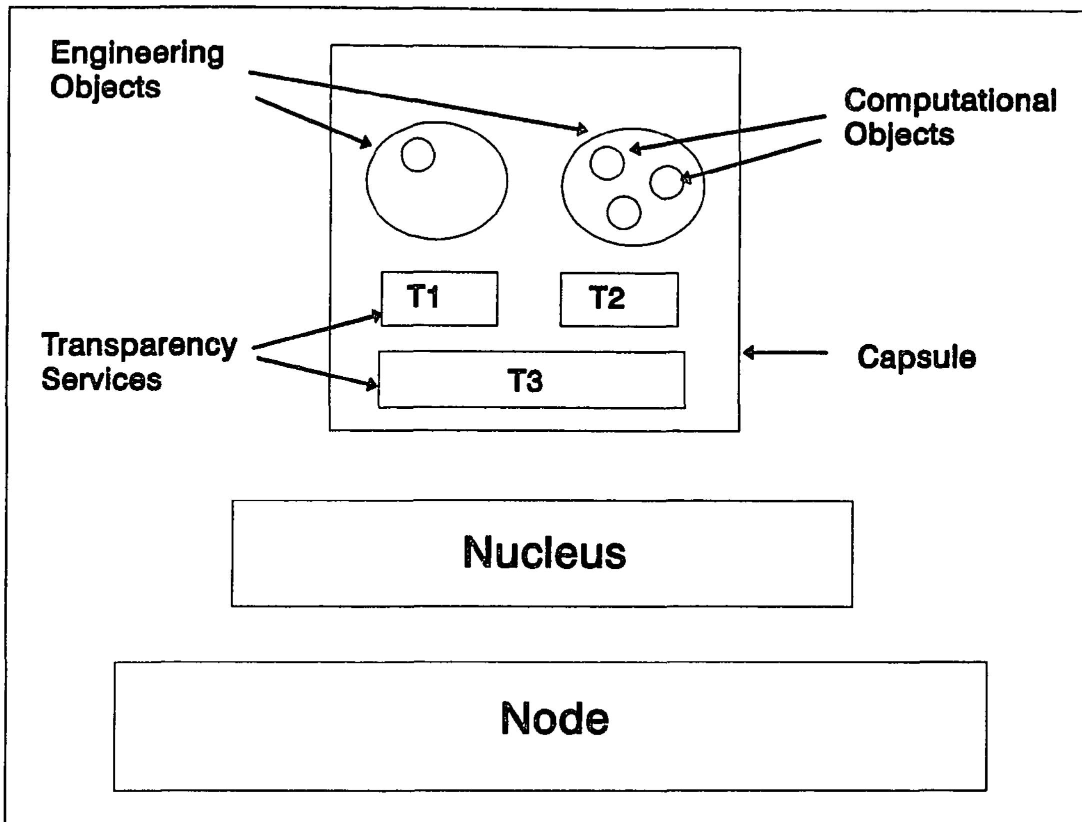

Figure 7.4 ANSAware's Capsule and Nucleus (Copied from [ANSA 92b])

A thread is an independent execution path through a sequence of operations within a capsule. Threads share data structures and can synchronize with each other at significant points. A task is a virtual processor which provides a thread with the resources it requires. The number of tasks within a capsule determines the degree of parallelism in the capsule's execution. A thread is bound to a task until the thread terminates. ANSAware includes a coroutine package to support multi-tasking in operating systems that do not include multi-tasking. Eventcounts and sequencers are used for synchronization between threads.

A socket is the unit of addressing for inter-capsule invocations. A registration operation allows a socket to be published, and thus be made accessible to clients outside the capsule. All communications are targeted to sockets. A plug is the access point for the

client of an interface. Inter-capsule operations are invoked by plugs. Each plug is bound to a corresponding socket. The path from the plug to the socket is called a channel. A socket represents the server end of an interface, whereas a plug is associated with the client end.

ANSAware specifies the interface reference for identifying interface instances to connect clients to servers. The interface references are created by the binder service in each capsule. Before a capsule can obtain an interface reference for any external service, it must obtain an interface reference to the trader. This interface reference is furnished to each capsule.

A computational object is transformed into an engineering object by two compilers. The first compiler provides transparency services. The second compiler provides interaction services. The interfaces to computational objects are defined by an Interface Definition Language (IDL).

ANSAware supports traders, factories, and node managers as network-wide management services. The trading services allow engineering objects to register the services they provide and to look for services they intend to use. The trading services also support dynamic binding. Factory services support dynamic creation of engineering objects.

7.4.2.2 Mapping of the ANSAware Reference Model to MegSDF's Concepts

In our terminology ANSAware is an infrastructure that implements only parts of the ANSA architecture. Although it does not specify service groups within the infrastructure,

it is possible to map the ANSAware services to MegSDF infrastructure service groups. ANSAware does implement some of the structural view concepts and supports components of different granularity: computational objects, engineering objects, capsules, and nodes. ANSAware is based on message passing and provides a trader for transparency services. ANSAware supports various types of control units, e.g., thread and task, and uses eventcounts and sequencers for synchronization. The data view includes only an Interface Definition Language. ANSAware does not support any environment view concepts, but it does provide development tools that enable access transparency and using ANSAware services in an embedded format. Table 7.8 summarizes this discussion.

Table 7.8 Comparison of MegSDF views and ANSAware services.

MegSDF Service Group	ANSAware Services
Structural	Capsule, Nucleus
Communication	Within an engineering object - local communication. Inter-capsules by the nucleus, traders, interface references, sockets and plugs Transparency services by pre-compilers. Interfaces are defined by IDL.
Control	Threads, tasks, eventcounts, sequencers
Data	Interface Definition Language for interfacing different data types representation
Environment	Not defined
Domain Specific	Not defined

7.4.3 IBM's Systems Application Architecture (SAA)

This section describes IBM's Systems Application Architecture (SAA) [MART 91]. It was developed by a vendor and supports a wide range of products of this vendor.

7.4.3.1 SAA Concepts

SAA was developed by IBM in an attempt to bring coherence to IBM's wide range of products. IBM's product line includes an assortment of different and incompatible hardware, multiple operating systems, and an assortment of software systems used in different operating environments. This complexity is a hindrance to IBM and its customers, who use computing systems ranging from personal computers to large systems. IBM people decided that developing such an architecture is essential to its corporate viability.

Support for SAA will be provided across a wide range of hardware and software by IBM and other vendors. IBM has committed broad support for SAA across future offerings operating in the systems software environments, including Multiple Virtual Storage (MVS), Virtual Machine (VM), Operating System 400 (OS/400), Operating System/2 Extended Edition (OS/2 EE), and other environments to be supported in the future.

An application developed according to SAA specifications must operate consistently across all SAA-supported environments. This means that it should be possible to compile and run an SAA application in any supported environment without extensive reprogramming. An SAA application's interface should appear the same, regardless of the

environment in which it runs. An SAA application should be able to communicate with other SAA applications running in any of the environments.

The foundations of each SAA hardware family and operating system (environment) is provided by three types of products: application enablers, communication subsystems, and control programs. Application enablers include programming languages, CASE tools, application generators, database management systems, and data presentation and dialog management services. Communication subsystems allow a computing system to communicate with its own peripheral devices and with other computing systems attached to a computer network. System control programs include the operating system (and its extensions) that controls a computing system in a specific hardware environment. The products that constitute the software foundation vary with each environment.

SAA standardizes three types of interfaces on top of each environment:

• Common User Access (CUA) interface provides end users a consistent view of their different applications. This promotes user productivity and reduces the time to learn new applications. The Common User Access (CUA) is a set of rules and guidelines for presentation and user interaction, e.g., organization of panels, windows, use of colors, icons and standard actions.

• Common Programming Interface (CPI) defines a set of languages and services for application developers consistent across the different environments, both in terms of how they are used and in the results they produce. This promotes portability of the systems. CPI defines a set of languages and programming services that application developers can use in developing SAA applications. These services include communication, database,

query, dialog, and presentation interfaces. Though the Common Programming Interface includes services common to all environments, it does not include features specific to an environment or an operating system, e.g., job control language.

• Common Communication Support provides consistent methods for exchanging data across a network. It consist of a set of protocols, services, and standardized data stream formats that can be used to interconnect applications, systems, and networks.

Besides defining the three interfaces and providing system software support for those interfaces, IBM also intends to develop applications that conform to the SAA standards and guidelines. These applications will be consistent and usable across all of IBM's major computing environments. IBM's OfficeVision[5] Product Family, an integrated set of applications that provide extensive office automation services, is an example that conforms to the SAA architecture. Other vendors have announced their support of the SAA standard interfaces and their intention to develop applications that will run on all SAA-supported computing environments. Figure 7.5 illustrates the components of the SAA.

Most of the services defined by the SAA are based on standards, e.g., the ISO/OSI, the American National Standard Database Language - SQL, etc. SAA also supports both SNA and ISO protocols to ensure its openness. In response to market demands for UNIX as the environment of choice for programmable workstations, IBM developed the AIX, its version of UNIX. AIX is compatible with SAA's communication and programming interfaces.

---

[5] OfficeVision is a trademark of IBM, Inc.

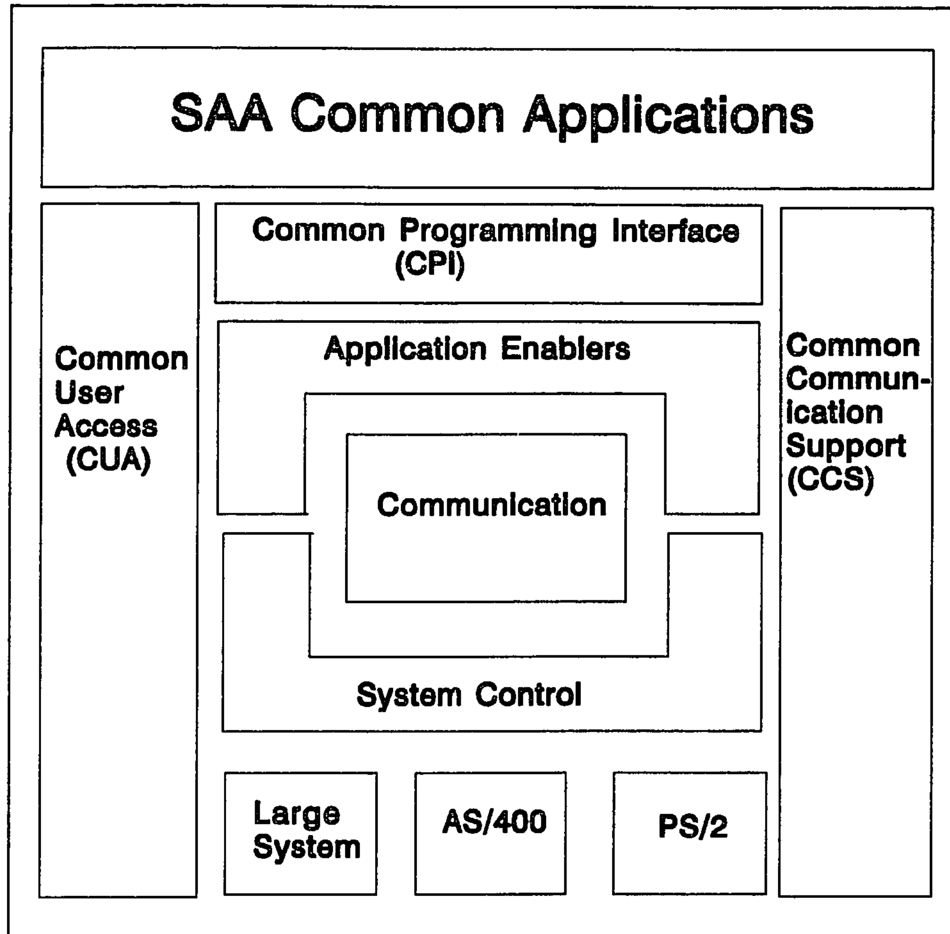

Figure 7.5 The Elements of IBM's SAA
(copied from [MART 91])

7.4.3.2 Mapping of SAA to MegSDF's Concepts

SAA is intended to provide a layer between technologies and applications, to promote the portability of software systems, and to support communication between processes. The architecture seems to have been defined in a bottom-up manner, driven by technologies and not application needs. SAA standardizes the way the technologies are used. In MegSDF terminology, the SAA concepts can be considered as a model for an

infrastructure, since it defines interfaces to technologies, but not a conceptual architecture, since it does not provide design concepts for the applications, e.g., decomposition guidelines.

SAA does not explicitly specify any structural concepts. Accordingly, its infrastructures (support environments) do not explicitly support components of the structural view. The Common Communication Support implements concepts of the communication view. SAA supports Common User Access and defines communication with other systems based on SNA and ISO. These elements belong to the environment view. Parts of the Common Programming Interfaces can be mapped to the control and data views. Table 7.9 compares SAA services to MegSDF's infrastructure service groups.

Table 7.9 A Comparison of MegSDF views and SAA Service Groups

MegSDF Service Group	**SAA Services**
Structural	Not defined
Communication	Common Communication Interface
Control	Parts of the Common Programming Interface, i.e., operating systems
Data	Parts of the Common Programming Interface, i.e., database management systems and the Common Communication Support, i.e., objects, data streams
Environment	Common User Access for user interfacing; Common Communication Support for interfacing with other systems
Domain Specific	Not defined

CHAPTER 8

THE META-MANAGEMENT, SYSTEM, AND MEGA-SYSTEM SYNTHESIS TASKS

This chapter defines the Meta-Management task, the System task, and the Mega-System Synthesis task. These are extensions of existing tasks in traditional systems development. Our discussion emphasizes how they are incorporated into the MegSDF process model and how the elements of these tasks have been adapted.

8.1 The Meta-Management Task

Meta-management is the organizational unit (see also section 3.3.1) responsible for developing the Mega-System. MegSDF defines the activities of this unit in a separate task called the meta-management task. This section describes the role of the meta-management task in MegSDF and specifies the meta-management task as a process.

8.1.1 The Role of the Meta-Management Task

The meta-management task is an extension of traditional software management [DEMA 82], [PAGE 85], [GILB 88]. It addresses the difficulties in software development described in chapter 1, and particularly problems caused by the neglect of general, long

term objectives; problems of coordination and communication; and multiple customers needs and objectives. These difficulties are summarized below as an inverted sub-table of the problem list (table 1.1).

Table 8.1 Difficulties and Problems Addressed by the Meta-Management Task

Difficulties	Caused By	Aspect	Problems
General objectives are neglected	More than one system	Management	There is no clear distinction between general, long-term objectives and local, short-term objectives
Coordination and communication problems on a larger scale	More than one developer		
Different aims and needs	More than one customer		
No standardization of tools	Heterogeneous environment		
Long term objectives are neglected	Long life cycle		

A meta-management with a clear definition of its tasks and role is the only way to ensure a distinction between general and long term versus local and temporary issues. It must coordinate the developer groups, and balance the diverse objectives and needs of the customers. To achieve these goals meta-management must conduct two types of activities:

• Plan development

• Control and coordinate the various tasks

As the responsible agent for general, long term objectives, meta-management determines the direction and trends that the product and development process will take. Meta-Management is responsible for definition of strategies, e.g., testing strategies [HETZ

88], general procedures for quality assurance [DUNN 90], and configuration management [BABI 86]. Its plan should specify the schedules and estimate the resources required for the development process.

Planning should consider both customer requirements and feedback from the tasks of the process. Meta-management communicates with customers to ensure their satisfaction and to understand market needs. It balances diverse aims and needs of the customers. It also defines global priorities and an optimized schedule that includes all the tasks of MegSDF's development process. The meta-management is responsible for activating, suspending, or deactivating systems and synthesis tasks and for specifying milestones for the Mega-System tasks according to actual needs. For example, meta-management might suspend an active system task developing a system for a single customer and use its resources to activate a system task developing a system that can be used for several customers.

Some MegSDF tasks have lower level management. We must differentiate between the responsibilities of lower level and meta-management. The coordination units for meta-management are tasks (projects) as a whole. Meta-Management specifies a global schedule. Lower-level management activities are those of traditional management and lie within the scope of a single MegSDF's task. Lower level management must be coordinated with meta-management. For example, lower level management is responsible for the local schedule of a system task (project), but this schedule must be coordinated with the global schedule which includes other System, Mega-System synthesis, and Mega-System tasks.

The planning activities are similar to software development planning in traditional software development approaches [PRES 92]. However, these plans have a larger scope and must also consider the special characteristics of the Mega-System. For example, the new tasks suggested by MegSDF must be included in the plans.

We propose that risk analysis be used as an essential tool for decision making at the meta-management level. Risk analysis can identify potential problem areas, quantify associated risks, and generate alternatives that reduce risk [CHAR 89], resulting in a more effective "risk-driven" schedule. Thus, risk analysis enables meta-management to activate/deactivate System/Synthesis tasks, allocate adequate resources for critical problems, and solve them expeditiously. Periodic risk analysis and corresponding schedule revision can help assure the schedule fits the real needs.

Controlling the process means allocating resources, activating tasks, and monitoring their operations. Meta-management should monitor the global schedule and resource use, and evaluates both the product as well as the process itself. Like traditional software management, meta-management has to assure conformance of tasks to the global standards and policies, e.g., software quality and configuration management standards. Meta management must also assure compliance of the tasks with the domain model, the Mega-System architecture, and the chosen infrastructure. Meta-management must coordinate the various tasks and resolve communication problems between them.

8.1.2 The Process of the Meta-Management Task

8.1.2.1 Purpose

The purpose of the meta-management task is to plan and control the development of the Mega-System as a whole.

8.1.2.2 Interfaces

Inputs

- Customers requirements - Requirements of the customers/users of the systems.
- Feedback - Feedback from the various tasks to the meta-management task.

Outputs

- Management Control - The schedule assigned to the tasks by the Meta-Management.

8.1.2.3 Processing

The meta-management is responsible for the development of the Mega-System. To ensure an effective development, the meta-management plans the development based on global, long term objectives and controls the development process. Meta-management controls the various tasks of the process on the basis of these plans. Figure 8.1 illustrates the Meta-Management task.

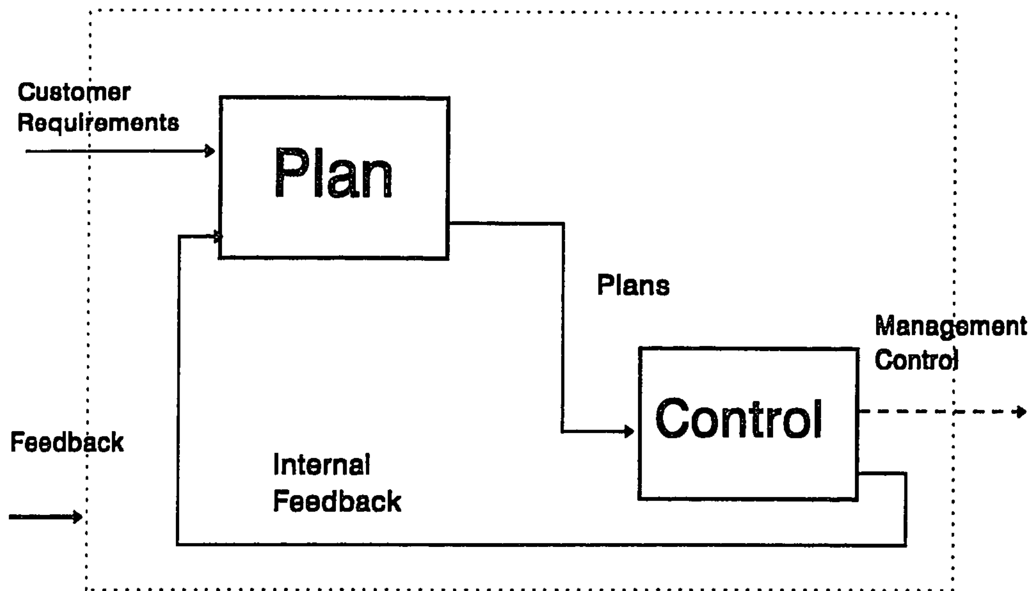

Figure 8.1 Meta-Management Process Diagram

8.1.2.4 Timing

The Meta-management task is a continuous process. It should be active as long as systems are developed and maintained in the domain. Since changes in the domain induce changes in requirements, the meta-management and its activities must continuously adapt themselves to these ever changing needs.

8.2 The System Tasks

This section describes the System tasks in MegSDF. Section 8.2.1 describes the role of the system tasks and their relationship to the other MegSDF tasks. A system development process is defined in section 8.2.2.

8.2.1 The Role of the System Tasks in MegSDF

The MegSDF process model divides the development of a Mega-System into several projects called system tasks. System tasks develop the constituent systems of the Mega-System. A system task is responsible for development of a new system or maintenance of an existing system. In the first case a system is developed from scratch. In the second case, an existing system is repaired, improved, or expanded. Several system tasks may be active concurrently.

Following [MITT 91], we propose that the approach used to develop each system be determined by the special characteristics of the system. One can use the waterfall [BOEH 76], rapid prototyping [GOMA 90], the spiral model [BOEH 88], etc., but MegSDF requires that each system task use the engineering coordination tools of the framework, viz., the domain model, the Mega-System architecture, and the infrastructure.

Systems developed according to MegSDF are "pre-planned". The domain model serves as a basis for further refinements or specializations during the requirement specification phase of each system task (project). The Mega-System architecture provides concepts to be used in the design phase of these system tasks, as well as definitions of the

boundary of the system and its interfaces. An infrastructure that integrates the enabling technologies and allows for efficient incorporation of new technologies is used in the implementation phases. Feedback to the Mega-System task from the system tasks is used to improve and correct the domain model, the Mega-System architecture, and the Infrastructure. The relationship of the system tasks to the other elements of MegSDF is illustrated in Figure 8.2.

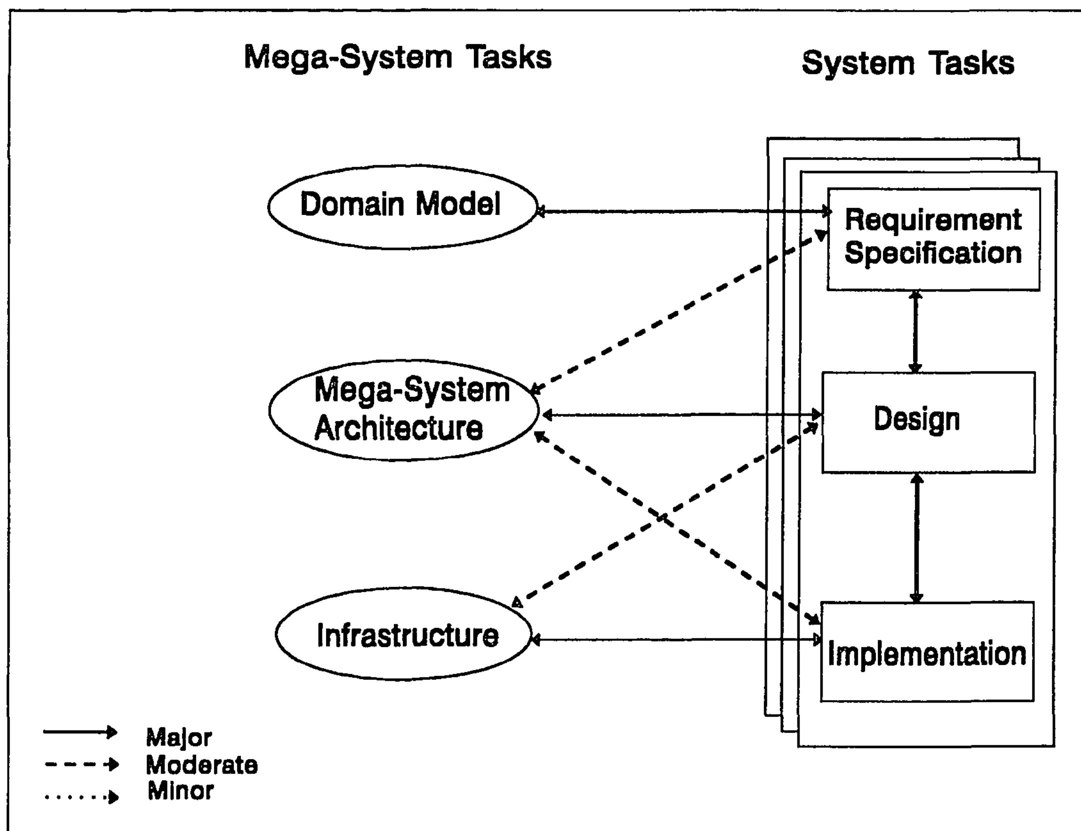

Figure 8.2 Relationship of System Tasks to other Elements of MegSDF

Traditional approaches are typically intended for developing isolated systems that are not part of a system of systems or family of systems. Therefore, they do not include means to ensure coordination and consistency of the developed system with other systems. They do not use a domain model, a Mega-System architecture, or an infrastructure as essential tools. Unlike traditional system development, system tasks in MegSDF are optimized to be part of the entire effort.

8.2.2. The System Development Process

The process of system development is defined using the format described in chapter 4.

8.2.2.1 Purpose

The purpose of the system task is to develop a constituent system.

8.2.2.2 Interfaces

Inputs

• Domain Model - A model of the domain ,defined in section 5.2.

• Application Architecture - Part of the Mega-System Architecture, defined in section 6.2.

• Customers requirements - Requirements of the customers/users of the systems.

• Existing Development Approaches - These approaches will be evaluated in order to define the appropriate development approach.

• Feedback - feedback from the Mega-System synthesis task for modification.

Control

• Management Control - The assigned schedule to the task by the meta-management task.

Circumstance Inputs

• Conceptual Architecture - Part of the Mega-System architecture that includes concepts and a design guideline, defined in section 6.2.

Mechanism

• Infrastructure - The chosen infrastructure of the domain as specified in section 7.2.

Outputs

• System - One of the constituent systems of the Mega-System.

• Feedback - Feedback from the system task to other tasks of the process and to the meta-management.

8.2.2.3 Processing

It is possible to identify three major activities in existing approaches for development of software systems: analysis, design, and implementation. The relationships between these activities, and their detailed content vary with the approach. These activities generalize the prototyping approach [GOMA 90], the spiral model [BOHE 88], and the waterfall model [BOHE 76]. Figure 8.3 uses these activities to illustrate how these approaches will be used in MegSDF's system task (see also section 8.2.5).

Verification, validation, and quality assurance activities are assumed to be part of every task and sub-task to ensure the system provides the required functionality. MegSDF

does not restrict the system tasks to specific techniques, e.g., structured analysis or object oriented analysis. System tasks can use techniques appropriate to the given situation.

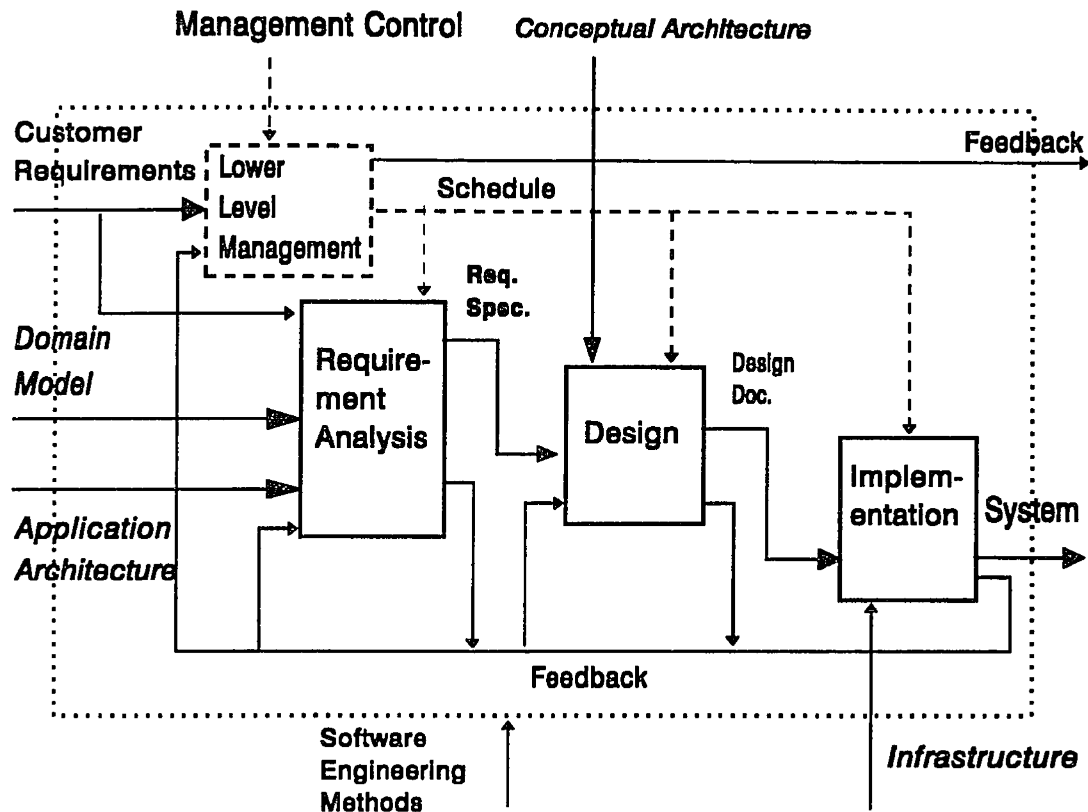

Figure 8.3 System Process Diagram

8.2.2.4 Timing

A system task has a schedule restricted according to the global schedule. Several system tasks might be active concurrently, each developing a constituent system. We propose the same steps for maintenance, i.e., analysis, design, and implementation, as for basic development.

8.2.2.5 Sub-tasks

Requirement Analysis

This task specifies the requirements for the system. The analysis is based on the domain model and the customers' requirements. The application architecture is used for the specification of the system boundary and its interfaces. This task can use structured analysis [YOUR 89], [WARD 86], object oriented analysis techniques[COAD 91a], [RUMB 91], etc.

Design

The constituent system is designed in this task. The process is guided by the concepts defined in the conceptual architecture. This task can use any design technique, e.g., structured design [PAGE 80], [ROSS 77], or object-oriented design [COAD 91b].

Implementation

The system is implemented in this task using the infrastructure. The programs are either directly developed or generated by code generators (when available), debugged, and integrated. The system as a whole is verified and validated.

Lower-Level Management

This task controls the process of constituent system development. A development approach is selected, e.g., the waterfall approach [BOHE 76] or its variations, the spiral model [BOHE 88], or rapid prototyping [GOMA 90]. Lower level management is also responsible for coordinating the techniques used in each sub-task. It is also responsible for planning the development process, scheduling the sub-tasks, and internal quality assurance

and configuration management. The management of the system task is responsible for communication and coordination with the meta-management and the other system tasks.

8.3 Mega-System Synthesis in MegSDF

This section defines the Mega-System Synthesis task. Section 8.3.1 describes the role of Mega-System synthesis in MegSDF. Section 8.3.2 defines a Mega-System synthesis process.

8.3.1 The Role of Mega-System Synthesis

This task is responsible for providing an effective and efficient Mega-System to the customers. It integrates the constituent systems into a coherent Mega-System. As a framework for the development and integration of Mega-Systems, one of MegSDF's goals is to simplify the activity of systems integration by developing constituent systems that are "pre-planned".

The Mega-System synthesis task is guided by the application architecture which specifies the systems and the building blocks of the Mega-System according to the domain model. After evaluation of the capacity needs for the system, e.g., number of users, frequency of transactions, number of data records, etc., a Mega-System configuration is defined. This configuration specifies the hardware configuration and the allocation of building blocks to appropriate hardware components. Scheduleability analysis [HALA 91] can be used as a tool for optimization and verification. The infrastructure enables the

linkage of these systems into a coherent Mega-System. Feedback from the Mega-System synthesis task is used to improve the Mega-System architecture and the constituent systems.

The Mega-System must operate efficiently, with reasonable response time. However, the system must also to be cost-effective. Accordingly, Mega-System synthesis strives for an optimized configuration which enables effective use of the system at a reasonable cost.

Extendibility is an essential requirement for a Mega-System. The long life cycle of a Mega-System, with attendant changes in requirements, means the need for additional resources is to be expected.

The Mega-System Synthesis task is not responsible for developing and implementing constituent systems. However, feedback from this task may improve the constituent systems. This task evaluates the Mega-System architecture and verifies the operability of the infrastructure.

The Mega-System synthesis task may be considered as a generalization of the integration phase of traditional systems development. However, it deals with interoperating systems and not with system components. It also includes specification of a Mega-System configuration that addresses both software and hardware configurations. Moreover, it uses the Mega-System architecture and the infrastructure as essential tools.

Mega-System synthesis is essentially a customization of the system according to the customer needs. This process specifies the parts of the system that will be used in the actual Mega-System. It may identify parts that were not yet developed or which need

modification. Since the Mega-System architecture identifies elements that need customization, this task includes specification of the actual parameters for these elements.

8.3.2 The Process of Mega-System Synthesis

8.3.2.1 Purpose

The purpose of the Mega-System synthesis task is to provide a coherent Mega-System to a customer.

8.3.2.2 Interfaces

Inputs

- Application Architecture - The meta-design of the Mega-System, defined in section 6.2.
- Customers requirements - Requirements of the actual customers/user of the specific Mega-System.
- Systems - The constituent systems that were developed by the system tasks (projects).

Control

- Management Control - The schedule assigned to the task by the meta-management task.

Mechanism

- Infrastructure - The chosen infrastructure of the domain, specified in section 7.2.

Outputs

- Mega-System - A Mega-System that fits the requirements of the customers.

• Feedback - Feedback from the task to the Mega-System architecture design, infrastructure acquisition, system and meta-management tasks.

8.3.2.3 Processing

This task determines the software configuration on the basis of the application architecture, actual user needs, and available systems. It then defines a suitable Mega-System configuration, including hardware configuration and allocation of software systems and building blocks to hardware components. The various components are customized and linked together. Finally, the usability of the Mega-System as a whole is verified. Local management controls the process and communicates with the customers, the system tasks, and the meta-management. Figure 8.4 illustrates the Mega-System Synthesis task.

8.3.2.4 Timing

Several tasks of Mega-System synthesis may be active concurrently, each responsible for providing an appropriate Mega-System to a specific customer. In the case of generic system of systems, each Mega-System synthesis task provides a specific system to a specific customer. For systems of systems, the Mega-System synthesis tasks may provide distinct versions of the Mega-System to a single customer.

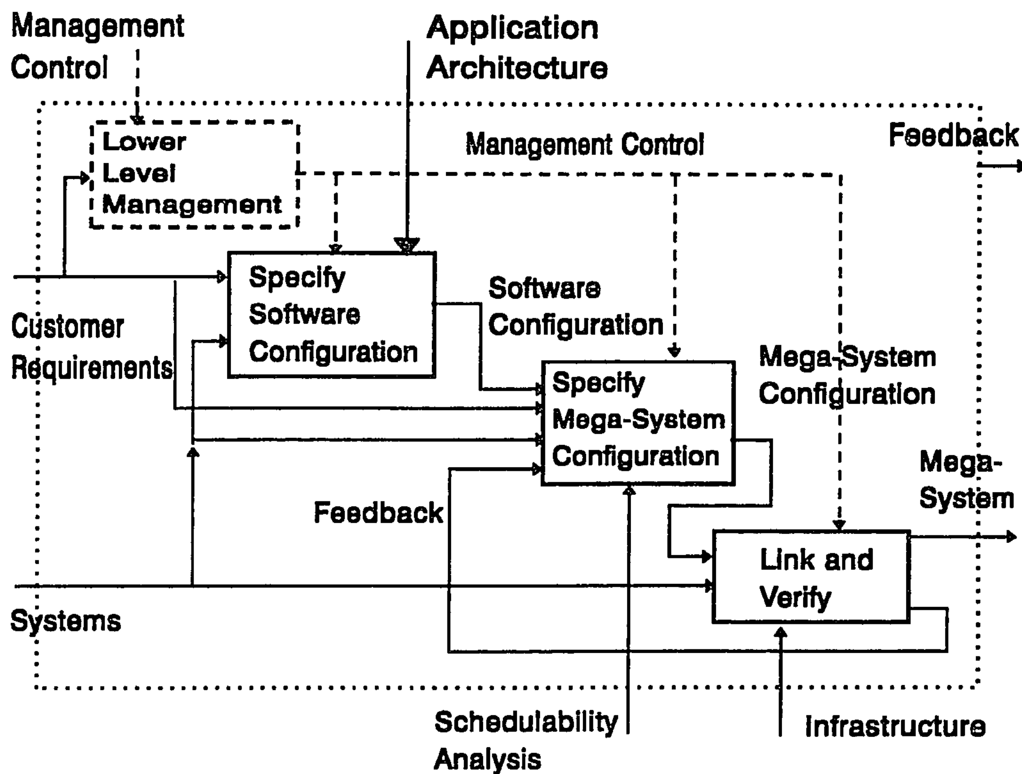

Figure 8.4 Mega-System Synthesis Process Diagram

8.3.2.5 Sub-Tasks

Specify Software Configuration

The software configuration is determined based on the actual customer needs and guided by the application architecture. The software configuration defines the list of building blocks and systems with an actual version and release numbers. This task can be considered as a generalization of software configuration management [BABI 86]. However, the components are systems of larger scope than regular software components. The output of this task includes a definition of the software configuration, with the actual parameters identified for the various systems and building blocks.

This task is not responsible for implementing building blocks. Therefore, feedback from this task might include modification requests, e.g., a requirement to modify an existing building block/system, or to develop a new one. These modifications are done by the system tasks. Thus, feedback from this task improves the application architecture and constituent systems.

Specify Mega-System Configuration

This task specifies the hardware configuration and allocates systems and building blocks to the hardware components.

Link and Verify

This task links together the hardware and software components. The software is installed on the hardware components. No actual linkage may be required; the systems may "start" by beginning to communicate using the infrastructure. In other cases, installation may be more complicated, involving specification of parameters and other administrative activities. The Mega-System as a whole is tested to ensure its usability. The objective of MegSDF is that this task be as simple as possible and that the effort required for linkage and verification be minimal. This task is an extension of the integration phase of traditional software development approaches.

Local Management

This task is responsible for planning and controlling the Mega-System synthesis process. It specifies the schedule for the sub-tasks and local policies and procedures. This task is also responsible for communicating with the meta-management and systems tasks.

CHAPTER 9

A SCENARIO

This Chapter describes how MegSDF can be used for the development of Mega-Systems. Section 9.1 describes the current status of a hypothetical software house that develops systems in the insurance domain using traditional software engineering methods. Section 9.2 describes how MegSDF can be used. Section 9.3 discusses the advantages of applying MegSDF.

The scenario is based on personal experience and discussions with software developers. It does not describe a specific software house, and there may of course be many cases where only some of the problems exist. The scenario does not include development of the elements of the framework. It is only used to illustrate the use of the framework and the process model.

9.1 Current Status

A software house develops and maintains a number of large, complex systems in the insurance domain. The systems consist of hundreds of programs. The total amount of code is greater than 1M lines of code. The algorithms implemented within the systems are

complex and include actuarial and escalation formulas. We describe the current state of affairs from the viewpoint of the customers, the developers, and the systems.

The Customers

The software house has a large number of customers, including both insurance companies and large insurance agencies. The customers are located in different states, each state with its own laws and insurance regulations. The insurance companies and agencies sell different kinds of insurance, e.g., life, property, liability, etc. Most sell policies for all types of insurance, but several agencies specialize in a specific kind of insurance, e.g., life insurance. The users must operate different systems to accomplish their jobs.

The Systems

The systems have undergone many generations of modification. The developers of these systems are not available and documentation is poor and out-of-date. They are hard to maintain, update, or integrate.

The systems operate in environments of different vendors, varying from mainframes with hundreds of terminals, e.g., IBM 3090s or CDCs, to smaller systems with several terminals, e.g., VAXs and personal computers (IBM compatible and Macintoshs[6]). The computers use different operating systems.

Most of the systems were developed using COBOL. Some old systems include assembly programs that no one dares to change since the algorithms used in these systems are not documented. Some new systems were developed using application generators. The status of data handling and communication within the systems is similar. Some old

---

[6] Macintosh is a trademark of Apple Computers, Inc.

systems use files and indexed files to store information. Other systems use different database management systems. The mainframes use different and incompatible tools and communication networks, e.g., SNA for IBM products, Dec Net, etc.

The interaction between the systems is often done either manually or by batch processing. In the worst case, an operator must type information given as paper reports of one system to another system. In other cases, special batch programs must be run in order to download information from one system onto disk files, which other programs then upload to other systems. Files are transferred from one system to another by Remote File Transfer and similar mechanisms. The connection between the systems is, generally, not transparent, and requires the intervention of human operators, a type of interaction which is unreliable and can cause inconsistency and affect the integrity of the data stored in the systems.

Different systems with the same functionalities have been developed to operate in different environments. Over time, their functional commonality disappeared in response to the requirements of different users with different needs and objectives. Moreover, there are redundant functionalities, e.g., every system handles the insurance clients separately; every system handles policies. Thus, the software house deals with an enormous number of modules.

There is redundant and often inconsistent data in the different systems. A change of insured address or phone number, for example, requires updates to the life insurance, property, agent, and vehicle systems. Every system stores different information for the same entities. There are differences in attribute names, types, even semantics. A

modification first requires identifying all affected systems. A specialized solution, fitting the system data and functionalities, must then be implemented for each system.

There is no standard user interface; every system uses its own layout of screens and approach to user interaction. Training new agency employee requires several weeks.

The Developers

The developers are located in several sites. Furthermore, the organization of the software house separates the system analysts from the designers and programmers. Even groups working on the same project are located in different sites.

Each group develops its system using local procedures and standards. Coordination between the groups is mainly administrative and is inefficient. Technical solutions are shared on a voluntary basis. There is no real coordination between the developer groups. Similar functionalities are developed by different groups because of a lack of knowledge, or because of a lack of authority that might enable imposition of some simple restrictions that would compel the various parties to use an existing solution, or decide to develop a general solution for all groups, with attendant saving in both development and maintenance effort.

Developers are usually assigned to specific systems since training new software engineers is time consuming. The average turnaround time for a software engineer in the software house is about two years. It is generally impossible to move a programmer from one system to another one, since mastering application complexity and system-specific development procedures and standards requires extensive training.

CASE tools are used primarily for reverse engineering and documentation. These tools yield a large amount of information but it is hard to read, understand, or upgrade.

The Problems

The previous discussion may be summarized by the following problem list:

• The software house has a list of hundreds of "urgent" requests for modifications of existing systems. These requests/demands are often ambiguous or contradictory. Any change request requires tremendous effort. The response time to customer needs is unacceptable.

• The temporary, local problems of the customers govern the system.

• Customers insist on working with one system that includes all functionalities, rather than having to use a number of "independent" systems.

• There is a shortage of professional programmers and software engineers who are also experts in the insurance domain.

• Maintenance is the essential part of the work and new systems are almost not developed, while competition with other companies is intense. New software houses offer new systems at cheaper prices. These systems are based on new technologies and offer new functionalities that do not exist in the systems of the software house.

9.2 A Solution Based on MegSDF

In view of current problems, we recommend the company adopt MegSDF's concepts and process model.

Meta-Management

The company must establish a meta-management team. This team will include all managers of the various software developer groups and will be responsible for enforcing standards and policies for all groups. Meta-management will communicate with customers and balance their requests. Meta-management will schedule all activities and handle budgeting. Meta-management will also determine trends and strategy for the entire system.

Domain Analysis

A special group, consisting of both computer and insurance experts, will develop a domain model. This group will be independent of a specific development project. The group will provide a general, comprehensive model of the insurance domain. The group will be responsible for updating the model as a result of domain dynamics and feedback from development teams.

Mega-System Architecture Design

A separate group will study existing architectures for software systems. The group will define a Mega-System architecture which will be used as a reference model and a guideline for developing systems in the domain. The design and implementation concepts

will be used by all development groups. The application architecture will determine the overall structure, the components, and their interfaces.

Infrastructure Acquisition

Another group will be established to examine existing infrastructures. This group will choose, on the basis of the conceptual architecture, an appropriate infrastructure that will be used as the basis for implementing systems in the domain. The responsibility of the group includes verification and validation that the infrastructure supports the concepts specified in the conceptual architecture. The group must also support the operation of the infrastructure as an active part of the Mega-System, e.g., registration of services, installation of building blocks, allocation of resources, and measurement of resource utilization. This group is responsible for evaluating new technologies and incorporating them as required into new versions of the infrastructure.

System Tasks

Using the domain model and the Mega-System architecture, existing systems will be evaluated for compatibility. As a result, a number of applications will be modified and other systems will be developed from scratch. Meta-management will decide the order of development. The systems will be developed according to the Mega-System architecture based on the selected infrastructure. All systems will be developed as independent projects.

Mega-System Synthesis

A Mega-System synthesizing group will be assigned to every customer. The synthesizers will integrate a Mega-System in view of customer requirements. The synthesizers will

choose the components, customize the systems, and request modification of existing systems or development of new systems as required.

9.3 Advantages of Using MegSDF

Under MegSDF, the software house develops a Mega-System of the generic system of systems type as a domain-wide solution. We clarify the benefits of using MegSDF from the viewpoints of the customers, developers and systems.

The Customers

The Mega-System consists of systems for insurance companies, integrated with systems for agents. These systems handle policies, maintain insured information, and support computation of insurance rates, claims adjustment, accounting, and other insurance functionalities. The systems assume there exists a large number of customers with diverse needs and objectives. The Mega-System is scalable and can be configured for insurance companies, general agencies that offer all types of insurance, and specialized agencies, e.g., life insurance agencies.

The Mega-System is developed with emphasis on user transparency. Users operate a unified, coherent, large system, with a common user interface, that offers assorted services, rather than operating multiple systems. All screens of the Mega-System will have the same structure and interaction with the system will be based on the same interaction

types. The functional keys have the same role in all systems. Thus, training of a new insurance company or agency employee will require less time.

The connection between the systems is automatic, requiring minimal human operator involvement. For example, an agent fills out, using the agent system, the insured application and issues a temporary policy. The application is automatically transferred to the insurance company system. In this system, a policy is underwritten based on the application information, approved, and issued to the customer.

Meta-management balances the multiple, even contradictory requirements of customers. It resolves contradiction in requirements, imposes unified, generalized solutions, and specifies a global schedule, optimizing the demands of the customers. For example, all system will use the same attributes for insured identification, policy identification, etc. If other representations for insured identification are required, interfaces will be supplied to translate the exceptional attributes to the common attributes.

The Systems

The systems of the software house are now developed as a generic system of systems (a coordinated set of federated systems). The common set of functionalities is specified based on the common domain model. The constituent systems are developed according to the conceptual architecture, utilizing the infrastructure.

The systems are developed as general solutions and therefore fit many customers with different needs and objectives, not for a specific insurance company in a specific state. The systems will be developed with parameters to enable their efficient

customization. For example, it will be possible to adjust the installment plan, the profit percentage, etc., according to the insurance company policy.

The domain model facilitates specification of the relationship of a system with its environment and enables earlier identification of integration requirements. For example, since all information on the different types of policies of clients is accessible, the marketing department could analyze consumer behavior to identify appropriate new offerings. Similarly, using the claims system information, the actuary will be able to identify high-risk geographical areas and determine appropriate insurance rates.

The domain model enables the identification of similar functionalities. Data redundancy is reduced; if redundancy is required to improve availability or efficiency, it is controlled. This reduces problems with inconsistent data. For example, insured information is handled by a single system which is open to retrieval request from other system, so address correction will be done only in a single system.

Developing systems according to common design and implementation concepts and using a common infrastructure eases development and maintenance of systems and simplifies systems integration. Using these concepts, it is possible to maintain the consistency of the systems and their structure even after the designer of the system leaves the company.

The common infrastructure and architectural concepts promotes the portability of the systems. Rather than developing different versions of systems for different environments, e.g., claims systems for DOS, IBM mainframe, and CDC, drivers for these environments are developed. For example, drivers to different types of databases e.g.,

IMS, IDMS, or indexed files, which provide the same data handling functionalities, will be developed. Thus, it becomes possible to use the same software in different environments with different operating systems, (IBM 3083, personal computers, VAXs, etc.) with minimal effort. Porting a system to a new environment primarily requires mainly developing drivers for the environment.

These improvements reduce the number of different systems and so also reduce opportunities for decentralized modification over time. These characteristics simultaneously improve the processes of modifications and upgrading, since fewer programs must be considered and changed per modification. For example, a change of attribute for a policy information will be local to the policies system.

The Developers

MegSDF promotes global coordination between the different groups developing constituent systems and provides a means for improving communication between them. The domain model provides a basis for common understanding among the different developer groups. It uniformizes the terms of the insurance domain for all developers. In the event of conflicts between developer groups, the domain model will be used as a reference.

The Mega-System architecture imposes common design and implementation concepts, e.g., common user interface and design constructs, to be used in the systems of the Mega-System. It requires compatibility of the constituent systems with these elements, thus uniformizing the implementation of the entire system.

The infrastructure uniformizes the handling of technologies for all systems and is not merely an option. It also enables developers to restrict their attention to domain problems. Based on existing enabling technologies, it provides standardized tools for data handling or user-interfacing. Thus, the infrastructure precludes the need to develop database management systems or user-interfacing tools within projects and restricts the number of different technologies used in the domain. The infrastructure provides a means for interaction and facilitates integrating the constituent systems.

Meta-management is responsible for coordinating developer groups. The domain model and application architecture are used to reduce the risk that groups replicate functionalities because of lack of coordination.

The mobility of developers among groups is also improved. The knowledge of a specialized group is generally specified in the domain model and so is easier to learn. Since the development of all constituent systems is based on shared concepts a "new" developer can easily adjust.

Looking back to the problem list of section 9.2.1, it is possible to summarize the new status:

• The list of hundreds of "urgent" requests may be shorter and different in nature. Since the systems are built as general solutions, fewer adaptions are required.The meta-management balances the demands of the customers and imposes general, common solutions.

• Change requests require fewer efforts. There are fewer systems to examine or modify for each change request. The openness of the systems improves their integratability.

• Meta-management specifies a global schedule and determines directions in consideration of customer needs.

• The customers use a unified system (with assorted functionalities) since the systems are developed according to the common user interface concepts of the conceptual architecture, utilizing an infrastructure that facilitates interaction of systems.

• Mastering the application domain is easier. The domain model provides an effective basis for understanding the domain. The mobility of developers is improved since procedures and implementation concepts are explicitly specified and common to all systems.

• Developing systems using common design concepts and utilizing a common infrastructure enables developers to focus on domain related problems. The infrastructure reduces the effort required to incorporate new technologies. Thus, the developers have more time to develop new systems and applications.

CHAPTER 10

CONCLUSIONS AND SUMMARY

This chapter includes conclusions and summarizes the thesis. Section 10.1 evaluates MegSDF according to the characteristics indicated in chapter 3 and identifies the contribution of each task to the quality of the Mega-System. Section 10.2 discusses prerequisites for success in implementing MegSDF. Section 10.3 summarizes the thesis.

10.1 Requirements Verification

10.1.1 Realization of Framework Requirements

Chapter 3 established the following requirements for a framework for developing Mega-Systems:

- General,
- Comprehensive,
- Operative, and
- Open

MegSDF is ***general***. It is domain independent. It is appropriate for various application domains, e.g., data-processing and real-time systems, and applicable to different types of Mega-Systems.

MegSDF is ***comprehensive***. It incorporates engineering, managerial, and technological aspects.

MegSDF is **operative**. It defines an engineering process that specifies the tasks required to develop Mega-Systems, their deliverables and interconnections. The framework is coherent: all its parts are interrelated, the results of tasks are used as inputs to other tasks, and all the activities are integrated into an engineering process.

MegSDF is ***open*** and flexible. Developers can select an appropriate technique to implement their tasks. Domain analysis, for example, allows various suitable modeling approaches. System tasks can be implemented by any traditional system development approach, provided they use the requisite framework elements: the domain model, the Mega-System architecture, and the infrastructure. The process is adjustable, allowing activation and deactivation of systems and synthesis tasks according to actual necessities. The Mega-System and Meta-management tasks continuously evaluate changes in the domain, customers requirements, and technologies, as well as feedback from developers, incorporating them in the Mega-System as required.

10.1.2 Quality Attribute Map

A Mega-System must not only meet the requirements of the customer but also be:

- Effective,
- Open,
- Efficient,
- User-friendly,

• Reliable, and

• Maintainable

Effective

The system should meet the requirements of the customers. In the case of Mega-Systems, requirements are not well defined; they might be ambiguous, and/or contradictory. The process model addresses this in two ways. First, a domain model provides a comprehensive, general domain representation, but not one specific to an individual customer. The Mega-System synthesis tasks tailor the Mega-System to the special needs of the customers.

Open

The system should be open, that is integratable, scalable, extendible, and upgradable. It should be possible to integrate the system with other systems, to define different configurations for different customers with different set of functionalities, to extend the system with new functionalities, and upgrade the system with new technologies.

The Mega-System architecture design and infrastructure acquisition tasks support the openness of the Mega-System. The conceptual architecture specifies concepts which ensure the extendibility, scalability, and integratability of the system. The infrastructure supports these concepts and enables efficiently incorporating new technologies, and upgrading existing technologies.

Efficient

The system should be efficient, optimizing hardware price, performance requirements, development efforts, and quality requirements, with an emphasis on long-term solutions, not local, temporary ones.

The infrastructure and system tasks facilitate the efficient operation of both the constituent systems and the entire Mega-System. The Mega-System synthesis task provides a balanced optimization over the non-functional requirements of the customer, e.g., response time and hardware cost.

Development effort is reduced since all developer groups use the domain model, common design and implementation concepts, and a common infrastructure. The infrastructure also enhances the portability of systems developed using its services.

User-friendly

The system should be user-friendly in the sense of consistency and adjustability. Users should have the feeling of using a single system. They should have a consistent user interface and interaction types. Since a Mega-System may have a heterogeneous user group, it should also allow adjustment and customization of features according to user preference.

The environment view of the conceptual architecture includes concepts for a common user interface. These concepts, supported by the infrastructure, promote uniformity and consistency over the entire Mega-System.

Reliable

The system should be reliable, i.e., highly available, fault-tolerant, and secure. It should ensure consistency of data in the event of failure and protect resources and data from unauthorized use.

The conceptual architecture specifies mechanisms to ensure the reliability of the entire Mega-System and the infrastructure supports these mechanisms. Of course, the developers of the constituent systems should develop reliable systems using the methods of traditional software engineering and the services of the infrastructure.

Maintainable

The system should be maintainable. It must be divided into cohesive, minimally coupled building blocks to ensure its manageability. The constituent systems must be consistent and use common design and implementation concepts.

The application architecture design task specifies building blocks and clusters on the basis of the domain model. The conceptual architecture provides common design and implementation concepts. The system tasks use these concepts, together with traditional software engineering methods, to produce a maintainable Mega-System.

Table 10.1 identifies the quality attributes each task contributes to the Mega-System.

Table 10.1 Impacts of MegSDF's Tasks on Quality

	Domain Analysis	**Archi-tecture Design**	**Infra-structure Acquisition**	**System Tasks**	**Synthesis Tasks**
Effective	+				+
Open	+	+	+		
• Extendable	+	+	+		
• Scalable		+	+		
• Upgradable			+		
• Integratable	+	+	+		
Efficient		+	+	+	
• Performance			+	+	+
• Development efforts	+	+	+		
• Hardware price					+
• Portable			+		
User-friendly		+	+		
• Consistent		+			
• Adjustable		+			
Reliable		+	+	+	
• Available		+	+		
• Fault-tolerant		+	+		
• Secure		+	+		
Maintainable		+		+	
• Modular		+		+	
• Consistent		+		+	

10.2 Prerequisites for Success

The implementation of MegSDF requires an organization with software process maturity [HUMP 88], [SCHL 92] and a comprehension of software engineering methods. It cannot be applied in an organization at the "initial" level, lacking procedures or processes: an organization that uses ad-hoc solutions can neither develop nor use means for engineering coordination, e.g., common design and implementation concepts.

On the other hand, for organizations at the appropriate level, the success of MegSDF depends on:

- The commitment of management,
- The development and maintenance of accurate means for engineering coordination, and
- The adequate use of the engineering coordination concepts and tools by the developers.

Management commitment is required because MegSDF emphasizes long term solutions. Developing such solutions requires an initial investment that is often expensive and time consuming. The benefits of these solutions are not seen immediately, but only in the long run. Without management commitment, and allocation of appropriate resources, the MegSDF development environment will be infeasible.

MegSDF includes development of a domain model, a Mega-System architecture, and acquisition of an infrastructure. These elements are common and general means for engineering coordination that transcend the development of the constituent systems. Since they are an integral part of the development process they must be maintained as long as systems are developed and maintained in the domain. Since domains and technologies

change, it is mandatory to update these means to assure their effectiveness. It is also required to develop them accurately.

In addition to the commitment of the management and the development of appropriate means, the developers themselves must use these means properly as circumstances and guidelines. The domain model, Mega-System architecture, and infrastructure must be used by all developers. They must be part of the engineering culture, thus requiring changes in methods and working style. Local, temporary solutions become counter-practical from this viewpoint. It is necessary to understand these means provide a way of coping with the complexity of Mega-Systems and enable the developers to solve the real problems of the application domain.

10.3 Summary

This thesis specifies a framework for developing large, complex software systems which we call Mega-Systems. MegSDF incorporates the engineering, managerial, and technological aspects and a process model for coordinating these aspects.

MegSDF proposes developing Mega-Systems as domain wide, long-term systems following a pre-planned approach. The Mega-Systems are developed as open distributed systems (federated systems) which share data and functionalities and are planned to be integrated with other systems and to be changed in the future.

MegSDF partitions the development process into multiple coordinated projects, each developing one of the constituents systems. Two levels of managements are proposed in order to enforce distinction between global, and long-term versus local and short-term issues. Meta-Management is responsible for the development of the entire Mega-System, while lower level managements are responsible for development of constituent systems.

The MegSDF engineering process model specifies the activities or tasks required to develop Mega-Systems, including their deliverables and interrelationships. Some of these tasks generalize traditional activities, e.g., system or meta-management tasks, while others substantially extend existing approaches and are specific for MegSDF.

The process model consists of System, Mega-System Synthesis, Mega-System, and Meta-Management tasks. System tasks develop constituent systems. Mega-System Synthesis tasks assemble Mega-Systems from constituent systems according to actual customer needs. The Meta-Management task plans and controls the entire process. Mega-Systems tasks provide a means for engineering coordination and include Domain Analysis, Mega-System Architecture Design, and Infrastructure Acquisition tasks. MegSDF process is active for the duration of software systems in the domain. The meta-management and the Mega-System tasks are continuous. The systems and synthesis tasks are activated according to actual domain needs.

Domain analysis provides a general, comprehensive, non-constructive domain model. The domain model is used by the developers of the Mega-System as a common knowledge base. It is built as an integration of multiple perceptions, each of which represents the domain from a significant viewpoint. A domain modeling schema (with

modeling primitives) is also proposed to facilitate modeling and integrating multiple perceptions.

A Mega-System architecture is proposed as a primary means of engineering coordination, to assure the uniformity and consistency of the entire system. The conceptual architecture defines common design and implementation concepts. The application architecture specifies the overall structure of the Mega-System, its components, and their interfaces. The Mega-System architecture provides an explicit definition of common design and implementation concepts for systems in the domain. A model for a conceptual architecture and an outline for an application architecture are defined in MegSDF.

An infrastructure is proposed as a common service-based platform that integrates all enabling technologies. It supports the architecture on the implementation level. It also promotes portability, simplifies bridging different technologies, and facilitates incorporating emerging technologies in a unified way. MegSDF partitions the infrastructure into service groups based on the conceptual architecture. The applicability of existing infrastructures to MegSDF is evaluated.

APPENDIX A
MegSDF PROCESS DIAGRAMS

A.1 MegSDF First Level

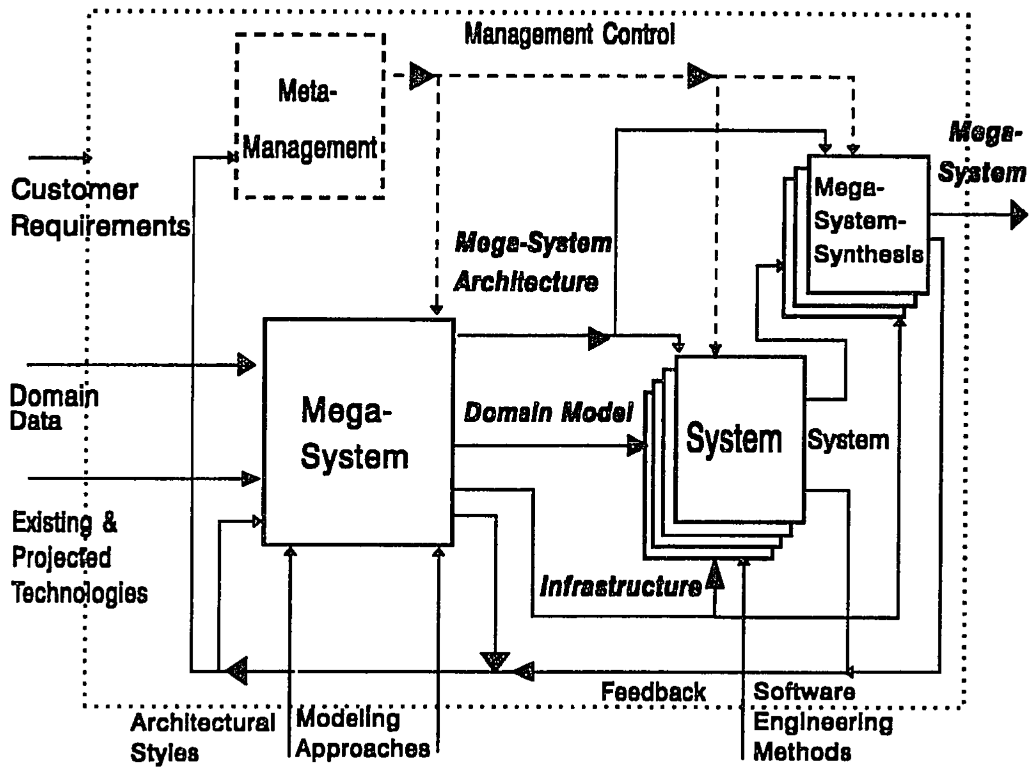

Figure A.1 MegSDF First Level Process Diagram

A.2 Mega-System Tasks

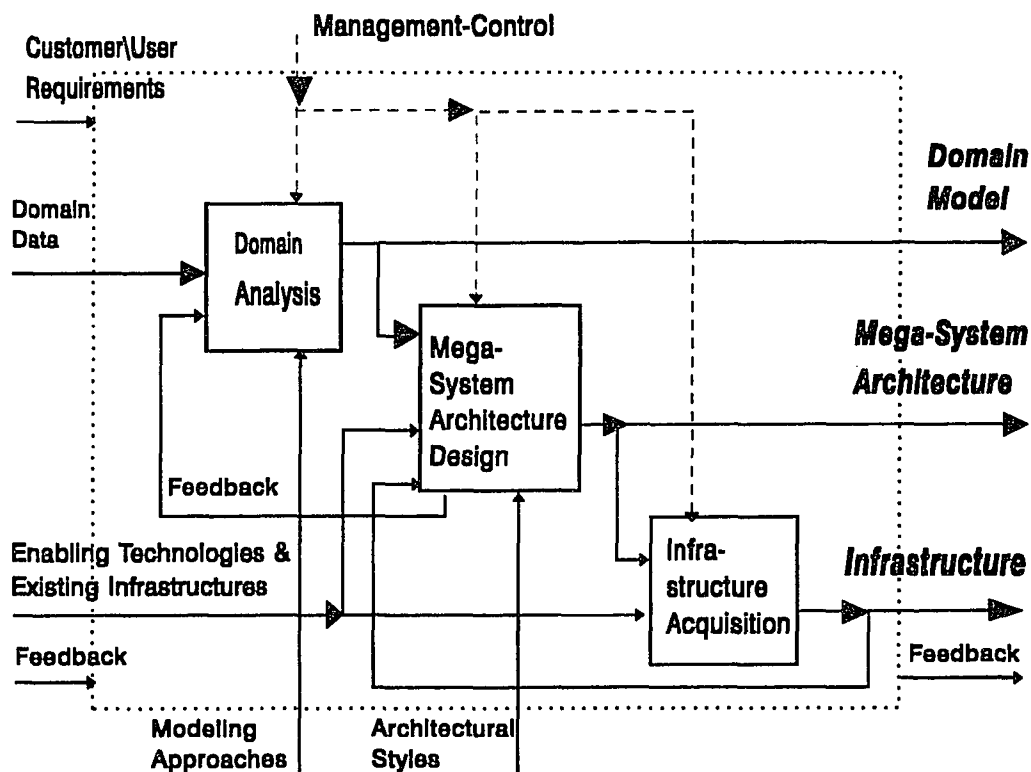

Figure A.2 Mega-System Tasks Process Diagram

A.3 Domain Analysis

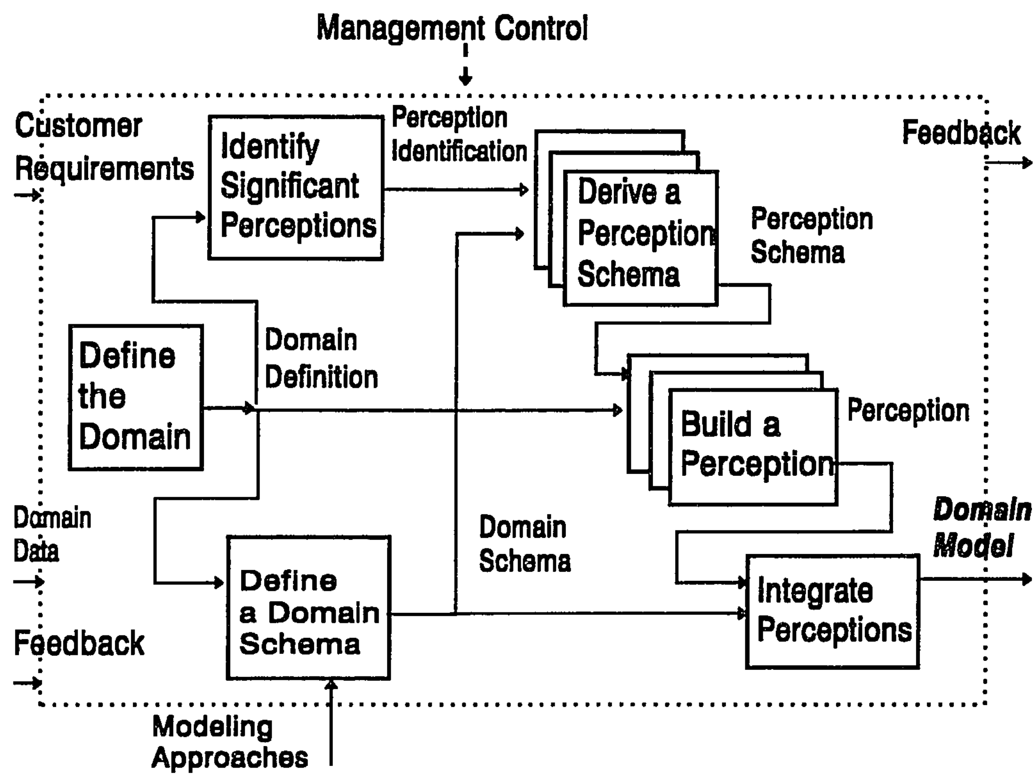

Figure A.3 Domain Analysis Process Diagram

A.4 Mega-System Architecture Design

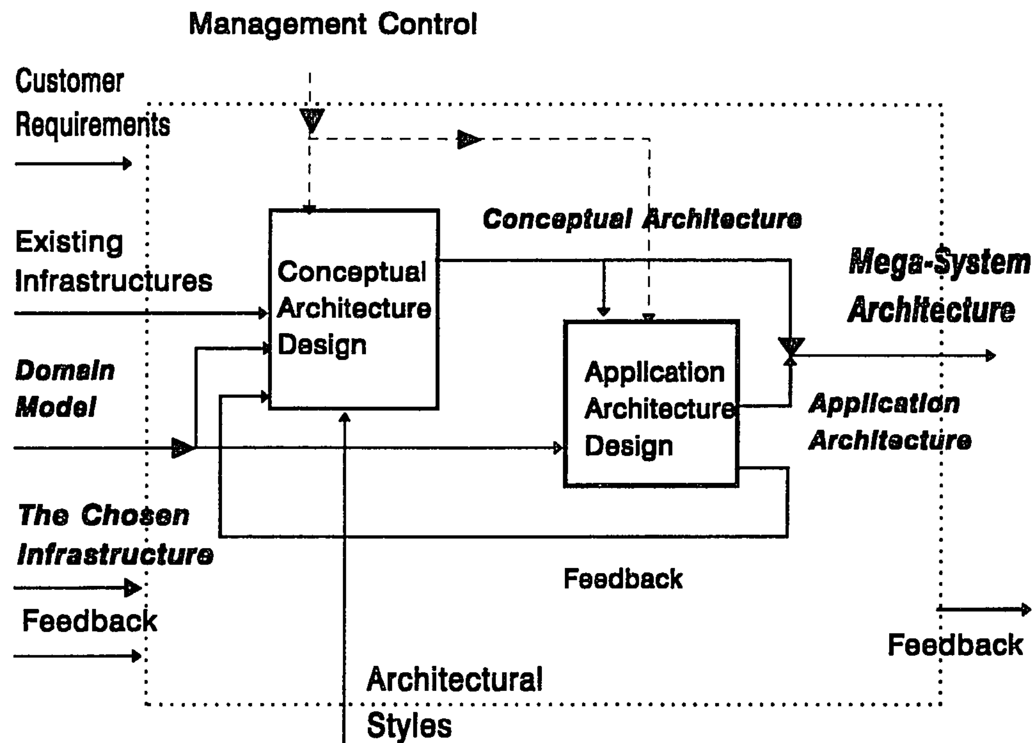

Figure A.4 Mega-System Architecture Process Diagram

A.5 Conceptual Architecture Design

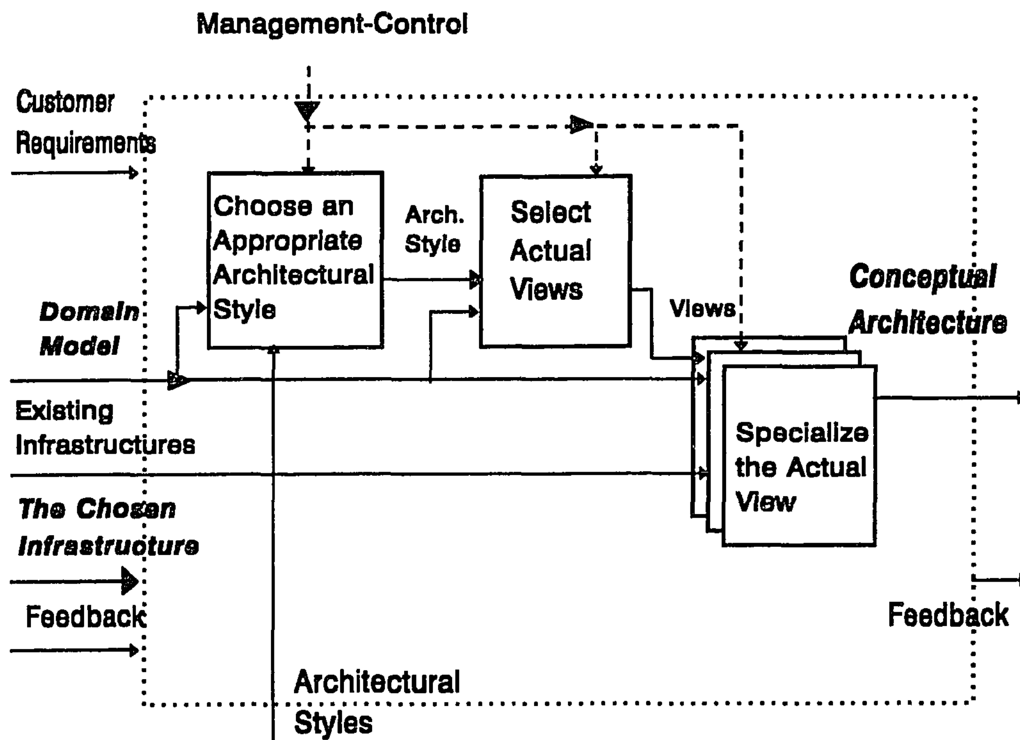

Figure A.5 Conceptual Architecture Design Process Diagram

A.6 Application Architecture Design

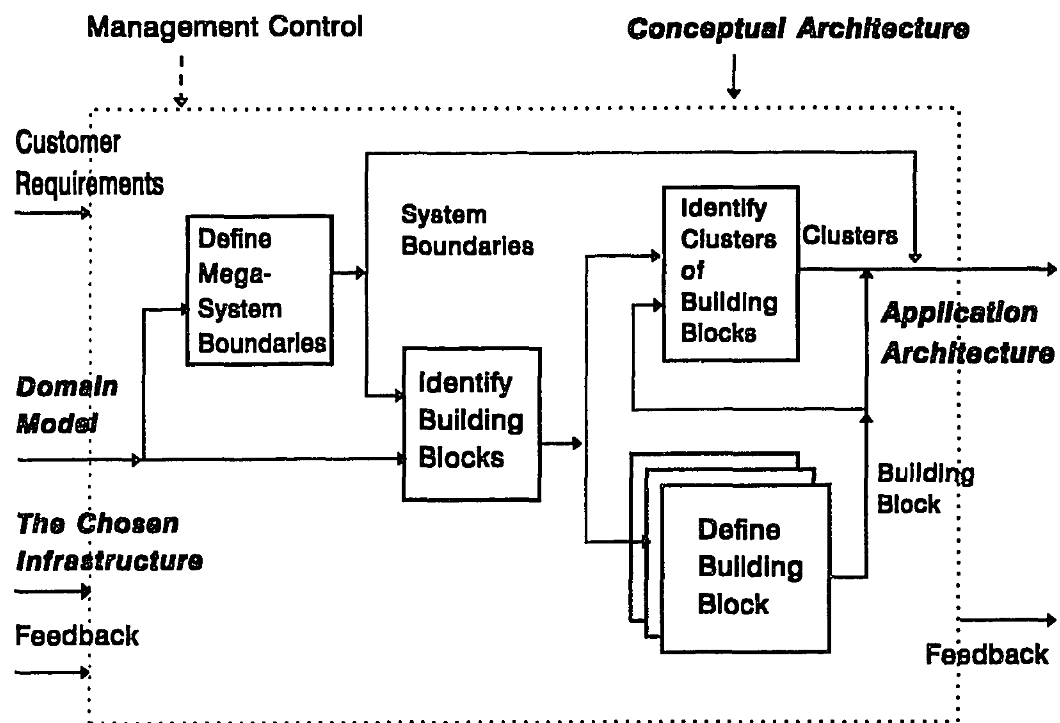

Figure A.6 Application Architecture Design Process Diagram

A.7 Infrastructure Acquisition

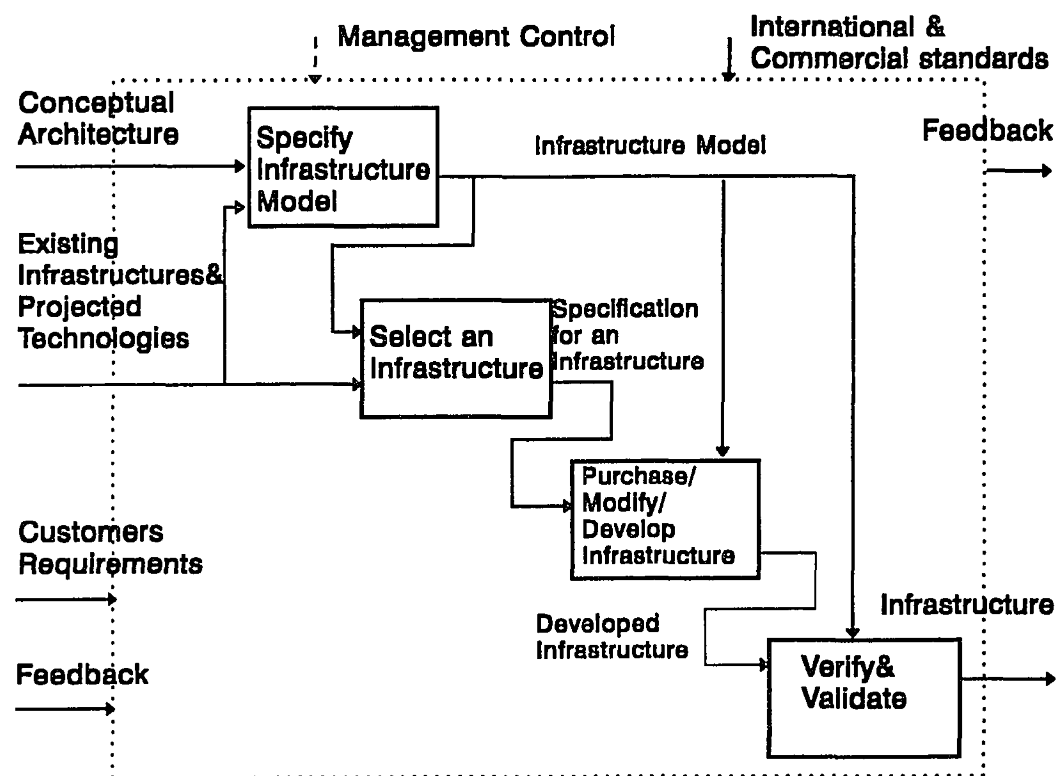

Figure A.7 Infrastructure Acquisition Process Diagram

A.8 Meta-Management

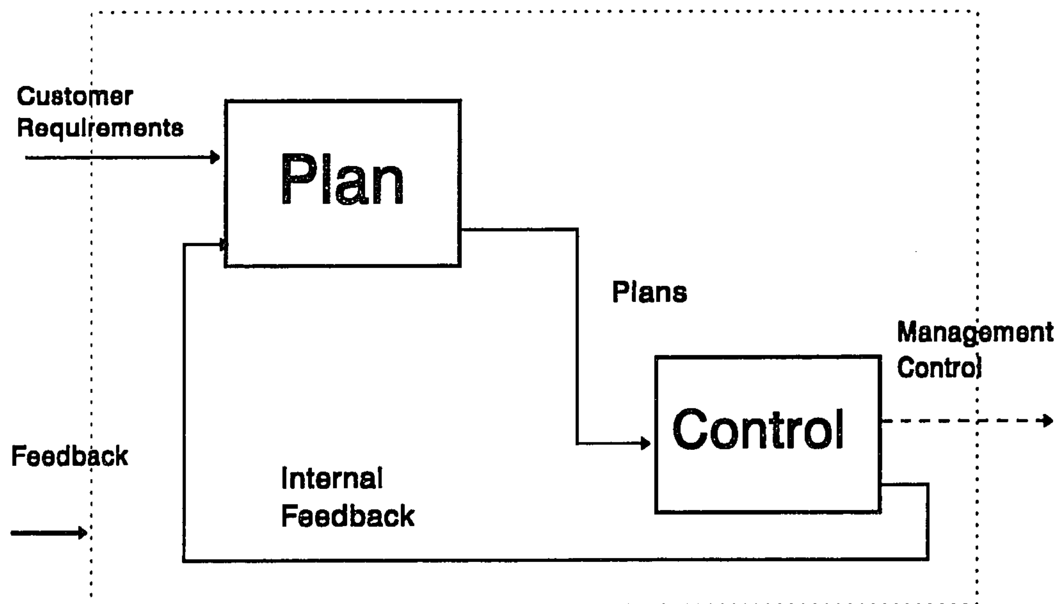

Figure A.8 Meta-Management Task Process Diagram

A.9 System Task

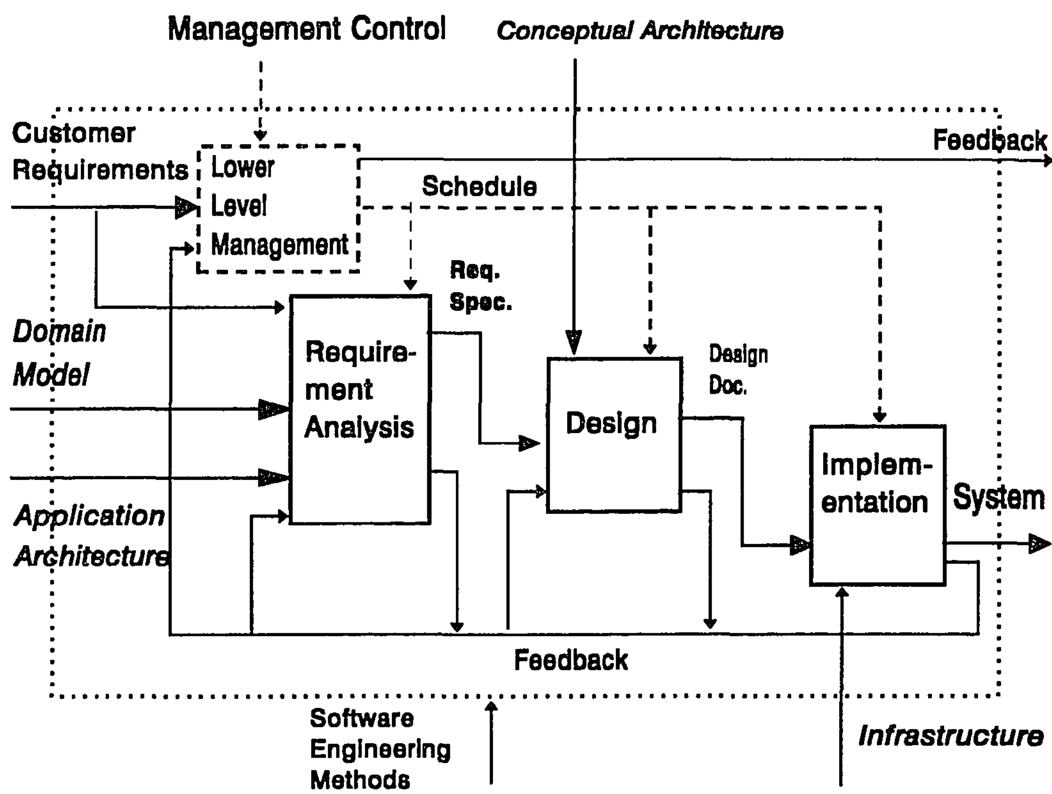

Figure A.9 System Task Process Diagram

A.10 Mega-System Synthesis

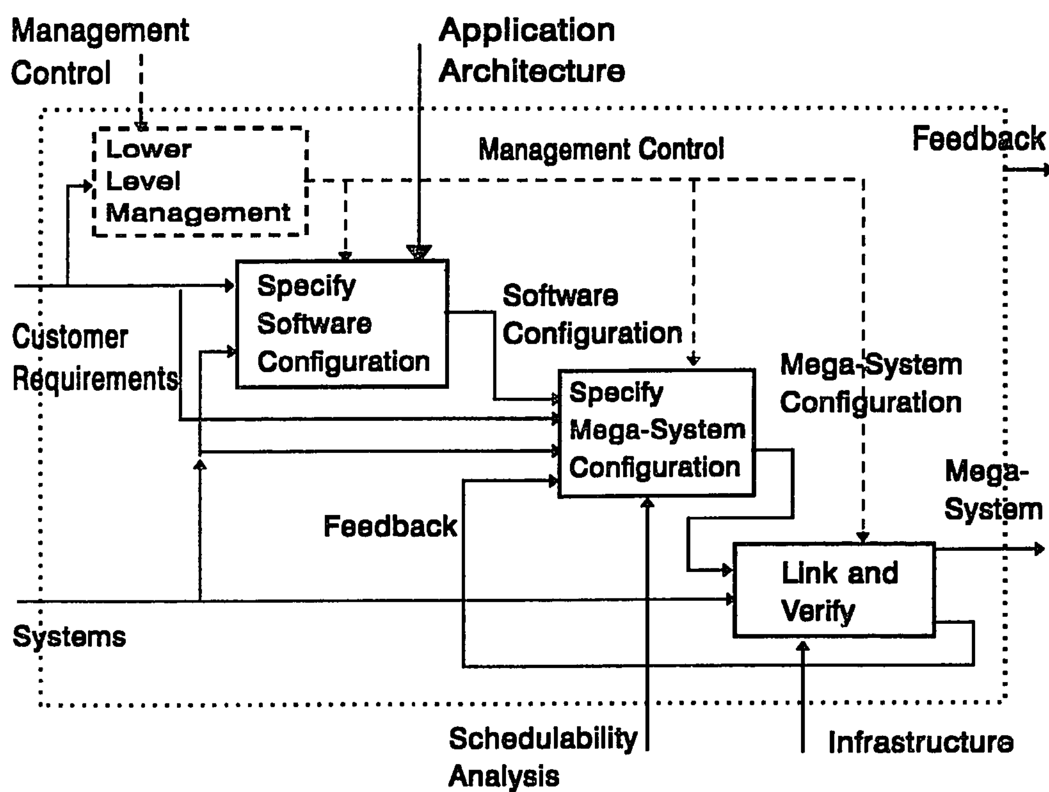

Figure A.10 Mega-System Synthesis Process Diagram

BIBLIOGRAPHY

1.[ACKE 92] Ackerman, L .F., "After Accolade: Time for New Law?," IEEE Software, November 1992, pp.100-192.

2.[ADOM 92] Adomeit, R., Deiters, W., Holtkamp, B., Schulke, F., Weber, H., "$K/2_R$: A Kernel for the ESF Software Factory Support," in Proceedings of IEEE Second International Conference on Systems Integration, Morristown NJ, June 1992, pp.325-336.

3.[AGAN 93] Aganovic, S., *Domain Analysis for the Insurance Domain,* Master Project, NJIT, 1993.

4.[ANSA 89] *ANSA: An Engineer's Introduction to the Architecture*, Release TR.03.02, Architecture Projects Managments Limited, November 1989.

5.[ANSA 92a] *An Overview of ANSAware 4.0*, Document RM.099.00, Architecture Projects Managments Limited, March 1992.

6.[ANSA 92b] *ANSAware 4.0 Application Programmer's Manual*, Architecture Projects Managments Limited, Document RM.102.00, March 1992.

7.[ANSI/IEEE Standard 729-1983] ANSI/IEEE Standard 729-1983, *IEEE Standard Glossary of software Engineering Terminology,* The Institute of Electrical and Electronics Engineering, Inc., NY, Approved by American National Standards Institute August 9, 1983.

8.[ARAN 91] Arango, G, and Prieto-Diaz, R., "Domain Analysis Concepts and Research," in Prieto-Diaz R. and G. Arango (editors), Domain Analysis and Software Systems Modeling, IEEE Computer Press, Los Almitos CA, 1991, pp. 9-26.

9.[ARTH 85] Arthur, L. J., *Measuring Programmer Productivity and Software Quality*, Wiley-Interscience, 1985.

10.[BABI 86] Babich, W. A., *Software Configuration Management, Coordination for Team Productivity,* Addison-Wesley, 1986.

11.[BATI 86] Batini, C., Lenzerini, M., Navathe, S. B., "A Comparative Analysis of Methodologies for Database Schema Integration," ACM Computer Surveys, Vol. 18, No. 4, December, 1986, pp. 323-361.

12.[BAKE 72] Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Vol. 11, No. 1, 1972, pp. 56-73.

13.[BEST 90] Best, L. J., *Application Architecture-Modern, Large-Scale Information Processing,* Wiley & Sons Inc., ISBN 0-471-51089-0, 1990.

14.[BETT 92] Betts, M., "House Bill Restricts Air Reservation Systems," Computerworld, Vol. 26, No. 33, pp. 1 and 16, August 1992.

15.[BHAN 93] Bhansali, P.V., "Survay of Software Safety Standards Shows Diversity," IEEE Computer, Vol. 26, No. 1, January 1993, pp. 88-89.

16.[BLUM 92] Blum, B. I., *Software Engineering a Holistic View*, Oxford University Press, 1992.

17.[BOEH 76] Boehm, Barry W., "Software Engineering," IEEE TR-C, December 1976, pp.1226-1241.

18.[BOEH 82] Boehm, B. W., *Software Engineering Economic*, Prentice Hall, Englewood Cliff, New Jersey, 1981.

19.[BOEH 88] Boehm, Barry W., "A Spiral Model of Software Development and Enhancement," IEEE Computer, Vol. 21, No. 5, May 1988, pp.61-72.

20.[BOEH 89] Boehm, Barry W., *Software Risk Management,* IEEE Computer Society Press, 1989.

21.[BOOC 91] Booch, G., *Object Oriented Design with Application*, Benjamin Cummings, Redwood City, CA, 1991.

22.[BROO 87] Brooks, F. P., Jr., "No Silver Bullet: Essence and Accidents of Software engineering," Computer, 20 (4), April 1987, pp. 10-19.

23.[CHAR 89] Charette, R. N., *Software Engineering Risk Analysis and Management,* McGraw Hill/Intertext, 1989.

24.[CLAR 92] Clarck, D., Krumm, J. M., Bieleski, S. T., "Broker: A System Integration Approach, in Proceedings of IEEE Second International Conference on Systems Integration. Morristown NJ, June 1992, pp. 162-170.

25.[CLEM 91] Clemons, E. K., "Evaluation of Strategic Investments in Information Technology," CACM, Vol. 34, No. 1, January 1991, pp. 20-36.

26.[COAD 91a] Coad, P., and Yourdon, E., *Object-Oriented Analysis*, second edition, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ , 1991.

27.[COAD 91b] Coad, P., and Yourdon, E., *Object-Oriented Design*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ , 1991.

28.[CSTB 90] Scaling Up: A Research Agenda for Software Engineering, Communication of the ACM, Vol. 33, No. 3, pp. 281-293, March 1990.

29.[CURT 88] Curtis, B., Krasner, H., and Iscoe, N., "A Field Study of the Software Design Process for Large Systems," Communication of the ACM, Vol. 31, No. 11, pp.1268-1287, 1988.

30.[CURT 92] Curtis, B., Kellner, M. I., and Over, J., "Process Modeling," in special issue on "Analysis and Modeling in Software Development," Communication to the ACM, September, 1992, pp. 75-90.

31.[DAVI 92] Davis, A. M., "Why Industry Often Says 'No Thanks' to Research," IEEE Software, November, 1992. pp.97-99.

32.[DALY 92] Daly, B. and Lindquist, C., "Judge Reaffirms Ruling Against Apple," Computer World, Vol, 26, No. 33, pp. 97, 1992.

33.[DEMA 78] De Marco, T., *Structured Analysis and System Specification*, Yourdon Press, 1978.

34.[DeMA 82] De Marco, T., *Controlling Software Projects,* Yourdon Press, 1982.

35.[DESA 92] Desai, S. and Mills, J. A., "Object-Orientedness and Large Scale Interoperability," Bellcore, 1992.

36.[DICK 78] Dickover, M. E., McGowan, C. L., and Ross, D. T. , "Software Design Using SADT," in Tutorial: *Software design Strategies,* second edition, Bergland, G.D. and Gordon, R.D., (edts), IEEE Computer Society Press, 1981.

37.[DoD-STD-2167a] *Defense System Software Development*, USA, Department of Defense, February 29, 1988.

38.[DUNN 90] Dunn, R., *Software Quality Assurance*, Prentice Hall, 1990.

39.[EISN 91] Eisner, H., Marciniak, J., and McMillan, R., "Computer Aided Systems of Systems (S2) Engineering," in Proceedings of the 1991 IEEE/SMC International Conference on Systems, Man, Cybernetices, Charlottesville, VA, IEEE, Computer Society Press, October 1991, pp. 531-537.

40.[ELMA 89] Elmazri, R., and Navathe, S., *Fundamental of Data-Base Systems,* Benjamin Cummings, 1989.

41.[ESF 89] *ESF Technical Reference Guide*, Version 1.1, ESF - Eureka Software Factory, 1989.

42.[ESF 90] *ESF - Project Overview 1990*, ESF - Eureka Software Factory, 1990.

43.[FRAN 92] France, R. B., Semantically Extended Data Flow Diagrams: A Formal Specification Tool, IEEE Transactions on Software Engineering, Vol. 18., No. 4, April 92.

44.[FRED 92a] Fredriksson, L.-B., "A CAN Kingdom," KVASER AB, Sweden, 1992.

45.[FRED 92b] Fredriksson, L.-B., "Some Thoughts About: CAN System Integration," KVASER AB, Sweden, 1992.

46.[FREE 87] Freeman, D. P., *Software Perspectives*, Addison-Wesley, Reading, MA, 1987.

47.[GARL 93] Garlan D., "The Need for A Science of Software Architecture," in Summary of the Dagstuhl Workshop on Future Directions in Software Engineering, Feb. 1992, Software Engineering Notes, Vol. 18, No. 1, January 1993, pp. 39.

48.[GELL 91] Geller, J. Perel Y., and Neuhold, E., "Structural Schema Integration with Full and Partial Correspondence using the Dual Model," Technical Report, Department of Computer & Information Science, NJIT, 1991.

49.[GELL 91a] Geller, J. Perel Y., and Neuhold, E., "Structure and Semantic in OODB Class Specification," in Special Issue: Semantic Issues in Multidatabase Systems, SIGMOND Record, ACM Press, Vol. 20., No. 4, pp. 40-43, December 1991.

50.[GELL 92] Geller, J, Mehta, A., Perl, Y. and Sheth, A., "Algorithms for Structural Schema Integration," in Proceedings of IEEE Second International Conference on Systems Integration, Morristown NJ, June 1992, pp. 604-614, 1992.

51.[GHED 91] Ghedina, J., and Oppenheim, D., "Software Engineering Prespectives, KPMG Peat Marwick's System Integration Framework," Software engineering, Sep/Oct 1991.

52.[GILB 88] Gilb, T., *Principles of Software Engineering Management,* Addison-Wesley, 1988.

53.[GOMA 90] Gomaa, Hassan, "The Impact of Prototyping on Software Systems Engineering," in Tutorial: *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman (eds), IEEE Computer Society Press, 1990, pp.543-552.

54.[GRIF 91] Griffin, M., *Lotus 1-2-3 Release 2.3 in Business*, SAMS, Carmel, IN, 1991.

55. [HALA 91] Halang, W. A., Stoyenko, A. D., *Constructing Predictable Real Time Systems*, Kluwer Academic Publishers, 1991.

56.[HERB 89a] Herbert, A. J., "The ANSA Project and Standards," in *Distributed Systems*, Mullender, S. (editor), ACM Press, Frontier Series, 1989, Chapter 17, pp. 391-399.

57.[HERB 89b] Herbert, A. J., "The Advanced Networked Systems Architecture," in *Distributed Systems*, Mullender, S. (editor), ACM Press, Frontier Series, 1989, Chapter 18, pp. 401-415.

58.[HERB 89c] Herbert, A. J., "The Computational Projection of ANSA," in *Distributed Systems*, Mullender, S. (editor), ACM Press, Frontier Series, 1989, Chapter 19, pp 417-437.

59.[HETZ 88] Hetzel, B., *The Complete Guide to Software Testing, Second edition*, QED Information Sciences, INC., Wellesley, MA, 1988.

60.[HUBE 90] Hubert, L., and Perdreau, G., "Software Factory: Using Process Modeling for Integration Process," in Proceedings of the First International Conference on Systems Integration, Morristown, NJ, IEEE Computer Society Press, April 1990, pp.14-25.

61.[HUMP 88] Humphrey, W. S., *"Characterizing the Software Process: A maturity Framework,"* IEEE Software, Vol. 5, No. 2, March 1988, pp.73-79.

62.[HUNT 87] Hunt, V., and Zellweger, A., "The FAA's Advanced Automation System: Strategies for Future air Traffic Control Systems," IEEE Computer, Vol. 20, No. 2,pp. 19-23, February 1987.

63.[ISCO 91] Iscoe, N., Williams, G. B., Arango, G., (eds), in Proceedings of the Domain Modeling Workshop, 13th International Conference on Software engineering, Austin, Texas, May, 1991.

64.[JONE 86] Jones, C., *Programming Productivity,* MacGraw-Hill, 1986.

65.[JOHN 92] Johnson, M., "Lotus Strikes Suit Deal with HP for Unix Application," Computerworld, Vol. 26, No. 33, Aug. 1992, pp. 8.

66.[JOSE 89] Joseph, T. A., and Birman, K. P., "Reliable Broadcast Protocols," in *Distributed Systems,* Mullender, S. (editor), ACM Press, Frontier Series, 1989, pp. 293-317.

67.[KIM 90] Kim, W., "Object Oriented Databases: Definition and Research Directions," Transactions on Knowledge and Data Engineering, IEEE, Vol. 2, No. 3, Sep. 1990.

68.[KLIN 90] Kling, R., "Information Systems, Social Transformations, and Quality of Life," in Proceedings of the Conference on Computers and Quality of Life, 1990, pp. 76-85.

69.[KOKO 89] Kokol, P., "Metamodels for system development," Software Engineering Notes, Vol. 14, No. 5, ,July 1989, pp. 118-123.

70.[KRAS 92] Krasner, H., "The ASPIRE Approach to Continuous Software Process Improvement," in Proceedings of IEEE Second International Conference on Systems Integration, Morristown NJ, June 1992, pp. 193-201.

71.[LAWS 92a] Lawson, H. W., "Application Machines: An Approach to Complexity Reduction," CBSE workshop, March 1992.

72.[LAWS 92b] Lawson, H. W., "Application Machines: An Approach to Realizing Understandable Systems," Keynote at the Euromicro Conference, Paris, September, 1992.

73.[LAWS 92c] Lawson, H. W., "Engineering Predictable Real-Time Systems," Lecture Notes for the NATO Advanced Study Institute, Real-Time Computing, October, 1992.

74.[LEHM 90] Lehman, M. M., "Software Engineering - The Role Of CASE," in CASE '90, Irvine, December 1990.

75.[LUND 91] Lundell, J., Notess, M. "Human Factors in Software Development Models, Techniques, and Outcomes," in Proceedings of Computer Human Interaction CHI' 91 - Human Factors in Computing Systems, 1991, pp. 145-151.

76.[MART 91] Martin, J. with Chapman, K.K., and Leben, J. *"Systems Application Architecture: Common User Accesss,"* Prentice Hall, Englewood Cliffs, NJ, 1991.

77.[MAYE 89] Mayers, W. "Software Pivotal to Strategic Defense," IEEE Computer, Vol. 22, No. 1, January, 1989, pp.92-97.

78.[MILL 90] Mills, J.A. "The Operations Systems Computing Architecture: Semantic Integrity of Totality of Corporate Data," in Proceedings of the First International Conference on Systems Integration, Morristown, NJ, IEEE Computer Society Press, April 1990,pp. 482-491.

79.[MITT 87] Mittermeir R.T. and Rossak, W., "Software Base and Software Archives - Alternatives to Support Software Reuse," in Proceedings of the FJCC 87, Dallas, TX, October, 1987.

80.[MITT 91] Mittermeir R.T, "POWDER - A Recursive Methodology for Prototyping of Wicked Development Efforts with Reuse," Institut f. Informatik, Universitaet Klagenfurt, Austria, Report for International Software Systems Inc., Austin TX, April 1991.

81.[MONA 92] Monarchi, D. E., and Puhr, G. I., "A Research Typology for Object-Oriented Analysis and Design," in a special issue on "Analysis and Modeling in Software Development," Communication to the ACM, September, 1992, pp 35-47.

82.[MOOR 92] Moorehead, R., Keynote Address, the Second International Conference on Systems Integration (ICSI '92), Morristown, NJ, 1992.

83.[MOSE 92] Moser, R., "Working Together," in Output+Micro, Computer & Communication Osterreich, pp. 16-17, June 1992.

84.[NEFF 92] Neff R., "American Airlines Software Development Woes," in "Risk to the Public in Computers and Related Systems," Neumann, P.G. (moderaator), Software Engineering Notes, Vol. 17, No. 4, 1992, pp. 16-17.

85.[NEIG 81] Neighbors, J., *Software Construction Using Components,* doctoral dissertation, Univ. of California, Irvine, Calif., 1981.

86.[NIST 91] NIST Special Publication 500-201, Reference Model for Framework of Software Engineering Environment, (Technical Report ECMA TR/55, 2nd Edition), NIST, 1991.

87.[NOTK 88] Notkin, D., Black, A.P., Lazowska, E. D., Levy, H. M., Sanislo, J., and Zahorjan, J., "Interconnecting Hetrogeneous Computer Systems," CACM, Vol. 31, No. 3, pp. 258-273, March 1988.

88.[NEUM 91] Neumann, P. G., "The Not-So-Accidental Holist," Inside Risk, Communication of the ACM, Vol. 34, No. 9, Septmber 1991, pp. 122.

89.[ODSG 78] Datenschutzgestz - DSG Bundesgestzblatt fuer die Republik Oesterreich, Jahrgang 1978, 193, Stueck, 28. November, 1978.

90.[OSCA 92] *The Belcore* OSCA™ *Architecture,* Bellcore - Bell Communication Research, Technical Advisory, TA-STS-000915, ISSUE 3, March 1992.

91.[PAGE 80] Page-Jones, M., *The Practical Guide to Structured Systems Design,* Yourdon Press, 1980.

92.[PAGE 85] Page-Jones, M., *The Practical Project Management,* Dorset House, 1985.

93.[PARN 76] Parnas, D. L., "On the Design and Development of Program Families," TR-SE, Vol. SE-2/1, March 1976, pp. 1-9.

94.[PERR 89] Perry, D. E.(editor), "Experience with Software Process Models," in Proceedings of the 5th International Software Process Workshop, IEEE Computer Society Press, October 1989.

95.[PERR 89a] Perry, D. E., "Introduction" in Proceedings of the 5th International Software Process Workshop, IEEE Computer Society Press, October, 1989, pp. 3-5.

96.[PERR 92] Perry, D. E., and Wolf, A. L., "Foundation for Study of Software Architecture," Software Enginnering Notes, Oct, 1992, pp. 40-52.

97.[POWE 90] Power, L. R., "Post-Facto Integration Technology: New Discipline for an Old Practice," in Proceedings of the First International Conference on Systems Integration, Morristown, NJ, IEEE Computer Society Press, April 1990, pp.4-13.

98.[PRES 92] Pressman, R. S., *Software Engineering a Practitioner's Approach,* third Edition, McGraw-Hill, Inc., 1992.

99.[PRIE 90] Prieto-Diaz R., "Domain Analysis: An Introduction," Software Engineering Notes, Vol. 15. No. 2, April 1990, pp.47-54.

100.[PRIE 91a] Prieto-Diaz R. and G. Arango (editors), Domain Analysis and Software Systems Modeling, IEEE Computer Press, Los Almitos CA, 1991.

101.[PRIE 91b] Prieto-Diaz R., "Making Software Reuse Work: An Implementation Model," Software Engineering Notes, Vol. 16(3), Jul 1991, pp.61-68.

102.[PRIE 91c] Prieto-Diaz, R., "A Domain Analysis Methodology," in Proceedings Domain Modeling Workshop, 13th International Conference on Software Engineering, Austin, Texas, May 1991.

103.[RETT 90] Retting, M., "Software Teams," CACM, Vol. 33, No. 10, pp. 23-27, October 1990.

104.[RHEI 91] Rheingold, H., *Virtual Reality,* Summit Books, 1991.

105.[ROSE 89] Rose, M. T., *The Open Book A Practical Perspective on OSI,* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

106.[ROSS 77] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transaction on Software Engineering, January 1977, pp 16-33.

107.[ROSS 87a] Rossak, W., and Mittermeir, R. T., "Structuring Software Archives for Reusability," Proc. IASTED 5th international Symposium on Applied Informatics, Grindelwald, Switzerland, 1987.

108.[ROSS 91a] Rossak, W., and Ng, P. A., "Some Thoughts on System Integration - A Conceptual Framework," International Journal of Systems Integration, Vol. 1, No. 1, 1991, pp.97-114.

109.[ROSS 91b] Rossak, W., and Prasad, S., "Integration Architectures - A Framework for Systems Integration Decisions," Proc. of the IEEE Internat. Conference on Systems, Man, and Cybernetics, Charlottesville VA, October 1991, pp. 545-550.

110.[ROSS 91c] Rossak, W., "Integration Architectures - A Concept and A Tool to Support Integrated Systems Development," Technical Report, Department of Computer & Information Science, NJIT, 1991.

111.[ROSS 92a] Rossak, W., and Ng, P. A., "System Development with Integration Architectures," in Proceedings of IEEE Second International Conference on Systems Integration, Morristown NJ, June 1992, pp. 96-103.

112. [ROSS 92b] Rossak W., Zemel T., Ng. P. A., "Systems Integration - A Framework," Tutorial on Systems Integration for the IEEE Second International Conference on Systems Integration, Morristown NJ, USA, June 1992.

113. [ROSS 92c] Rossak W., Welch L., Zemel T., and Eder J., " A Generic Systems Integration Framework for Large and Time-Critical Systems," in Halang W., Stoyenko A. (eds.): NATO Advanced Study Institute (NATO ASI 910698) in Real-Time Computing, Mullet Bay, Saint Martaan, Springer Verlag, October 1992, to appear.

114.[ROSS 93a] Rossak W., and Zemel, T. "Engineering Large and Complex Systems with Integration Architectures," PD-Vol. 49 - Computer Applications and Design Abstractions, ETCE '93, Houston TX, USA, February 1993, pp. 189-195.

115. [ROSS 93b] Rossak W., Zemel T., and Lawson H., "A Meta-Process Model for the Planned Development of Integrated Systems," International Journal of Systems Integration, Kluwer Academic Publ., Dordrecht, Holland, 1993, to appear.

116.[RUMB 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, and F., Lorensen, W., "Object-Oriented Modelling and Design," Prentice Hall, 1991.

117.[SALK 91] Salkind, N., *WordPerfect 5.1 in Business*, SAMS, Carmel, IN, 1991.

118.[SAMU 92] Samuelson, P., "Updating the Copyright Look and Feel Lawsuits," in Communications of the ACM, Spetember 1992, pp. 25-31.

119.[SCHA 90] Schafer, W., and Weber, H., "European Software Factory Plan - The ESF Profile," in *Modern Software Engineering,* Ng, P.A., and Yeh, R.T., (eds), Van Nostrand Reinhold, 1990, pp. 613-637.

120.[SCHL 92] Schlemmer, R. A., "Software Process Improvement - Optimizing Your Process, An instrument for development, Assessment, and Improvement of Software Processes," Master Thesis, Universitaet Klagenfurt, 1992.

121.[SHAW 89] Shaw. M., "Larger Scale Systems Require Higher-Level Abstractions," in Proceedings Fifth International Workshop on Software Specification and Design, Pittsburg, PA, May 1989, pp.140-146.

122.[SHET 88] Sheth, A. P., Larson, J. A., Cornellio, A., and Navethe, S., "A Tool for Integrating Conceptual Schemas and User Views," Proceedings of the Fourth International Conference on Data Engineering, Los Angeles, CA, Feb 1988, pp. 176-183.

123.[SHET 90] Sheth, A. P., and Larson, J. A., "Federated Databases Systems for Managing Distributed, Hetrogeneous, and Autonomous Databases," ACM computer surveys, Vol. 22, No. 3, September 1990.

124.[SHLA 92] Shlaer, S., Mellor, S. J., *Object Lifecycles - Modeling the World in States,* Yourdon Press Computing Series, 1992.

125.[SMIT 87] Smith, B. D., Triechmann, J.S., Wiening, E. A., *Property and Liability Insurance Principles,* Insurance Institute of America, Malvern, Pennsylvania, 1987.

126.[SS2000a] *Introduction to Ship System 2000*, NobelTech.

127.[SS2000b] *Ship System 2000 Operational Concepts,* NobelTech.

128.[TAYL 92] Taylor, I., "A Process Reference Model For Large-Scale Software Development," in Proceedings of IEEE Second International Conference on Systems Integration, Morristown NJ, June 1992, pp. 268-274.

129.[TANE 92] Tanenbaum, A. S., *Modern Operating Systems,* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

130.[THAY 90] Thayer, R. H., and Thayer, M. C., "Glossary," in Thayer, R.H., and Dorfman, M., (eds.), System and Software Requirements Engineering, IEEE Computer Society Press Tutorial, 1990, pp. 605-676.

131.[THIM 92] Thimm, H., "Domain Analysis for Systems Integration," Master's Thesis, Department of Computer and Information Science, NJIT, 1992.

132.[TICH 92] Tichy, W. F., Habermann, N., and Prechelt, L., "Summary of the Dagstuhl Workshop on Future Directions in Software Engineering," Feb. 1992, in Software Engineering Notes, Vol. 18 No. 1, Jan. 1993, pp. 35-39.

133.[TRAC 91] Tracz, W., "A Conceptual Model for Megaprogramming," Software Engineering Notes, Vol. 16. No. 3, July 1991, pp.36-45.

134.[TULL 88] Tully, C., (editor), "Representing and Enacting the Software Process," Proceedings of the 4th International Software Process Workshop, Devon, UK, ACM Press, May, 1988.

135.[ULLM 88] Ullman, J., D., *Principles of Database and Knowledge-Base Systems,* Vol. 1, Computer Science Press, 1988.

136. [VEIJ 88] Veijalainen, J. and Poescu-Zeletin, R., "Multidatabase systems in ISO/OSI environment. In *Standards in Information technology and Industrial Control,* Malagardis, N., and Williams, T., (eds), North-Holland, The Netherlands, pp. 83-97, 1988.

137.[WARD 86] Ward, P. T., and Mellor, S. J., *Structure development for real-time systems,* , Vol. 1, 2, 3., Yourdon Press, Englewood Cliffs, NJ, 1986.

138.[WEBS 91] Webster's College Dictionary, Random House, 1991.

139.[WIED 92] Wiederhold, G., Wegner, P., and Ceri, S. "Toward Megaprogramming," Communication of the ACM, Vol. 35, No. 11, Nov. 1992 , pp. 89-99.

140.[WIMM 92] Wimmer, K., and Wimmer, N., Conceptual Modelling Based on Ontological Principles, submitted to Knowledge Acquisition, 1992.

141.[YEH 80] Yeh, R. T., and Mittermeir, R. T., "Conceptual Modeling as a Basis for Deriving Software Requirements," International Computer Symposium, Taepei, Taiwan, December, 1980. pp. 1-14.

142.[YEH 91] Yeh, R. T., Schlemmer, R. A., and Mittermeir, R. T., "A Systematic Approach to Process Modeling," Journal of System Integration, Vol. 1, No. 3/4, pp. 265-282, Nov. 1991.

143.[YOUR 89] Yourdon, E., *Modern Structured Analysis,* Prentice Hall, 1989.

144.[YOUR 92] Yourdon, E., *Decline and Fall of the American Programmer*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, NJ, 1992.

145. [ZEME 92c] Zemel T., and Rossak, W.,"Mega-Systems - The Issue of Advanced Systems Development," in Proceedings of IEEE Second International Conference on Systems Integration, Morristown NJ, June 1992, pp. 548-555.

146. [ZEME 92b] Zemel T., Rossak W., and Thimm H., "Domain Analysis as a Major Component of Integrated Systems Development," in Proceedings of SERF '92, 1992 Software Engineering Research Forum, Indialantic FL, November 1992, pp. 217-224.

147. [ZEME 92a] Zemel T., and Rossak W., "A Framework for the Development of Complex Computer Based Systems as Mega-Systems," Workshop on Computer Based Systems Engineering, London, England, December, 1992.

GLOSSARY

This glossary includes definitions of the concepts used in MegSDF. It relies on the glossary found in the "Systems and Software Requirements Engineering - IEEE Computer Society Press Tutorial" written by Richard H. Thayer and Mildred C. Thayer; these definitions are designated by [THAY 90]. The same convention is followed with definitions drawn from other sources. Concepts new to MegSDF or concepts adapted for MegSDF are designated by [*]. Other definitions represent standard terminology.

1. Application Domain

An application domain is a comprehensive, internally coherent, relatively self-contained area or business enterprise supported by software systems. An application domain consists of phenomena of various types, e.g., objects, relations, constraints, activities, and processes. See chapter 5. [*]

A domain is a separate real, or hypothetical, or abstract world inhabited by a distinct set of objects that behave according to rules and policies characteristic of the domain. [SHLA 92]

2. Application Architecture [*]

An application architecture defines the boundaries of the Mega-System within the application domain, the various systems and components of the Mega-System, their

interfaces, and the services provided by each component. It is part of the Mega-System architecture, an output of the Mega-System architecture design task.

The application architecture is used by the Mega-System synthesis and the systems tasks as a reference model and guideline. It is an instantiation of the conceptual architecture. The application architecture is domain specific as opposed to the conceptual architecture. See chapter 6.2.3.

3. Autonomous System

An autonomous system is a system developed to work on its own, independently of other systems. An autonomous system has both self-contained functionality and self-contained technical environment. The development of an autonomous system is usually done by a software team and according to team standards and procedures. See chapter 2 and 3.

4. Architectural Style

Architectural styles are common software architectures defining general design and implementation concepts. The architectural styles are used as inputs to the conceptual architecture design task. See section 6.2.3.

5. Conceptual Architecture [*]

A conceptual architecture defines common design and implementation concepts for the Mega-System, as well as guidelines for deriving the application architecture of the Mega-System. It is a part of the Mega-System architecture, an output of the Mega-System

architecture design task. MegSDF proposes dividing the concepts of the conceptual architecture into interrelated views, e.g., structural, communication, control, data, and environment. The conceptual architecture is a specialization of an appropriate architectural style.

The infrastructure and the domain model have a major influence on the conceptual architecture since they are used as inputs to the decision process that selects the conceptual architecture. The conceptual architecture is used as an input for the infrastructure acquisition and the system tasks developing constituent systems of the Mega-System. See section 6.2.2.

6. Derived From [*]

The relationship between a set of functionalities specified at a conceptual level and a system developed as an instantiation of this set of functionalities, or the specialization of such a set of functionalities followed by their instatiation. See section 2.3.

7. Dimension [*]

A part of the domain model, consisting of interrelated elements representing the phenomena of the domain. The number of dimensions in a domain model, and their content depend on the actual domain and the modeling approach. See section 5.2.

8. Domain Analysis [*]

Domain analysis is one of the Mega-System design tasks. In this task an application domain is abstractly modeled. Each phenomenon in the domain is represented as an element and described from various aspects. *To achieve a comprehensive model of the domain we propose viewing the domain from different perceptions and then integrating these perceptions into a unified model. Unlike domain analysis methods used for reusability, MegSDF domain analysis does not include constructive issues, e.g, design and implementation, for the domain.* See section 5.2.

9. Domain Model [*]

An output of domain analysis which describes the phenomena for a specific application domain from different points of view (perceptions). Each phenomenon is represented by an element and is described from various aspects. Possible phenomenon types are objects, relations, processes, and constraints. The domain model is used by the Mega-System architecture design task as an input that affects the choice of Mega-System architecture. Parts of the domain model are used by the various systems tasks as inputs for the requirement analysis phase. See section 5.2.

10. Domain Schema [*]

A set of modeling primitives (element-types) to be used for building a domain model. A domain-schema is defined on the basis of a suitable modeling approach and the actual

domain. MegSDF proposes dividing the schema into domain schema dimensions each consisting of interrelated element-types. See section 5.2.

11. Element[*]

An element is a component of the domain model representing a phenomenon of the domain. See section 5.2.

12. Enabling Technology

An enabling technology is one that makes another technology or technologies possible. [RHEI 91]

Enabling technologies are the diverse parts of the infrastructure. Their integration enables the development and operation of Mega-Systems. These technologies are the mechanisms that support the implementation and the integration of the various systems.[*] See chapter 7.

13. Environment

An environment is the circumstance under which a software system operates, consisting of processors, operating systems, programming languages, and development and debugging tools. See chapters 1 and 3.

14. Framework [*]

A framework is a comprehensive approach or reference model in a domain. It defines a set of mutually integrated methods and concepts and is used for the solution of a complex problem.

We propose MegSDF as a framework for the development of Mega-Systems. It is composed of engineering, management, and technical aspects. It is used as a guideline in the planning and execution of a process for developing a Mega-Systems. See chapter 3.

15. Generic

Of, pertaining to, or applicable to all the members of a genus, class, group, or kind.[WEBS 91]. See section 2.3.

16. Generic System

A generic system is a specification of a set of interrelated functionalities and the actual systems derived from this specification. A functionality is specified on an abstract and conceptual level by natural languages or formal definitions. Different systems are derived by an instantiation or specialization of the abstract functionalities.

Generally, a generic system is developed by several software teams belonging to the same organization. Each software team develops a derived system as an independent project. See section 2.3.

17. Generic System of Systems[*]

Generic Systems of systems is a subclass of the system of systems and generic systems classes. A Generic system of systems is developed for a domain without a specific technical environment, time frame, or customers, and can have multiple configurations consisting of several systems at a given point of time. See section 2.4.

18. Homogeneous Environment

A homogeneous environment is an environment in which a software system operates, consisting of one type of operating system, a specific set of tools, one language, and a homogeneous hardware configuration. See chapter 1.

19. Heterogeneous Environment

A heterogeneous environment is an environment, in which a software system operates consisting of several different operating systems, a mixed set of tools, various programming languages, and several hardware configurations. See chapter 1.

20. Heterogeneous User Group

A heterogeneous user group is a group of users of a software system characterized by a large number of users with diverse roles located in diverse sites such users have no common user profile or fixed requirements. See chapter 1.

21. Huge System

A huge system is a large, complex software system developed by a large software team and/or over a long period. These systems usually solve a particular problem for a well-defined user group. A huge system is composed of multiple subsystems, where each subsystem is designed and developed only as a part of the whole system. A huge system operates in homogeneous or heterogeneous environments.

Huge system are generally developed as one large project. However, we propose developing huge systems as systems of systems. Huge systems are type of Mega-System. See chapter 2.1.

22. Infrastructure [*]

An infrastructure is an environment that integrates all enabling technologies that facilitate the development and operation of a Mega-System. It is chosen in the infrastructure acquisition task. Essential enabling technologies of an infrastructure include communication, database management system, and user interface. The infrastructure forms a stable layer between the various constituent systems of the Mega-System and the enabling technologies.

The infrastructure is used as a mechanism for the development of the constituent systems and for their integration in the Mega-System synthesis tasks. It is an input for the Mega-System architecture design task and has a major influence on the architecture of the Mega-System. See chapter 7.

23. Infrastructure Acquisition [*]

The infrastructure acquisition task includes choosing, developing or purchasing, validating, and supporting an infrastructure that integrates the enabling technologies into a unified platform. This process is based on the conceptual architecture of the Mega-System and aims at selecting an infrastructure.

Currently, only a few infrastructures exist and therefore an infrastructure typically must be developed. We believe that in the future infrastructures will be standard products, hence this task will tend to be solely a decision and certification process and will not involve development. See chapter 7.

24. Instantiation [*]

A system S is an instantiation of a specification of a set of functionalities if S implements these functionalities using a specific algorithm, programming language, or hardware configuration. See section 2.3.

25. Integrate-to [*]

A relation between a constituent system Si and its parent Mega-System P, wherein the system Si is integrated with other constituent systems {S1, S2, ..., Si-1,, Si+1, ..., Sn} to form the Mega-System P. A system S that is "integrated to" might be considered as a stand-alone system. Each constituent system is developed by a separate team, under different management, procedures and standards, and with its own schedule. However, all

the constituent systems operate as a coherent larger system after the integration. See section 2.2.

26. Integration

The act or instance of incorporating or combining into a whole. [WEBS 91]

In software development, integration is performed at several phases of the life cycle. Different types of components include lines of code, modules, subsystems, and the most sophisticated components - systems. The integration of components developed by different individuals is very difficult. It is further complicated when components are developed by different groups, using different standards and procedures, working at diverse sites. [*] See section 2.2.

27. Mega-Project[*]

A project for the development of a Mega-System. It includes multiple projects that develop the constituent systems of the Mega-System. The mega-project is managed by a meta-management. *We recommend developing a mega-project in accordance with the process model of MegSDF.* See section 3.3.

28. Mega-System[*]

Mega-Systems are large, complex software systems with one or more of the following characteristics:

- Consist of more than one system,

- Developed by more than one group of developers,

- Have a large and heterogeneous group of users,

- Have More than one customer,

- Operate in a heterogeneous technical environment.

We propose development of a Mega-System using MegSDF as a mega-project controlled by a meta-level management with multiple projects for the development of the constituent systems. We identify huge systems, systems of systems, and generic systems as subclasses of the Mega-System class. See chapter 2.

29. Mega-System Architecture Design Task

Mega-system architecture design is one of the Mega-System tasks. It plans the Mega-System as a whole and includes the specification of common design and implementation concepts, definition of Mega-System boundaries, allocation of domain elements to systems, and definition of systems interfaces. The inputs to this task are the domain model and the chosen infrastructure. The output of this task is a Mega-System architecture, including a conceptual and an application architecture. See section 6.2.2.

30. Mega-System Tasks[*]

The Mega-System tasks are a group of tasks in the engineering process for the development of a Mega-System including domain analysis, infrastructure validation, and the Mega-System architecture design. *The tasks in this group are essential for Mega-*

System development and have the role of engineering coordination of the various systems tasks and of the whole process. See section 3.3.

31. Mega-Systems Synthesis [*]

Mega-System Synthesis is a task for forming a Mega-System from its constituents. It includes specification of software and hardware configuration based on analysis of non-functional customer requirements and the application architecture, and an instantiation and customization of the components according to these requirements. See section 8.3.

32.MegSDF [*]

A framework for development of Mega-Systems. It incorporates engineering, managerial, and technological aspects and focuses on an engineering process. The engineering process consist of the required activities for development of Mega-Systems and emphasizes the engineering coordination of the development of constituent systems.

33. Meta-Management[*]

The meta-management is the organizational unit responsible for the development of a Mega-System. The meta-management plans the development of the entire Mega-System and controls the Mega-System tasks and the various systems tasks. It is responsible for determining policies, directions, and the global schedule, and for allocating resources based on actual domain needs. See sections 3.3, 8.1.

34. Meta-Management Task

A group of tasks performed by the meta-management for controlling the process of a Mega-System development. These tasks are considered as scaled up traditional management tasks and include scheduling, budgeting, quality assurance, and configuration management. See section 8.1.

35. Method

A detailed approach for solving an engineering problem.[THAY 90]

A procedure, technique, or a planned way of doing something.[WEBS 91]

36. Methodology

A general approach for solving an engineering method.[THAY 90]

A set or system of methods, principles, and rules used in a given discipline, as in the arts or science.[WEBS 91]

37. Module

A program unit that is discrete and identifiable with respect to compiling, linking, and loading.[THAY 90]

A logically separable part of a program. [ANSI/IEEE Standard 729-1983]

A module is linked with other modules to form a software subsystem or system.

38. Part-of

A relation between a subsystem s_i and its parent system S. The subsystem s_i is linked with other subsystems $\{s_1, s_2, .., s_{i-1}, s_{i+1}, .., s_m\}$ to form the system S. Each subsystem of the system is planned and developed to work only in the context of the whole system. See section 2.1.

39.Perception

A representation of a domain from the specific point of view of a significant perceiver; modelled using perception-elements that represent relevant phenomena of the domain. See section 5.2.

40. Post-facto Integration

Post-facto integration is the process of systems integration in which the constituent systems were developed before the design of the entire system of systems. Each system was developed as an autonomous system. This contrasts with pre-facto integration in which the system of systems is known in advance and the constituent systems are developed in the context of the system of systems. A synonym for post-facto integration is a posteriori integration. See section 3.3.6.

41. Pre-facto Integration

Pre-facto integration is a process of systems integration in which the decision on the organization of the system of systems, as well as all components of the system are known

in advance and the constituent systems are designed to work in the context of the system of systems. A synonym for pre-facto integration is a priori integration. See section 3.3.6.

42. Pre-Planned Approach [*]

A fundamental design principle for development of software systems. According to this principle, even though no known requirements exist in advance, each system is designed to efficiently accommodate the following operations:

- integration with other systems,
- extensions of the system with new functionalities, and
- customization of the system with user selected actual parameters.

We propose domain analysis, Mega-System architecture design, and infrastructure acquisition tasks to support this principle. See Section 3.3.6.

43. Process

A set of activities (tasks) whose execution is required to achieve a specific goal. A software development process includes all those activities which are required to build a software system. See section 4.

44. Process Model

A representation of a system/software development process activity intended to explain the behavior of some its aspects.[THAY 90]

We propose an engineering process model for the development of a Mega-System. This model includes the definition of the various activities, their relations, and the data and control flows between them. See section 4.

45. Project

A project is the set of activities, functions, tasks, both technical and managerial, required to satisfy the terms and conditions of the project agreement. It is a temporary activity, characterized by having a starting date, specific objectives and constraints, established responsibilities, a budget and schedule, and a completion date. [THAY 90]

In megSDF, a project is responsible for developing a constituent system or synthesizing a Mega-System to a specific customer. See section 3.3.

46. Risk Analysis

The methodical process of identifying:

- areas of potential risks,

- the associated probability of occurrence, and

- the seriousness of the consequence of the occurrence.

[THAY 90]. See section 8.1.

47. Service Group[*]

A service group is a part of the infrastructure consisting of a set of interrelated services offered by the infrastructure. *MegSDF proposes that service groups correspond to the architectural views.* See section 6.2.

48. Software Engineering

1. The practical application of computer science, management, and other sciences to the analysis, design, construction, and maintenance of software and its associated documentation.

2. The systematic application of methods, tools, and techniques to achieve a stated requirement or objective for effective and efficient software systems. [THAY 90]

The application of methods, tools, and disciplines to produce and maintain an automated solution to a real-world problem. [BLUM 92]

49. Software Configuration Management (SCM)

The discipline of identifying the configuration of a software system at discrete points in time with the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system life-cycle. [THAY 90]

50. Software Quality

Software quality is the degree to which software possesses a desired combination of attributes.

Attributes of software that affect its perceived value, for example, correctness, reliability, maintainability, and portability. [ANSI/IEEE Standard 729-1983]

51. Software Quality Assurance (SQA)

A planned and systematic pattern of all actions necessary to provide adequate confidence that the software and the delivered documentation conform to the established technical requirements. [ANSI/IEEE standard 729-1983]

52. Software System

A software system is a collection of software modules/subsystems linked together to accomplish some common objectives.

A software system is developed by a software team. It is designed to work on its own and for a specific purpose. Any subsystem or module in a system is developed to work as a part of the entire system.

53. Structured Analysis (SA)

A software analysis technique that uses data flow diagrams (DFDs), data dictionaries, and process descriptions to analyze and represent software requirements.[THAY 90]

54. Structured Analysis and Design Technique (SADT)

SADT is a framework for the analysis and design activities of software system development. It is based on graphical notations drawn as a hierarchy of diagrams. It also defines the various personnel roles in a software project. [THAY 90].

55. Sub-system

A set of modules, sub-systems, or both, functionally related and with high coupling. Sub-systems are linked together to form a system. Usually, a sub-system is developed by one software team. See section 2.1.

56. System Architecture

In systems engineering, the structure and relationship among the components of a system. The system architecture may also include the system's interface with its operational environment. [ANSI/IEEE Standard 729-1983]

57. Systems Integration[*]

System integration is the process of planning, implementing, and maintaining a system of systems. This process might be considered as the most sophisticated level of integration, where the components for integration are stand-alone systems.

58. System of Systems[*]

A system of systems integrates several systems to form a larger system. The coupling between the various systems which form the system of systems is low. The constituent systems are developed independently, by various software teams. See section 2.2.

59. System Task[*]

A system task is one of the engineering sub-processes of MegSDF. It includes the development or maintenance of a constituent of a Mega-System as a project. There may be multiple concurrent instances of a system task. System tasks can apply any traditional software systems development approach and are controlled by the meta-management task. See section 8.2.

60. Task [*]

An activity in an engineering process with a specific objective and schedule. Several tasks performed to achieve a particular purpose constitute a process. A complex task may be decomposed into several sub-tasks. See chapter 4.

61. Technique

A technique is the body of specialized procedures and methods used in a specialized field, especially in an area of applied science. [WEBS 91]

62. Tool

A tool is a step-by-step, formalized, manual, or automated process for solving an engineering problem. [THAY 90]

Anything used as a mean of accomplishing a task or a purpose. [WEBS 91]

63. View[*]

A view in MegSDF is a part of the conceptual architecture, consisting of a set of interrelated design and implementation concepts. The conceptual architecture is divided into several views and all views together form a conceptual architecture. *MegSDF proposes dividing the conceptual architecture into structural, communication, control, data, and environment views.* See section 6.2.