Fall 1-31-1999

Domain architecture a design framework for system development and integration

Vassilka D. Kirova
New Jersey Institute of Technology

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

ABSTRACT

DOMAIN ARCHITECTURE: A DESIGN FRAMEWORK FOR SYSTEM DEVELOPMENT AND INTEGRATION

by
Vassilka D. Kirova

The ever growing complexity of software systems has revealed many shortcomings in existing software engineering practices and has raised interest in architecture-driven software development. A system's architecture provides a model of the system that suppresses implementation detail, allowing the architects to concentrate on the analysis and decisions that are most critical to structuring the system to satisfy its requirements. Recently, interests of researchers and practitioners have shifted from individual system architectures to architectures for classes of software systems which provide more general, reusable solutions to the issues of overall system organization, interoperability, and allocation of services to system components. These generic architectures, such as product line architectures and domain architectures, promote reuse and interoperability, and create a basis for cost effective construction of high-quality systems. Our focus in this dissertation is on domain architectures as a means of development and integration of large-scale, domain-specific business software systems.

Business imperatives, including flexibility, productivity, quality, and ability to adapt to changes, have fostered demands for flexible, coherent and enterprise-wide integrated business systems. The components of such systems, developed separately or purchased off the shelf, need to cohesively form an overall computational environment for the business. The inevitable complexity of such integrated solutions and the highly-demanding process of their construction, management, and evolution support require new software engineering methodologies and tools. Domain

architectures, prescribing the organization of software systems in a business domain, hold a promise to serve as a foundation on which such integrated business systems can be effectively constructed.

To meet the above expectations, software architectures must be properly defined, represented, and applied, which requires suitable methodologies as well as process and tool support. Despite research efforts, however, state-of-the-art methods and tools for architecture-based system development do not yet meet the practical needs of system developers.

The primary focus of this dissertation is on developing methods and tools to support domain architecture engineering and on leveraging architectures to achieve improved system development and integration in presence of increased complexity. In particular, the thesis explores issues related to the following three aspects of software technology: system complexity and software architectures as tools to alleviate complexity; domain architectures as frameworks for construction of large scale, flexible, enterprise-wide software systems; and architectural models and representation techniques as a basis for "good" design. The thesis presents an architectural taxonomy to help categorize and better understand architectural efforts. Furthermore, it clarifies the purpose of domain architectures and characterizes them in detail.

To support the definition and application of domain architectures we have developed a method for domain architecture engineering and representation: GARM-ASPECT. GARM, the Generic Architecture Reference Model, underlying the method, is a system of modeling abstractions, relations and recommendations for building representations of reference software architectures. The model's focus on reference and domain architectures determines its main distinguishing features: multiple views of architectural elements, a separate rule system to express constraints on architecture element types, and annotations such as "libraries" of patterns and

"logs" of guidelines. ASPECT is an architecture description language based on GARM. It provides a normalized vocabulary for representing the skeleton of an architecture, its structural view, and establishes a framework for capturing architectural constraints. It also allows extensions of the structural view with auxiliary information, such as behavior or quality specifications. In this respect, ASPECT provides facilities for establishing relationships among different specifications and gluing them together within an overall architectural description. This design allows flexibility and adaptability of the methodology to the specifics of a domain or a family of systems. ASPECT supports the representation of reference architectures as well as individual system architectures. The practical applicability of this method has been tested through a case study in an industrial setting.

The approach to architecture engineering and representation, presented in this dissertation, is pragmatic and oriented towards software practitioners. GARM-ASPECT, as well as the taxonomy of architectures are of use to architects, system planners and system engineers. Beyond these practical contributions, this thesis also creates a more solid basis for exploring the applicability of architectural abstractions, the practicality of representation approaches, and the changes required to the development process in order to achieve the benefits from an architecture-driven software technology.

DOMAIN ARCHITECTURE: A DESIGN FRAMEWORK FOR SYSTEM DEVELOPMENT AND INTEGRATION

by
Vassilka D. Kirova

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

January 1999

APPROVAL PAGE
(1 of 2)

DOMAIN ARCHITECTURE: A DESIGN FRAMEWORK FOR SYSTEM DEVELOPMENT AND INTEGRATION

Vassilka D. Kirova

Dr. Wilhelm Rossak, Dissertation Advisor — Date
Professor of Computer Science, FSU, Jena, Germany

Dr. Alexander Stoyen, Dissertation Advisor — Date
Associate Professor of Computer and Information Science, NJIT

Dr. Franz Kurfess, Committee Member — Date
Assistant Professor of Computer and Information Science, NJIT

Dr. Harold Lawson, Committee Member — Date
Lawson Förlag Konsult AB, Sweden

APPROVAL PAGE
(2 of 2)

DOMAIN ARCHITECTURE: A DESIGN FRAMEWORK FOR SYSTEM DEVELOPMENT AND INTEGRATION

Vassilka D. Kirova

Dr. Thomas Marlowe, Committee Member — Date
Professor of Mathematics and Computer Science, Seton Hall University

---

Dr. Peter Ng, Committee Member — Date
Professor and Chair of Computer Science, University of Nebraska - Omaha

---

Dr. Gary Thomas, Committee Member — Date
Professor of Electrical Engineering, NJIT

BIOGRAPHICAL SKETCH

Author: Vassilka D. Kirova

Degree: Doctor of Philosophy

Date: January 1999

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, 1999
- Master of Science in Computer Science and Engineering and Bachelor of Science in Computer Science and Engineering, Electromechanical University of St. Petersburg, Russia, 1980

Major: Computer Science

Publications:

Papers

Kirova V., Kradjel H.: "DirSA Case Study: An Introduction to Software Architecture Technology", Bell Labs Technical Journal, Vol.3, No. 3, September 1998, pp. 125-139.

Kirova V., Kradjel H., Rossak W., Marlowe, T.: "Engineering and Representation of Software Architectures: The DirSA Case Study," In Proceedings of SDPS Conference on Integrated Design and Process Technology, Vol. 4, Berlin, Germany, July 1998, pp. 55-62.

Kirova V., Rossak W.: "ASPECT: The Generic Architecture Description Language and its Customization Facilities," In Proceedings of IEEE Conference and Workshop on Engineering of Computer Based Systems, Jerusalem, Israel, April 1998, pp. 192-199.

Kirova V., Jololian L., Lawson H., Zemel T.: "A Generic Model for Software Architectures," IEEE Software, Vol. 14, No. 4, July/August 1997, IEEE Computer Society Press, Los Alamitos CA, pp. 84-92.

Kirova V., Rossak W., Howard Kradjel H., Sokoler D.: "Software Architecture Engineering: Where Research Meets Practice," Lucent Technologies Bell Laboratories Software Symposium, October 1996.

Kirova V., Rossak W., Marlowe T.: "Enterprise Software Architectures: A Case for Development of Enterprise-Wide Information Systems," In Proceedings of AIS Americas Conference on Information Systems, Phoenix, Arizona, August 1996, pp. 784-789.

Kirova V., Rossak W.: "Elements of Software Architectures - The GenSIF/GARM Model," INFORMATICA, The International Journal of Computing and Informatics, SSI, Vol.20, April 1996, pp. 159-172.

Kirova V., Rossak W.: "ASPECT - An Architecture SPECification Technique: A Report on Work in Progress," In Proceedings of IEEE Symposium and Workshop on Engineering of Computer Based Systems, Friedrichshafen, Germany, March 1996, pp. 220-227.

Kirova V., Rossak W.: "Representing Architectural Designs: A Central Issue in the Development of Complex Systems," In Proceedings of IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), Ft. Lauderdale, FL, USA, November 1995, pp. 80-87.

Kirova V., Rossak W., Lawson H.: "Software Architectures: An Analysis of Characteristics, Structure and Application," In Proceedings of IEEE Workshop on Architectures for Software Systems, International Conference on Software Engineering, ICSE-17, Seattle WA, USA, April 1995, pp. 166-174.

Lawson H., Kirova V., Rossak W.: "A refinement of the ECBS Architecture Constituent," In Proceedings of IEEE Symposium and Workshop on Engineering of Computer Based Systems, March 1995, pp. 95-102.

Kirova V., Rossak W., and Jololian L: "Software Architectures for Mega-System Development - Basic Concepts and Possible Specification," In Proceedings of IEEE International Conference on Systems Integration, Sao Paulo City, Brasil, August 1994, pp.38-45.

Rossak W., Zemel T., Kirova V. and Jololian L.: "A Two-Level Process Model for Integrated System Development," In Proceedings of IEEE Symposium and Workshop on Systems Engineering of Computer Based Systems, Stockholm, Sweden, May 1994, pp. 90-96.

Rossak W., Kirova V.: "A Development Process for Systems-of-Systems," In Proceedings of ASME Computers in Engineering Symposium, ETCE'95-PD-Vol. 67, Houston TX, USA, January 1995, pp. 195-198.

Kirova V., Rossak W.: "Some Thoughts on Architecture Engineering," in Proceedings of ASME Computers in Engineering Symposium, ETCE'94 - PD-Vol. 59, New Orleans LA, USA, January 1994, pp. 59-68.

Posters:

Kirova V., Kradjel H., Choudhary, R.: "Leveraging architecture and Process to Achieve Software Asset Reuse," In Proceedings of IEEE Conference and Workshop on Engineering of Computer Based Systems, Jerusalem, Israel, April 1998, pp. 232-235.

Tutorials

Rossak W., Kirova V, Jololian L.: "Systems Integration," Invited Presentation and Seminar for INTAN (The National Academy of Civil Servants) - System Analysts Assoc., Kuala Lumpur, Malaysia, December 1995.

Rossak W., Kirova V.: "Integrated Systems Development: An Architecture Oriented Approach," Invited Presentation and Seminar for Philips Research, Eindhoven, The Netherlands, June 1995.

Rossak W., Kirova V., John J., and John S.: "Integrated System Development - Development of Integrated Systems; The GenSIF/MegSDF Integration & Development Framework," Seminar for FMV - The Swedish Defense Material Administration, Stockholm, May 1994.

Technical Reports

Kirova V., Rossak W.: "Representing Architectural Designs," Technical Report, CIS-95-19, Dept. of Computer and Information Science, NJIT, 1995.

Rossak W., Kirova V.: "A Generic Model for the Use and Specification of Software Architectures", Technical Report, CIS-95-07, Dept. of Computer and Information Science, NJIT, 1995.

To Nikolai and Chris

ACKNOWLEDGMENT

Over the last six years spent at NJIT I have learned much and gained much from the people around me, and I know that I will never be able to truly express my appreciation. I can only say: Thank you!

I am truly indebted to my advisors and mentors without the guidance and support of whom this work would not have come to fruition. I cannot overstate the debt I owe to my advisor Willi Rossak. He has taught me about research, about teaching, writing, and speaking; he has taught me about software engineering, about architecture, and about being professional. He has supported me from my first arrival at NJIT to the final completion of my dissertation. He has provided guidance, friendship and moral support when he was here in Newark and after he moved to Germany. He has truly gone beyond the call of duty. I also thank and deeply acknowledge my advisor Alex Stoyen without whom this dissertation may have come more quickly but would have been written with less wisdom. To my good fortune I found a great mentor in Tom Marlowe whose human touch, creative nature, and integrity towards research helped make my dissertation the best that I can make it. His feedback as well as his example have enriched my experience as well as my work.

I would like to express my sincere gratitude to the rest of my dissertation committee members: to Professor Peter A. Ng for his support and encouragement through all steps of my studies; to Franz Kurfess for his patient consideration and professional counsel; to Bud Lawson for his technical insights; and to Professor Gary Thomas for his valuable comments provided on short notice.

I would like to extend my thank you to the NJIT and especially CIS communities for the caring and friendly environment. A special thanks goes to Fadi Deek, Professor McHugh, Mike Tress, Leon Jololian, Carole Poth and all other faculty and staff members of the CIS department for the help and support that have made

my stay at NJIT an enriching and pleasant one. I would like also to thank my fellow graduate students and specifically the past and present members of Software Engineering and Dependable Real-time Systems Laboratories for their timely help and constructive feedback.

I would like to acknowledge gratefully the support of Lucent Technologies Cross-Product Architecture Department and to thank all my Lucent colleagues for creating an excellent environment for research, development and application of new ideas. I am deeply indebted to Howard Kradjel for his insightful and constructive feedback which has helped to improve this thesis significantly. I also extend my appreciation to Nick Felmlee, Joe Kuhn, David Sokoler, Don Stewart, Saswata Bhattacharya, John Yeager, Rahim Choudhary, Nancy Peterson, Pete Schramm, Harvey Waxman and all other former and current members of the department for their valuable time, insightful comments, help and support.

Finally, my deep thank you to my family and friends, and especially to my husband Nikolai and my son Chris whose support has kept me going and who are what makes it all worthwhile.

Thank you all!

TABLE OF CONTENTS

Chapter **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Enterprise-Wide Software Solutions: Challenges and Means

An important problem facing software developers is the increasing size and complexity of software systems in general and business systems in particular.

Business imperatives, including flexibility, productivity, quality, time-to-market, and ability to adapt to changes, foster demands for effective use of software resources, for sophisticated, flexible and responsive applications, and for enterprise-wide integrated computational facilities. Integrated business systems are expected to solve complex problems and to provide global, secure, open and distributed computational environment to a business.

As a consequence of these increased business demands, and based on rapid technological advances, business software systems are evolving toward comprehensive solutions – *large-scale, flexible, coherent, enterprise-wide integrated software systems.* Today, a large business system is built by a team of more than 150 people over the time of more than a year, [23]. Its major components are developed separately or purchased off the shelf and deployed at different geographical locations. Yet. these individually developed or acquired components must work smootly together.

The rapidly growing complexity of software systems has made the process of system development, integration and evolution support highly demanding, and has presented new challenges to software technology. Over the last decade, considerable research, aimed at new development paradigms, methods, and tools, has addressed these challenges. Related research can be found under subject such as megaprogramming [20, 156]; module interconnection languages [113]; module interface specification and analysis [106]; domain modeling [111, 122]; domain-specific software architecture [94, 43]; architecture modeling, description and configuration [108, 52, 158]; architectural environments [104, 1, 85, 84, 15]; and software process [159, 114]. As a

result of joint efforts of practitioners and researchers, three key directions in software development have emerged that are of particular interest to the work presented in this dissertation: *architecture-driven software development, domain-specific software development, and component-based "plug-and-play" software development.* All three of these approaches are oriented toward development and integration of systems from large scale components and leverage *architectures* – blueprints describing system organization – to deal with the complexity of software systems and the process of their construction.

1.2 Complex Systems and Software Architectures

It is impossible to provide a single definition of a complex system, although there are generally accepted common characteristics of these systems which include: heterogeneous functional and extra-functional (non-functional) requirements; large size; multiple types of components, interaction models, and patterns of organization; heterogeneous technical environments; long life cycles and evolutionary changes; large amount of persistent data; diversity of concepts, knowledge, and traditions; more than one group of developers; more than one customer, and a large and heterogeneous group of users, [105, 156, 98, 92].

As the complexity of software systems increases, the overall system organization – *its architecture* – becomes a critical aspect of system design, [46]. The essence of an architectural design is in providing a view of a software system as a composition of large scale interconnected components, [3, 53, 77]. Software architecture raises the level of abstraction at which developers can reason about the system and its properties: an architecture provides a model of the system that suppresses implementation detail and thus allows the architects to concentrate on the analysis and decisions that are most critical to the system organization.

Typical architectural concerns include: decomposition of the overall system into types of components; types of allowed interactions and interaction models; overall system control and global synchronization; types of autonomy; global data access, management, and representation strategies; performance; system evolution; assignment of functionality to design elements; analysis and selection among design alternatives, [137, 133]. The design decisions made when addressing these concerns are captured in a separate artifact: a *software architecture.*

In the presence of increased complexity, the scope and role of software architectures have changed significantly. From being an informal, *ad hoc* applied tool for representing the organization of individual software systems, architectures today have also become a means used to capture generalized architectural solutions for classes of software systems.

These generalized architectures, referred to as architectural styles, product-line architectures, or domain architectures, address the design issues at higher levels of abstraction, [20, 108]. They provide verified, reusable architectural solutions and serve as *references* in designing multiple specific software systems from the classes being addressed by the respective architectures.

Consequently, these architectures hold a promise for a simpler design process, improved system quality, and reduced cost. To meet these expectations, however, they have to be supported by additional activities in the development process as well as by methodologies and tools, which are yet to become available to the practicing architects.

1.3 Business Domains and Systems Integration

We introduce the concept of a *business domain* as an internally coherent, relatively self-contained space of human knowledge and action with well-defined functions, operations, services, and software engineering traditions, [122, 75]. The concept of a

domain is fundamental to many areas of research. Typically, a domain is defined by consensus, and its essence is the shared understanding of some community, [60]. The business domain concept, as used in this dissertation, denotes knowledge about a type of enterprise; banking, avionics, university, insurance, and telecommunications are all examples of business domains.

Another well-known concept in software engineering is the application domain, identified by the functionality of a class or family of systems, for example, the domain of command and control systems or the domain of messaging applications. This application domain concept is fundamental to the work on Domain-Specific Software Architectures (DSSA), [95, 150]. A discussion on application domains, (application) domain-specific software architectures, and their role as frameworks for software reuse can be found in Chapter 4.

As the main objective of our research is the development of methods and tools to support the construction of integrated business applications, in this thesis, we focus on business domains in particular[1].

Business domains are studied and modeled from different perspectives. For instance, the balanced "7-aspect enterprise model", [61], examines three subsystems (cultural, social and technical) and identifies the following important aspects: identity, strategy, structure, people and groups, functions and agencies, processes (primary process, responsible for creating the added value for the customer, supporting processes, managerial processes) and technical means.

From a software engineering perspective a business domain defines an *overall problem space* for the development of software systems, designed to support the domain's operations. The complexity of this problem space results in a number of systems, such as information services, data warehousing, workflow management, office systems, process automation, image processing, and multimedia applications.

[1]The GARM-ASPECT method, presented in this thesis, however, is general and can be applied in a variety of domains.

Collections of such systems support the daily operations of any enterprise. The majority of these systems are required to interoperate, to share services and information, and to support extended functionality, in order to meet the expectations of software users. This presents requirements for *system integration* and creates incentives for development of open systems and composable components, [23].

A distinction has to be made between two models for system integration: "post-facto" integration [110], and the construction of systems from large scale components, being developed for integration. These large scale components can be complete systems, [156], constituting flexible "systems of systems," [41, 121], when integrated.

"Post-facto" integration refers to the practice of integrating already existing software systems developed without any coordination, [110]. Typically, it requires extensive additional development effort to glue the components together. In addition, aspects, such as performance, security, and extendibility, are difficult to handle in this approach. In general, "post-facto" integration results in local and fragile solutions, for the following reasons: (1) the constituent systems have been developed as standalone solutions to particular problems in the domain; (2) they have been developed in isolation and in an uncoordinated manner; and (3) they have been based on "closed" software architectures, designed for simpler needs and without any concern for future integration.

Many of the problems encountered with "post-facto" integration are architectural problems, and could be successfully resolved and prevented if addressed early in the development process, during system planning and architecture engineering; if common and open architectures as well as infrastructures are adopted for the domain; and if appropriate planning and coordination are used throughout the process of systems construction, [124].

Assembling systems from "ready for integration" components, or the "LEGO block" style of software construction, is an attractive alternative to the post-facto

integration. Such an approach to software development and integration should result in systems of higher quality and lower cost. It further promises more evolvable and flexible software solutions that could easily be adapted to the business needs of a specific enterprise. To enable this flexible style of integration, new technologies and architectural frameworks, as well as process changes are required, [156, 23, 159, 121].

Central to such a flexible style of system development and integration is an architectural framework which prescribes the system organization and allowed component interactions. The components developed in conformance with such a framework can be composed and work together without additional integration efforts. In a similar way, an architectural framework can be defined and applied to support the system development and integration in a business domain. We call such a framework a *domain architecture.*

1.3.1 Domain Architecture: A Framework for System Construction

A domain architecture[2] is a software architecture which prescribes the overall organization of software systems in a business domain. It is a framework of domain-specific architectural solutions, constraints, and rationale that establishes a common approach to the design and interconnection of software systems in a domain. A domain architecture constitutes a point of reference and conformity in the development process. It prescribes, guides, and constrains the development and integration efforts in a business domain and in this way supports the independent development of composable systems that can work together predictably.

The technical objectives of defining and using domain architectures, therefore, are to ensure that a collection of independently developed, acquired, or already existing components and systems in a domain (all being developed in compliance

[2]The domain architecture concept used in this document differs from the domain-specific software architecture (DSSA) concept introduced by ARPA, [95, 150]. See Chapter 4 for details.

with the architecture) can interoperate and share information. Consequently, when "plugged" together these systems will form a coherent "system of systems" whose overall functionality will provide a flexible software solution for an enterprise.

A relation can be made between the development of ready-for-integration software components and system in a business domain and the development of open systems, in general. One of the key notions in the context of open systems development is the concept of conformance. By conforming[3] to a standard, a system becomes open to all other systems obeying the same standard, where openness implies interoperability, portability, and scalability.

The same conformance principle applies to the development of software systems in a specific business domain, where a heterogeneous set of components and systems have to be developed in a way that guarantees their interoperability. The domain architecture, as introduced above, serves the purpose of a "domain standard" and becomes a tool for the development of open, domain-specific software systems.

The production of interoperable plug-and-play software components and open systems is the basic argument in support of a domain architecture-based development. The unified approach to the design and interconnection of software systems in a domain, established by an architecture, creates a foundation for component-based-development, promotes reuse and contributes to lower cost development efforts, [20]. Such an architecture can also support planning and evolution of software systems in the domain. The common design basis, further, allows for development of domain-specific tools and environments.

A domain architecture, however, can meet the above expectations only if it is properly defined, precisely described, and commonly applied. This requires an understanding of the domain architecture concept, its scope and its role in the devel-

[3]The conformance has to be precisely defined, see [58].

opment process. It also makes the rigorous methods for architecture engineering and representation indispensable.

The current practice in describing reference architectures is to use informal diagrams and volumes of textual descriptions, where a precise representation of architectural decisions is achieved only if an infrastructure is implemented to support the architecture. Even then the issues of applications' organization and their semantics are typically left unspecified, a major problem when integrating business applications. The right representation form for an architectural description, of course, depends on its purpose and users; therefore, flexibility and the right balance between formal and informal descriptions seem to be the right approach.

Despite the research efforts mentioned earlier, the research community has not yet offered software developers and integrators practical methods and tools to support architecture-based software development, integration and evolution. The promise of architecture-driven software technology has not yet been achieved nor proven unachievable, nor even properly tested. All these have been a strong motivation for the research presented in this thesis.

1.4 Summary of Contributions

Beyond the overall contributions of demonstrating the practicality of a systematic, rigorous approach to architecture engineering, representation and application, the contributions of this thesis fall into three main categories: taxonomy of software architectures; domain architecture engineering; and methods for architecture engineering and representation – architectural models and languages. In each category the contributions are as follows:

- Taxonomy of Software Architectures

  We have defined a two-dimensional classification model, where software architectures are categorized, based on the following criteria:

1. According to the level of abstraction at which software architectures address the issues of system organization we distinguish between reference architectures (architectures for classes of software systems) and particular system architectures (denoting the architecture of an individual system). The two types have very different scope, role in the development process, and evolution paths.

2. According to the system aspects being addressed by an architecture a distinction is made between conceptual architectures (addressing the extra-functional, property-specific aspects of software systems) and application architectures (addressing the functional, problem-specific system aspects). This distinction is a basis for large scale reuse, as we discuss in Chapter 3.

This new two-dimensional classification scheme creates a basis for *taxonomy* of software architectures. It helps in understanding the scope and roles of different architectural efforts in general as well as in categorizing and managing those specific to an organization or enterprise.

- Domain Architecture Engineering

  We provide a detailed characterization of (business) domain architectures, including the aspects of their engineering and the scope of their applicability.

  In contrast to previous efforts, focused on family architectures for specific application domains, on general integration architectures, or on mega-system architectures, we outline a concept of a conventional business domain architecture in conformance to which a heterogeneous set of business systems and software components can be developed and integrated into a coherent whole. We define the purpose and the roles of domain architectures as means to the syntactical an semantic composabilty of a business domain applications.

We also outline a domain architecture engineering process and specifically the process of populating the conceptual architecture models with domain elements and appropriate interactions. To support the process of an architecture definition and application we have developed a method for architecture engineering and representation.

- A Method for Architecture Engineering and Representation

  We have defined a Generic Architecture Reference Model, GARM, which establishes a basis for describing reference architectures and domain architectures in particular. GARM provides a set of modeling concepts and guidelines for combining them into architectural descriptions. The distinguishing features of the model, implied by its focus on domain architectures, include: multiple views of components and connectors, a separate rule system to express constraints, and allowing capture of experience from applying an architecture as domain-specific patterns and guidelines.

  We have also developed an architecture specification technique, ASPECT, based on GARM. The ASPECT language provides templates for prescribing and/or describing the common structural aspect of software systems in a domain. It supports decomposition-aggregation of architectural elements as well as a refinement process. It allows external formal specifications and informal documents to be associated with the elements of structural specifications. A specification in ASPECT serves as a glue between different views of an overall architectural specification. This language design allows for integration with domain-specific development tools and for finding a right (for the purpose of the architecture) balance between the formal and informal elements of an architectural description. We have also developed a template-based prototype editor for storing, consistency checking, and

modifying of ASPECT specifications, which our experiences have shown to be highly desirable (see Chapter 7 and Appendix C). GARM and ASPECT together form a coherent method for domain architecture engineering and representation: GARM-ASPECT.

To evaluate the applicability of GARM-ASPECT we have tested the method in an industrial case study. The results from this case study have shown the practical value of the language for capturing the early architectural decisions about systems organization and the importance of a systematic, "model-based" approach to architecture engineering.

1.5 Thesis Organization

In the overview presented in this chapter, software architectures have been addressed in the context of two parallel lines of discussion: complex systems development and systems integration. These two lines are further explored in the next three chapters, adding more details and perspectives.

Chapter 2 points out a set of basic characteristics of large and complex software systems, identified from an architectural perspective. It underlines the important role of software architectures in the development of complex systems and justifies the need for a systematic approach to architecture engineering and application. Section 2.1 examines the general aspects of complexity while the focus in Section 2.2 is on the large-scale, domain-specific integrated systems of systems.

In Chapter 3 an architecture classification scheme is established. We provide our answers to questions like "What are the levels of abstraction employed in architectural thinking?" and "What system aspects are typically addressed by architectural designs?". In Section 3.1 the general concept of software architecture is addressed. In Section 3.2 we concentrate on domain architecture: its definition, engineering context, requirements, and special properties.

Chapter 4 surveys research related to this dissertation.

The first four chapters set the stage for presenting our approach to architecture representation, addressed in Chapter 5. This chapter resembles a toolbox of architectural concepts, relationships, and syntactic elements for describing software architectures. It is a key chapter for those who want an introduction to GARM-ASPECT method. In Section 5.1 an architecture reference model (GARM) is defined, which forms the basis of our approach to architecture representation. We introduce the modeling abstractions provided by the model and explain their purpose and rational. In Section 5.2 we introduce the ASPECT architecture description language. We explain the language objectives and its constructs. In Chapter 6 we focus on the ASPECT refinement process.

Chapter 7 demonstrates the value of GARM-ASPECT method through an industry-based case study.

In Chapter 8, we summarize the results of this thesis and discuss our plans for future work.

Appendix A provides maping between the concepts of GARM and the constructs of ASPECT. Appendix B includes the abstract grammar of ASPECT. Appendix C provides an overview of the tool support available for GARM-ASPECT method. Finally, Appendix D is a glossary.

CHAPTER 2

LARGE AND COMPLEX SOFTWARE SYSTEMS

Complexity is not a goal,
it is the inevitable reality
of today's system developer.
The author

In addressing the subject of system complexity, we are guided by our architecture-focused point of view. Therefore, the issues identified at the level of overall system design constitute our basic concerns while the complexity of the computational problems or algorithmic solutions are outside of the scope of our interest.

2.1 What is a Complex System?

According to general system theory, an entity called "system" is a "complex organization of elements or parts in interaction," [16]. In dealing with systems, three different *aspects of complexity* are typically addressed:

1. *"largeness" in numbers* – according to the number of elements,
2. *heterogeneity of concepts and procedures* – according to the types of elements, and
3. *complex organization* – according to the relationships among the elements.

These aspects of complexity apply to the system itself as well as to the process of its development.

2.1.1 Largeness

2.1.1.1 Largeness in Space: We distinguish between two perspectives of the "largeness in space" aspect: *quantity and distribution.*

Quantity is typically defined in terms of lines of code and number of components, or so-called largeness of size, [156]. Software components of a complex system are also commonly distributed over a network of machines and operate under the control of different authorities, resulting in additional factors of largeness.

From a development point of view, we further identify additional aspects that require special attention – a large requirements space, distribution of development efforts, and large (and heterogeneous) group of users.

The traditional "programming in the large" approach focuses mostly on solving problems related to quantity. However, distribution of execution, control, and development, as well as the additional development aspects mentioned above, raise new questions. To address these problems new levels of coordination in the development process as well as new elements of standardization in the solutions have been suggested, [124, 156].

2.1.1.2 Largeness in Time: We consider the following two aspects of "largeness in time" as sources of complexity: *long life cycles* (and evolutionary changes during these long life cycles as a consequence of inevitably changing requirements, [99]) and *persistence of information* (e.g., data surviving the execution of programs, [156]).

In fact, the business interest in protecting the long-term investments in computing technology justifies an evolutionary model. Evolutionary development, however, raises questions related to legacy code, closed or outdated application architectures, and incompatible environments. Thus, the evolutionary approach to systems development substantiates the needs for re-engineering and open solutions.

Persistent corporate data is another aspect of the largeness in time criterion. It raises a set of architectural issues related to data availability, integrity, and security, which need to be addressed during the development and maintenance of complex systems, [4, 82].

Evidently both "largeness in space" and "largeness in time" create difficulties in supporting behavioral, functional and data consistency between the diversity of constituent components of an overall complex system, as well as across different versions of components created over time. Thus, a need arises for new development methodologies that can deliver scalable and interoperable systems. Among others, the "largeness problem" can be addressed by introducing additional artifacts, such as domain models and reference architectures, which serve as special means, in the development process, that manage change, guarantee consistency, and provide for controlled evolution of systems. Consider as an example the domain-oriented methodologies which strive to provide comprehensive domain models, [149, 20, 122], together with rigorously-specified domain-specific software architectures, [130, 95, 17, 76]. Further research, however, is required to develop suitable techniques that can support such methodologies and increase their applicability.

2.1.2 Heterogeneity

Large and complex systems are typically characterized by a wide diversity in customers and users; in knowledge, concepts, and traditions; in development paradigms and programming languages; in operating systems and equipment. They are also characterized by an unusual variety of provided services, applications (where it is desired to integrate otherwise separate applications to perform a single task) and authorities. All these are different forms of a source of system complexity referred to as *heterogeneity.*

From the perspective of a software system designer, three major aspects of heterogeneity dominate: *heterogeneous requirements, heterogeneous development paradigms, and heterogeneous environments.*

2.1.2.1 Heterogeneous Functional and Extra-functional Requirements:

Typically, large and complex systems are characterized by a significant number of different and possibly conflicting functional as well as extra-functional requirements. This reflects the fact that such systems usually support broad application or business domains and have more than one customer as well as a large and heterogeneous group of users, [98]. The internal functional and organizational diversity of the problem space leads, on one side, to a high number of functional requirements and, on the other side, to a variety of extra-functional requirements, especially for openness (interoperability, portability, scalability), but also for dependability (reliability, availability, security, safety), performance, timeliness, fault-tolerance, cost, etc. Consider, for example, the requirements space of an integrated hospital information system. Such a system comprises multiple applications, servicing diverse departments such as medical and research units, ambulatory care, laboratory modules, pharmacy, medical records, and insurance clearing. Beyond providing a large set of services these applications have to meet diverse extra-functional requirements for security, reliability, availability and semantic consistency.

The necessity of reconciling, integrating, evaluating and tracing such diverse requirements makes the engineering process difficult and costly. It also increases the importance of formal methods and systematic approaches, based on reusable artifacts such as domain models, reference architectures and objective and cost function models.

2.1.2.2 Heterogeneous Development Paradigms: It is to be expected that any well-developed part of a complex (software) system has an internally consistent ontology reflecting a suitable conceptual framework and problem-solving paradigm. The different constituent parts of the overall system, however, are typically developed to solve different problems and satisfy different functional and extra-functional requirements. They are also often developed at different times and by designers with different "cultural" backgrounds and experience. As a result, a heterogeneous set of development paradigms must be expected to co-exist in a large and complex system.

This heterogeneity is inevitable and "design autonomy" is tolerated, provided that the constituent components meet their obligations as parts of a coherent overall system. A recognized way to make this possible is to exploit information hiding and hierarchical design, [105], in the context of a global conceptual framework for design standardization and coordination of all development efforts contributing to the overall system. Software architectures can serve successfully in this role, as they (if properly defined) designate high levels of abstraction, codification, standardization and style, [108] – properties that make them indispensable as frames of reference and conformity, guiding and constraining the development efforts and thus contributing to a consistent overall design and implementation.

2.1.2.3 Heterogeneous Technical Environments: It is also common that different constituent parts of a large and complex system operate in different execution environments. From the user perspective, the incompatibilities between varying computing environments is one of the basic obstacles in achieving full exploitation of the existing software resources, integration of software applications, and sharing of services and information. From the system developer's point of view, the problem is even more aggravated due to the heterogeneity of the development environments.

Thus, requirements are imposed for unification and standardization of operational as well as development environments. The recognized solution is a so-called open infrastructure, [11], also referred to as middleware software [15, 152], or software bus [115, 24] (with some differences in the emphasis put on runtime services or development support). Two directions in the development of such an infrastructure are noticeable: (1) an infrastructure defined as a specific set of tools for systems development and operation, standardized in a particular domain and supporting a given domain architecture, for example, BaseWorkX [36]; and (2) an infrastructure developed as a set of tools that support a general class of software systems, based on widely accepted standards and architecture reference models – ODP-RM [1], for example, is supported by ANSAware [11] and DCE [84], while CORBA is supported by CORBA-compliant ORBs [104].

2.1.3 Complex System Organization

The third aspect of complexity, we have focused on, examines the relations between the elements constituting the system and is defined in terms of *types of interactions and patterns of system organization.*

Complex and heterogeneous patterns of interaction as well as ad-hoc created "irregular" system structures significantly increase system complexity and complicate the development and maintenance processes. But while the first two aspects, largeness and heterogeneity, are inevitable and extended development support is needed so they can be properly accommodated, the complexity of system organization can be reasonably reduced by a proper system development strategy.

"System organization" is the aspect of complexity where architectural thinking can contribute significantly. An architecture-based approach to system construction addresses this system aspect systematically and early in the development process, and helps to avoid unnecessarily complex organizations. It stimulates reuse of estab-

lished patterns for system organization and supports the creation of a coherent model for organization and behavior of the overall system. Thus, it establishes a consistent high-level design framework – a software architecture – that, if used as point of conformance for all development effort contributing to an overall system, will support the production of coherent software systems. Further, just by being a system description at a high level of abstraction, an architectural design becomes a means in dealing with complexity. It makes the overall system "intellectually tractable" [48], hence, simplifies and eases its development and maintenance.

2.2 Domain-Specific Integrated Software Systems

In the research presented in this thesis we have focused on a specific type of complex system -- large-scale, integrated software applications which support the operations in a given business domain. Such *a domain-specific integrated system* is seen as a whole composed of independently developed, autonomously operating, and independently maintained software components (e.g., systems, subsystems, modules) which cooperate in constituting *a "system of systems" (S^2)*, [41].

It is easy to see that such domain-specific integrated systems are subject to most of the aspects of system complexity addressed in the previous section. They, for example, will be inherently large in space and time as they integrate a number of independent components (systems) and represent a substantial part of the overall investment in software technology for a business. They are prone to a large and heterogeneous requirements space, as they support a diverse set of operations in the domain. A heterogeneous set of development paradigms is also common for them due to the variety in the nature of the problems being solved by the different constituents of the overall system. If developed ad-hoc or as a result of post-facto integration, it is also reasonable to expect that the overall integrated system will be characterized by complex organization and multiple interaction models. The goal of "proper" system

integration, however, is to produce *coherent and open systems of systems* so that cooperation, scalability, and system evolution are readily supported.

As an example consider the Insurance domain and the following four domain-specific applications, supporting a broad range of operations in this domain: a front-end customer communication system, a statistical rating application, an insurance company "headquarters system," and an agent-customer accounting system, [69]. Integrating these applications (in a proper way) should provide an Insurance enterprise with a business-wide information system where data are freely exchanged and information processing capabilities are commonly reused among the applications, placing an extended and consistently presented set of functionality to the disposal of the end-users.

To make such open and coherent systems of systems a reality for the business world, new thinking, new technology, new software methodologies, and definitely new investments of resources are required that can support the development of "open applications[1]."

The goal is to replace the "old, closed applications" with a community of inter-operable components distributed across the enterprise. In fact, the notion of an application changes as the typical monolithic solution is being replaced by a group of externally-accessible components, which are functionally related, usually developed as a group, and delivered as a package[2]. Thus, the application boundary becomes transparent and the basic primitive construct of the S^2 down-sizes to an atomic, autonomous software component, we call "building block". (In this context the issue of legacy systems requires special attention, as legacy code needs re-engineering,

[1]The interest in Distributed Objects Management, CORBA, [104], or integrated software agents technology, [56, 118], for example, makes it evident that this is a noticeable trend.

[2]Consider, for example, " object frameworks," [40].

"wrapping", or other transformations, in order to resemble proper building blocks, [37, 56, 99].)

Integration in such open system of systems is achieved by cooperative work between the constituent components – the building blocks. As part of the overall system, each building block provides "freely-accessible" service(s) (operations and information) to its environment (system users and other components) and may also consume service(s). Thus, any service available to the end-users is provided by a building block or is a result of service sharing (interaction or information sharing) between a set of building blocks (contributing to the service), referred to as a "cooperating group[3]". Any building block may be contributing simultaneously to several cooperating groups, exhibiting different or equal behavior. As a result, additional services are offered to the end-users, and an opportunity is provided for an open exchange of information and for effective use of the software resources available in the enterprise.

The S^2 concept itself does not imply any specific model of interaction or system structure. A system of systems, for example, can be successfully organized as a *federation.* A federation is typically defined as a system that maintains the autonomy of the individual heterogeneous component-systems but yet supports cooperation to exchange services and information without the supervision of a central and global entity. For any component to be a member of the federation it should be able to communicate with the other member-components as well as to exhibit certain expected external behavior, constrained by some general principles defined for the federation, [68, 138, 59, 63, 83, 11, 56].

A different approach would be to apply the organization paradigm established by "megaprogramming," [156]. In the context of megaprogramming, integration between S^2 constituents would be achieved by a megaprogram. This megaprogram

[3]The "cooperating group" concept, as used here, is similar to the "cooperating objects group" concept used in object-oriented analysis and design techniques.

will act as a global "user-component" which directs the use of services provided by the S^2 components, in this way explicitly implementing those services that are provided to the end-user as result of the integration and, thus, demonstrating the "added-by-integration value". (In the context of megaprogramming, the services provided by the system of systems components are to be specified in a common interface language, while the megaprogram is written in a "glue" megaprogramming language that allows flexible composition of megamodules, [156]. Further discussion on megaprogramming can be found in Chapter 4.)

The open S^2 model, addressed above, can be made feasible if the constituent components (systems) are developed in conformance to a common and standardized domain-specific design model of the overall system of systems – *a domain architecture.* Such an architecture defines common information models and prescribes the allowed patterns of system organization. It also specifies the interaction models, the types of constituent components, their behavior and provided services. In this way, a domain architecture can establish the needed common framework for development of composable systems, which, if developed in conformance to the architecture, can be integrated into an overall coherent system of systems without additional development efforts.

CHAPTER 3

SOFTWARE ARCHITECTURE

The purpose of this chapter is to establish a sound understanding of the software architecture concept, to characterize domain architectures, and to develop a basis for architecture taxonomy. The discussion is structured as follows:

- Section 3.1: *Software Architecture: An Overview* – provides an introduction to the issues of software architecture engineering, definition, and representation. A classification grid for architectures is presented. First, two basic types of architectures are identified – reference architectures and particular system architectures – in respect to the levels of abstraction at which they address the issues of system design. Then, a view orthogonal to this perspective is presented, separating the extra-functional from functional aspects of an architectural specification into conceptual and application models, respectively.

- Section 3.2: *The Domain Architecture Concept* – develops an understanding of domain architectures. It presents a domain architecture as a reference architecture tailored to the specific properties of a business domain, and examines its organization, scope, engineering, and role in the development process.

3.1 Software Architecture: An Overview

In this section software architectures are discussed from a conceptual point of view. A more historical perspective is provided in Chapter 4.

3.1.1 Software Architecture: Research and Practice

The problem of software architecture has long been a concern for those building and evolving large software systems. Typically, the software architecture is in place

before the requirements are passed on to the developers for implementation and it functions as the overall structure within which the requirements are to be met.

More recently, software architecture has become an important topic in software engineering research, [131]. This research ranges from defining architectural processes; to classifying architectures; to investigating how to describe and analyze software architecture; to codifying components, structures, styles and patterns as well as proposing novel ones, [107].

In overall, the goal of researchers and practitioners is to systematize the experience of architecting and designing software systems into a comprehensive, well-founded discipline: to codify architectural experience, so it can be successfully reused; to establish a common conceptual basis for describing software architectures; to provide a formal semantics of architectural representations; to develop rigorous techniques as well as tools for architecture representation and analysis; and to clarify the implications of architecture-based development for software technology, [136, 120].

3.1.2 The Evolution of Architectural Thinking

The increased complexity of software systems has compelled system engineers to employ higher levels of abstraction in addressing the issues of system design. It has made the overall system organization an important aspect of system development and a *software architecture*, describing it, a separate artifact and a separate process entity, independent from the system design. Further, it has set new requirements for consistency and precision in representing architectural solutions. It has also made it necessary to approach architecture engineering in a systematic way, and has led to a rapid development of architecture technology.

A technology typically "... evolves from being a craft to an engineering discipline over time with the infusion of scientific theory and the need for broad

application," [18]. As a young discipline, early architectural efforts were characteristic of the *craft stage,* where an architecture is created from scratch, relying on the designer's personal experience, knowledge, and intuition. With the mass market involved, software architecture has now reached the economics-driven *commercial stage*, characterized by introduction of architectural standards, reusable reference architectures, and domain-specific software architectures. With more maturity and the presence of increased system complexity, one should also anticipate the software architecture field taking on the characteristics of a *professional engineering discipline:* underpinning theory, architecture support tools, standardized practices, licensed professionals, and a community of shared expertise.

The software architecture discipline is concerned with the principled study of the patterns of system organization, large-grained software components, their relationships and the models of interaction between them. It addresses the overall system properties at high levels of abstraction and from multiple perspectives such as structure, control, and data. Architectural designs address important system characteristics, including scalability, overall performance, security, and allocation of functionality to design elements, [108, 46, 120].

Prior to the emphasis on the architecture as a separate artifact in the software life-cycle[1], software architecture has been known as a description of the top-level design of an individual software system. Typically, this description has been created *ad hoc* and represented in terms of informal box-and-line diagrams, reflecting the system structure, and free-form textual descriptions clarifying the meaning of the diagrams and capturing some design rationale. Even being informal, such descriptions provide a critical first view of the solution and a starting point for determining whether a system can meet its essential requirements. They guide the developers in constructing the system and help management to organize the entire

[1]Indeed, there are those who still think that the consideration of architectures in the development process should be a part of the high level design of a system, [107].

project. Unfortunately, such architectural designs cannot be analyzed for consistency or completeness; often they are poorly interpreted throughout the lower levels of the development cycle; further, architectural constraints are difficult to preserve and enforce as the system evolves.

Recently, architectural descriptions are also used for codifying and reusing architectural knowledge. These software architectures, referred to as architectural styles, domain architectures, product line architectures, and architectural frameworks, are bodies of high level, often domain-specific design knowledge, references, and standards providing generalized solutions for classes of problems, [130, 3, 120].

These general architectures, denoted as reference architectures, have a broader scope as well as a new place and a new role in the development process. They prescribe the architectural design of software systems that belong to a given class, domain or family. They are planned to be widely reused, to serve for long periods of time, and to support a number of development and integration efforts. As a whole, they are expected to serve as bases for planning, development, reuse, acquisition, maintenance, and evolution.

The application of such architectures promotes a new approach to building software systems. This architecture-driven approach is more restrictive in respect to the freedom of software architects but at the same time, by the virtue of providing ready-to-use, verified solutions, it simplifies the process of designing complex systems, lowers the cost, and increases the quality of architectural designs.

In respect to the issues of engineering and representation of reference architectures, the research community is yet to offer practitioners practical and effective methods and tools.

3.1.3 Classifying Software Architectures

We have examined numerous architectures in order to build a basis for architecture classification. A brief discussion on several of them is provided in Chapter 4. Below, we present the results and the conclusions of the classification process.

3.1.3.1 Classifying Architectures – Levels of Abstraction and Scope: Architectural issues can be addressed at different levels of abstraction and architectural solutions can be captured in architectures with different scope and purpose (Fig. 3.1). Between them two basic types of architectures can be differentiated:

- *reference architecture* – an architecture of a class of software systems which serves as a prescription from which other architectures are derived. Examples include: CORBA [104], OSCA [4], a variety of domain-specific architectures (DSSA) [130], and "3-tier architecture" [128].

- *particular system architecture* – the architectural design of an individual software system which satisfies a specific set of functional and extra-functional requirements. Examples include the architecture of "MS Word"; the architectural design of "SIS NJIT" – the Student Information System of New Jersey Institute of Technology; and the CORBA-based architectural design of "VS," a surgery-support multi-media application, [91].

Reference Architectures:

Reference architectures constitute high-level design frameworks of concepts and analytical foundations (also notations) for normalized description of an arbitrary system that belongs to a certain class. They provide vocabularies of architectural design elements (for example, pipes and filters) from which system design models

<table>
<tr><th>Functional and extra func.
Level of abstraction</th><th>Conceptual Architecture
Property-specific Solution</th><th>Application Architecture
Problem-specific Solution</th></tr>
<tr><td rowspan="2">Style
Class of systems
1-2 aspects</td><td>Emphasis on: - Structure
types of components and
relations between them
- Interaction Model</td><td></td></tr>
<tr><td>Examples: Pipe-&-Filter
Client-Server
Event bus
Object oriented</td><td></td></tr>
<tr><td rowspan="2">Reference Architecture
Class of systems
All relevant aspects</td><td>Concerns: - Refinement and integration of styles
specialized component types and
interaction models -
e.g., invoc. scheme, data types
- Properties (esp. extra-functional)
- Common Services</td><td>Concerns:
Problem-Specific
Services</td></tr>
<tr><td>Examples: CORBA
OSCA
ANSA</td><td>Examples:
DSSA
(e.g. C^2 Architecture)</td></tr>
<tr><td rowspan="2">Particular System Architecture
One system</td><td colspan="2">Concerns:
Provides a high level design solution for a set of specific extra-functional and functional requirements.</td></tr>
<tr><td colspan="2">Examples:
CORBA-based architecture of a banking application
ANSA-based architecture of Astrophysics Integrated Services System</td></tr>
</table>

Figure 3.1 Taxonomy of Software Architectures

can be constructed, and define constraints (architectural rules) that restrict those elements and their static and dynamic compositions.

As an example, consider the ANSA architecture, [10] – a reference architecture for development of open distributed systems. ANSA defines a number of architectural elements: client, server, interface, operation, activity, termination, interface type, operation invocation. It also defines data type and invocation schemes. In addition, ANSA provides design solutions for properties like location transparency and fault tolerance.

A special case of a reference architecture are so-called *architectural styles*. Typically, they are defined as established patterns or idioms of system organization and include organizations such as pipes-&-filters, layered systems, and client-server,

[108, 53, 47, 3]. Although, we have chosen to show them separately in our classification grid (Fig. 5.1), there is no conceptual difference between a style description and a reference architecture. The separation is more of a pragmatic one and is based on the level of generality at which architectural issues are addressed – styles are at the root of the generalization hierarchy, a property that singles them out from the rest of the reference architectures.

Architectural styles emphasize only certain architectural aspects, typically, those fundamental to an architecture definition, structure and basic interaction principles. They address these aspects at a high level of abstraction, capturing only the important elements, while suppressing irrelevant implementation details and perspectives. A reference architecture can then provide a refinement of the basic concepts defined in a style, or may combine several styles.

Generally speaking, any reference architecture can be defined as a refinement of another reference architecture, addressing new aspects or providing refinements of specific elements. Therefore, a *de facto* continuum of architectural refinements can be observed. Consider, for example, the client-server style and the ISO Reference Model for Open Distributed Processing (ODP-RM architectural framework), [1]. The client-server style defines the basic types of components, namely, clients and servers, and constrains the relations between them, [7]. The client-server-based ODP-RM in addition defines a detailed set of design elements which can be used to model open distributed systems. The model also defines properties and rules such that if obeyed in architectural designs for an actual system, the system will be organized according to the client-server principles and will exhibit properties that will qualify it as an open distributed system. Subsequently, a "3-tier architecture" can be defined as an additional level of architectural refinement, derived from ODP-RM. This architecture reuses the architectural solutions of ODP-RM but further focuses on the organization of the applications in an enterprise-wide business software system and defines three

specific types of servers: "production server," "application server," and a "business logic server," [128].

The following are two types of reference architectures that deserve special attention: family architectures and architectural frameworks.

A *family architecture* is an architecture for a class of software systems with a common organization and functionality. This type of architecture is the basis for reuse in "software product lines" and as such is also called a *product line architecture.*

Architectural frameworks reflect the developers' perspective. They are consistent, evolving collections of architectural standards, specifications, rules, and agreements under the guidance and terms of which a collection of systems (typically, from several different families) and components are developed, integrated into flexible "customer offers," and maintained, [70].

Particular System Architectures:

When addressing the roles software architectures play in the development process, a clear distinction can be made between a general reference architecture and a particular system architecture, where a reference architecture is used as a "reference-tool" that supports the transformation of a specific set of functional and extra-functional requirements into an architectural design of the target system, referred to as a *particular system architecture.* This architecture is a design model of an individual system under development and describes system organization and behavior in terms of its specific components, their interactions, and the results of these interactions – a number of static and dynamic configurations, [53, 1, 75].

The goal of software architects, therefore, is to find a manageable collection of constructive architectural elements, to define the constraints on these elements, and to integrate them into a comprehensive and consistent framework (a reference architecture) in which the designs of a class of software systems can be described.

This reference architecture can then be re-used in defining architectural models of individual systems from the supported class, which in turn will specify how any one of these systems has to be organized and operate in order to meet its specific functional and extra-functional requirements.

In this way, the application of architectural styles and reference architectures creates a basis for reuse. It simplifies the design process and enhances productivity and product quality by providing verified, "routine solutions for certain classes of problems," [137, 48]. It also promotes reuse of underlying implementations, supports tool development, and finally reduces the associated cost, [79].

Styles, reference architectures and particular system architectures, with their characteristics, scope and purpose, mark one of the dimensions of our classification model, referred to as *level of abstraction*. The elements of the second dimension are discussed next.

3.1.3.2 Classifying Architectures – Functional and Extra-functional Properties: When classifying architectures, it is also important to examine how an architecture addresses the functional versus the extra-functional aspects of software systems (see the horizontal dimension of the classification grid, Fig. 3.1). As will be discussed later, this separation is much needed, especially when defining reference architectures, as it creates a basis for large scale reuse.

To this end two general aspects of an architecture reflect our basic concerns – architecture as a "property-specific" and as a "problem-specific" solution. We view an architecture as a "property-specific" solution when it captures the design decisions made in order to support certain engineering properties such as system organization, behavior, performance, security, and reliability. As a "problem-specific" solution, an architecture solves a particular set of (application-specific) customer problems, utilizing a certain property-specific system design model. As a result we distinguish

between two types of architectures, conceptual and application, as follows (see Fig. 3.1).

Conceptual Architectures:

A conceptual architecture is a high level design model of a class of systems with specific organization and behavior such as distributed systems and object-oriented systems. It defines a collection of element types and their allowed configurations. It also establishes a set of computational and engineering principles. Computational principles define the way in which computations progress in the system. They address the allowed types of interactions between the computational units (the components) as well as their organization. Engineering principles refer to the solutions provided in the architecture for a set of required system properties such as reliability, security, and location transparency. Conceptual architectures model software systems as "entities by themselves" which follow their own rules of "existence," namely, organization and behavior. Problem-specific functions, contributing to the solution of particular application problems, are transparent in such an architecture definition. The following are examples of conceptual architectures:

(1) OSCA, [4], and "3-tier architecture," [128] in which the main focus is on the organization of applications. Both architectures provide organizational frameworks for development and integration of business systems. They define types of components and constraints that guide the allocation of functions to them (see application architecture).

(2) CORBA, [104] – an object-oriented architecture that supports Distributed Object Management (DOM); and ODP–Reference Model, [1] – an architecture for Open Distributed Processing (ODP). Both architectures are frameworks for development of general types of systems – object-oriented and open distributed systems, respectively. Their main focus is on the standardization of component interactions

and on defining sets of common architectural services that support properties such as location transparency, reliability, and security. A natural way to implement the concepts of such an architecture is as *an infrastructure* – a development and execution environment, e.g., ANSAware [11], DCE [84], CORBA-ORBs [85]. Such an infrastructure, when used, "enforces" the architecture and makes the development of systems compliant to the architecture a better-supported process.

Application Architectures:

An application architecture is a domain- or function-specific refinement of a conceptual architecture, where problem-specific functions are explicitly defined and allocated to the design elements of the underlying conceptual architecture. The application architecture corresponds to the view of the system as a solution to the customer problems. Architectures such as the Domain-Specific Software Architecture for Navigation and Control, [5], or the architecture of US Navy Weapons Systems (AEGIS Weapon System (AWS)), addressed by ARPA's Domain Specific Software Architectures (DSSA) and Prototyping Technology Programs, [130, 51], are examples of this level of architecture definition.

Consider also the following example. Assume that a *client-server-based 3-tier architecture* serves as a conceptual basis for the development and integration of software applications that support the operations of an insurance company. According to this conceptual architecture, applications will be organized and interoperate as clients and servers. Further, the servers are categorized; for example, a "business logic server" is defined as a component type to which problem-specific functions will be allocated. The application architecture then will include descriptions of instances of this component type to which a service like "Calculate Quota", specific for the insurance business, will be assigned.

Discussion:

In general, a conceptual architecture is a "property-specific" solution while an application architecture is a "problem-specific" solution. The main task on the conceptual level is to identify and specify the involved architectural elements (for example, pipes and filters in a "pipe-&-filter" architecture) and the constraints on those elements (e.g., rules for their composition and operation). On the application architecture level the basic concerns are the allocation of problem-specific functions to the design elements, defined in the conceptual architecture, and the identification of architectural configurations required in order to solve customer problems. While there is some flexibility in the division, it is usually clear whether a given functional requirement belongs to the conceptual or application side of a system architectural model, for a particular domain. For instance, when developing business applications for an insurance domain the operating system functionality is common to all applications and belongs to the conceptual infrastructure model. However, if the target of development is an operating system, then its functions are part of the application model, while the system organization (e.g., a layered structure) constitutes the core of the conceptual model.

As we have seen earlier, the applications that support the operation of an insurance company can be organized as clients and servers. Alternatively, one may choose to organize these software applications as a set of batch programs acting on pre-stored sets of instances, in order to minimize search time, [91]. The architectural designs and implementations for these two cases will look fundamentally different, even though the insurance company didn't change at all and neither did the functionality provided by the applications. Evidently, the knowledge of how to organize a software system, captured in a conceptual architecture, is to a great extent independent from the specific functionality provided by the system and reflected in the application architecture. Therefore, a conceptual architecture can be applied

equally to support systems development and integration in an insurance company, a bank, a university or any other domain of application. Certainly, this separation of concerns creates foundations for large-scale design reuse. In order to achieve such reuse, however, the conceptual architecture has to be retained entirely separate from its application – the application architecture.

The conceptual and application architectures (or models) mark the second dimension of our classification model, which is defined by the functional and extra-functional system aspects addressed by software architectures.

In this way we have introduced a two-dimensional classification scheme (see Figure 3.1) which creates a basis for *taxonomy* of software architectures. It helps in understanding the scope and roles of different architectural efforts in general as well as in categorizing and managing the specification and application of those specific for an organization or enterprise.

3.2 The Domain Architecture Concept

In this section we define the domain architecture concept and examine the processes of its engineering and application.

3.2.1 Domain Architecture: An Introduction

As discussed in Chapters 1 and 2, the business practices of an enterprise are automated by numerous business applications which support business operations in areas like customer service, manufacturing, staffing, reporting, etc. These software applications collectively comprise the domain's "business systems portfolio", [18]. Some of these applications operationally "run" the business on a daily basis, others analyze and aid the progress, but all constantly evolve, and most of them are required to interoperate. The interoperating applications constitute integrated systems of systems. Years of experience in systems integration, maintenance, and evolution

support have revealed an array of problems and have made the corporate world and software developers recognize that:

- even careful crafting of software applications does not substitute for common design principles and standardization;
- poorly-defined or *ad hoc* created architectures are the underlying cause of many problems which restrict effective use of the computational facilities of an enterprise; and
- there is no architecture theory adequate to support engineering practitioners in developing or acquiring architectures, [13].

All these, in turn, have led to the following conclusions:

- the study of architectures cannot be further ignored;
- a systematic approach to architecture engineering is needed; and finally
- architecture standardization and strict compliance to the architecture are fundamental to the solution of the above problems.

Thus, a standardized architectural framework – *a domain architecture* – defining the gross organization of software resources in an enterprise, or more generally providing a model for organization and operation of software systems in a business domain, emerges as a promising means for addressing the above problems, [77, 76].

3.2.2 Domain Architecture: Definition

We define a domain architecture as a *domain-specific reference software architecture.* As with any reference architecture, a domain architecture constitutes a common, high-level design model of a class of software systems, while in particular addressing

the specific class of integrated systems of systems that meet the requirements and satisfy the constraints of a specific business domain. Thus, a domain architecture constitutes a reference framework under the terms of which the domain's collection of business applications is designed, implemented, integrated and maintained.

Being a reference architecture, a domain architecture description exhibits the typical division into a conceptual architecture (model) and an application architecture (model). As introduced in Section 3.1, a conceptual model prescribes the system's organization, behavior, and dependability. An application architecture, in turn, reflects the functional requirements in the domain; it defines an array of domain-specific services and prescribes the way they will be delivered in a target software system. Moreover, although, a reference architecture description in general may or may not address problem-specific aspects, and hence, may or may not contain an application model, for a domain architecture both conceptual and application models are fundamental.

Moving from the general concept of a reference architecture to the concept of a domain architecture incurs changes in the scope of the architecture and its constituent conceptual and application models. For the conceptual architecture, the transition mostly means an adaptation to the domain's flow of processing. For the application model, however, it is a substantial "change of status,", since in a domain architecture the problem-specific perspective starts to dominate.

As we have seen in Section 3.1, some conceptual architectures focus on software applications and their organization (e.g., "3-tier architecture"), while others, like CORBA and ODP-RM, put an accent on the model of interaction and thus provide an architectural model of a software infrastructure. A conceptual model of a domain architecture has to address both of these views but constrained by the requirements of a specific domain. It performs the following roles: (1) prescribes the system (of systems) organization – types of software components, types of configurations, global

control; (2) defines the model(s) of interaction between the components – invocations and data types; and (3) defines architectural services that support required properties such as transparency, transactions, reliability, and security, [1] (see Fig. 3.2). The conceptual architecture portion of a domain architecture guides the acquisition and customization of an infrastructure (such as CORBA, DCE and DCOM) that meets the needs of the domain. It also serves as a basis for the engineering of the application architecture portion.

As introduced in Section 3.1, application architectures capture problem-specific design solutions and therefore can be defined only in the context of specific problems. For example, in [133], M. Shaw examines several application architectures providing solutions to the well-known "cruise control problem." Similarly, but on a larger scale, the application architecture part of a domain architecture reflects the functional aspects of a domain and is defined only in the context of the particular domain. The application architecture constitutes a major portion of a domain architecture, which describes: (1) the business-specific services that are provided by the software applications in a domain, and (2) the configurations that have to be established among software components in order to meet the functional requirements of the domain. Application architectures support the implementation of software applications, from which domain-specific system of systems are built.

The view highlighted by a domain architecture is the domain specifics of the solution. This implies an engineering process that supports a mapping of the characteristics of a business domain into the corresponding architecture. A domain is characterized by well-defined business practices, established operations and patterns of information processing, which if properly addressed during this process and reflected within the architecture (as well as in the notations and tools developed to support this architecture) will lead to a reference software architecture tailored to the domain. The conceptual architecture part of such an architecture will better fit the

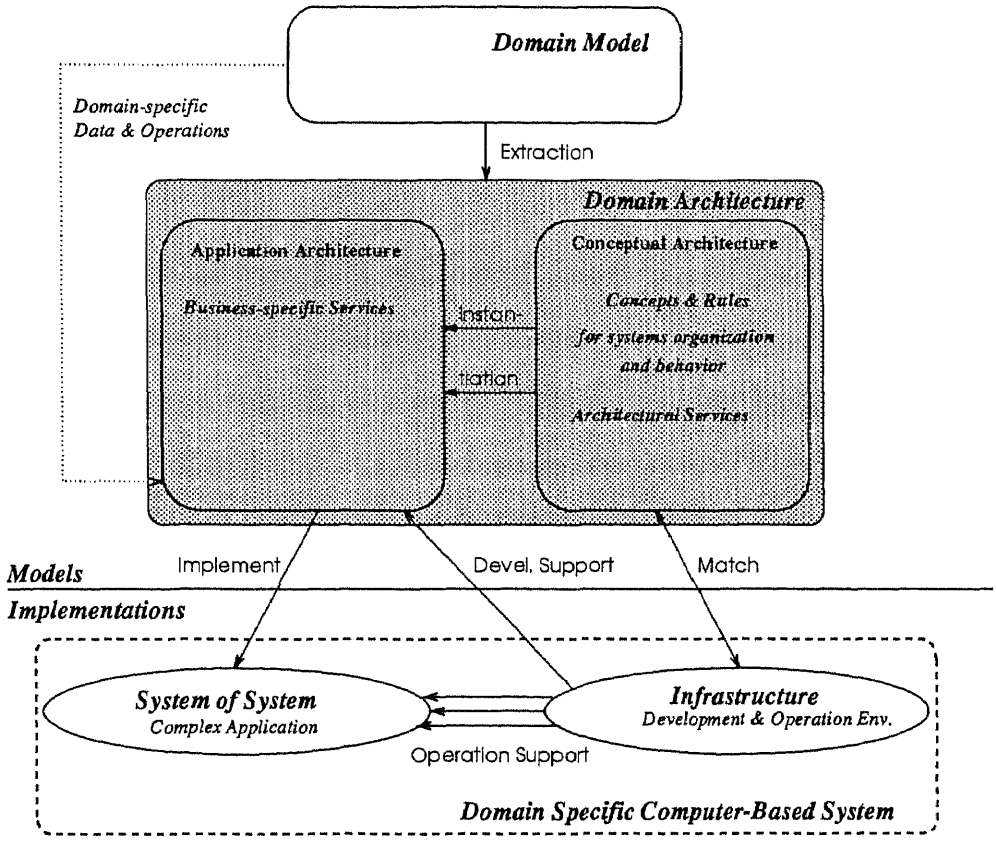


Figure 3.2 Domain Architecture Organization

patterns of information processing in the domain, while the application architecture will provide a consistent view of the information and business-specific services shared between the applications and offered to the end users in the domain.

3.2.3 Domain Architecture: Engineering and Purpose

3.2.3.1 Introduction – the Engineering Context of a Domain Architecture: In order to create a better understanding of the domain architecture concept and its scope, we, first, examine its engineering context. The elements of this context, their relationships and transformations are shown in Fig. 3.3 and discussed below:

1. A Business Domain is a generalized image of the enterprises comprising a business area. As discussed in Chapter 1, from a software engineering perspective, a domain constitutes a global problem space for the software systems developed to support its business operations. As commonly accepted, in order to create a basis for coordinated development of software applications, a domain has to be systematically studied and modeled. The results of this *domain analysis process* are captured in a Domain Model, which has to be constantly updated in order to be kept up-to-date with changes in the domain, [122, 159, 20].

2. During an *architecture engineering process* a Domain Model is "mapped" into a Domain Architecture, in this way providing for the development of an architecture tailored to the domain's characteristics and therefore better serving its needs. This mapping process can be supported by a set of Architectural Styles and (Generic) Reference Architectures such as CORBA, [104], or OSCA, [4].

3. The Domain Architecture serves as a standardized but evolving architectural framework for the domain. The domain-specific collection of business appli-

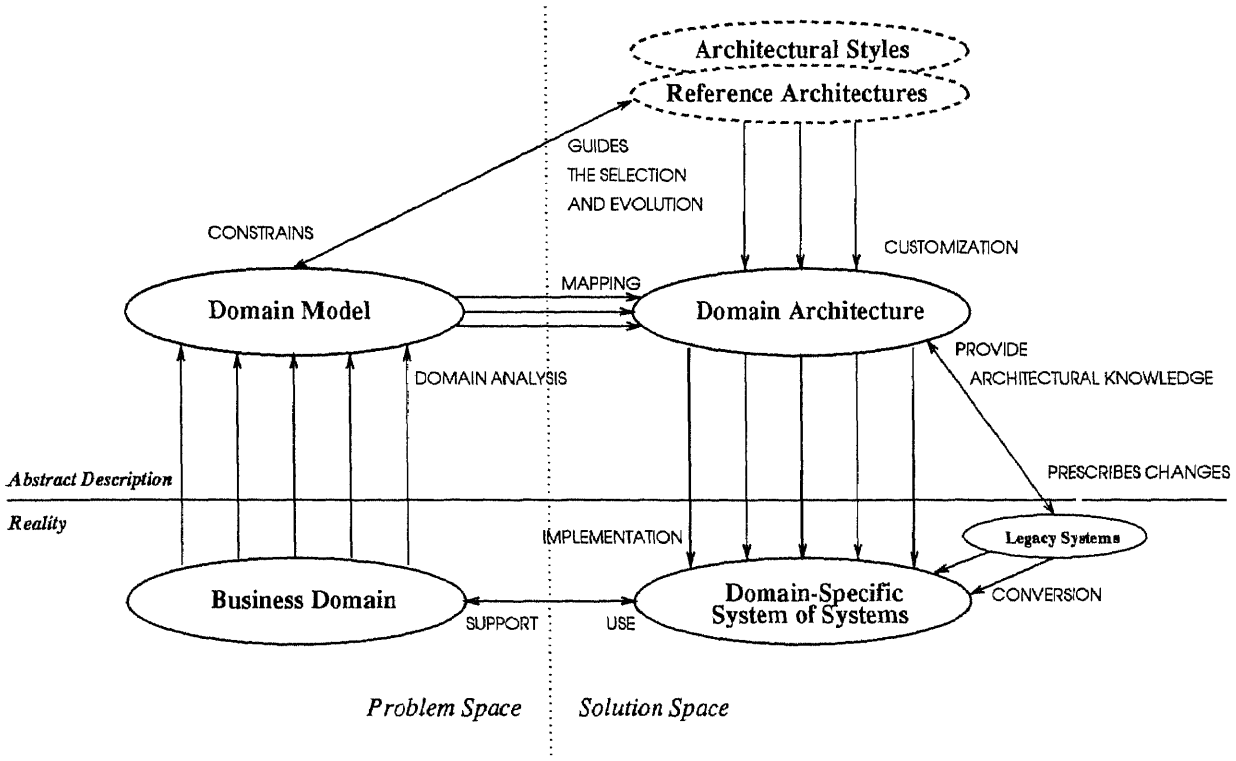


Figure 3.3 Problem Space vs. Solution Space: Reality and Models

cations are then *developed and integrated* into Systems of Systems according to the precepts of the Domain Architecture;

4. A System of Systems is finally utilized to support the business process of an enterprise within the respective Business Domain.

The above elements and transformations are presented from a forward engineering perspective, [124]. Legacy Systems in a domain (see Fig. 3.3) are also an important element of the overall picture. They require special consideration and additional architectural processes to support "architecture recovery" [107] – the extraction of design patterns and their integration into the domain architecture under development.

Two of the above steps directly involve a domain architecture: architecture engineering (step 2) and architecture usage (step 3).

3.2.3.2 Domain Architecture Engineering: The process of defining a common architectural model of software systems developed and integrated in the context of a business domain is referred to as *domain architecture engineering.*

The purpose of this process is to define a domain architecture which corresponds to the characteristics of the domain of interest and can be shared across the systems developed in the domain. The essence of domain architecture engineering lies in abstracting established structural, behavioral, and functional elements in a domain (captured in a domain model); transforming them into design concepts, rules, and service definitions; and finally integrating them into an overall architectural framework. This engineering process is constrained by the need to consider the domain's legacy systems and to satisfy a set of technological precepts, established by the current state of software technology or the characteristics of the existing infrastructures.

The two major entities, affected by an architecture engineering process, are a Domain Model and the corresponding Domain Architecture (see Figure 3.3). They provide two views of a domain, capturing the domain's phenomena, such as entities, relationships, operations, processes, and events, from two different perspectives:

- In the Domain Model – from the perspective of the real-world problems (customer-problems) they represent and solve;
- In the Domain Architecture – from the perspective of the way they work together in a software system to solve the real-world problems.

In order to bridge the semantic gap between the two models and to provide a point of transition, established architectural styles [53], and (standardized) reference architectures, such as ODP-RM, ANSA, OSCA, and "3-tier architecture" can be used. These architectures prescribe system organization and behavior and are a

	Standard Reference Model	Domain Architecture	Relations
Conceptual Architecture	X	X'	(1) $X \sim X'$ or (2) $X' = X + dX$ dX - the change is SMALL.
Application Architecture	$Y = 0$	Y'	(1) $Y \neq Y'$ The Application Architecture is typically introduced in the context of a Domain Architecture.

Figure 3.4 The Specifics of a Domain Architecture

major input to the process engineering the conceptual portion of a domain architecture (see Fig. 3.4).

In this process, two major possibilities for reusing a given conceptual architecture exist. In the best case, an existing conceptual architecture matches nicely with the properties of the domain and no further refinement or adaptation is necessary. Most of the time, however, one or several existing conceptual architectures will be reused but will need integration and further refinements in order to match the specific characteristics of the domain. As the conceptual model represents the property-specific portion of a domain architecture, this is where a useful match must be established with the (extra-functional) requirements of the addressed business domain. Reflecting these requirements into the architecture during the *conceptual architecture engineering* is the essence of the refinement process. For example, extending an architecture to meet specific security needs or to guarantee certain level of data-reliability are typical issues addressed by such a refinement process. A more specific example would be the extension of CORBA, [85], with mechanisms that support mobile objects, as discussed in [125].

It is important, however, that every refinement respects the limits of conceptual architectures. Even after adaptation, the conceptual architecture must remain exclusively property-specific and must not include any problem-specific elements. This

type of refinement is reserved for the application architecture engineering process. Adding, for example, a set of security mechanisms to the original architecture definition is a legal refinement for a conceptual architecture, while adding definitions of a business-aware service, such as "Student registration," to the conceptual model of an University Domain Architecture is inappropriate. This type of service should instead be defined in the context of the application architecture.

Being the problem-specific portion of the domain architecture, the application architecture is the natural recipient for all application-dependent services. The application architecture part of an overall domain architecture is created by allocating the problem-specific, business-aware functions and information, defined in the domain model, to the components of the conceptual architecture. This process is a major step in the development of a domain architecture and is referred to as *application architecture engineering*. The result is an architecture which provides solutions to the customer problems in a given business domain.

In a small case study we have addressed the mapping of a domain model into a domain architecture, [69]. We have reused two architectures - ANSA and OSCA to define a "toy" domain architecture for an Insurance domain. The two architectures nicely complement each other: ANSA defines a model of interaction while OSCA prescribes the application organization. Together they form a conceptual model that matches well with the characteristics of the Insurance domain. The only extension we made to this model was required in order to meet the specific security needs of the domain; we solved this problem by introducing a security server to the combined conceptual architecture (the versions of ANSA and OSCA architectures we used did not address the security aspect). This refined conceptual architecture and an Insurance domain model, [122], served as inputs in specifying the application architecture. The resultant architecture defines a set of services, specific to an insurance business, allocated to the component types of the conceptual architecture.

It also specifies a number of scenarios, prescribing the use of these services across the software components and systems.

This study was performed at the beginning of our work on domain architectures, and the difficulties we experienced have clearly shown the need of methods for architecture modeling and representation that can provide guidance to the process of architecture definition and suitable notations to express the architectural decisions. The results of this case study, the rationale of our decisions, and a summary of our conclusions can be found in [69, 77, 75].

3.2.3.3 The Purpose of a Domain Architecture: A domain architecture is used to support the development of large-scale integrated systems of systems. In this, there are two major issues that must be successfully addressed by a domain architecture in order for it to meet its purpose: (1) dealing with complexity and change, and (2) support for development of composable large scale components or systems that can be integrated into a coherent system of systems.

Complexity and Change:

Like any software architecture, a domain architecture provides a system design specification at a high level of abstraction where a software system is seen as a collection of large-scale interacting components constrained by a set of configuration and behavior rules. It encapsulates the irrelevant (to the overall picture) detail of individual systems but precisely defines their interfaces and prescribes the ways they will interoperate across an enterprise's full ensemble of software systems. Thus, a domain architecture is characterized by a high level of abstraction, encapsulation, and modularity, which are key mechanisms for reducing complexity and managing change.

Composability:

Development of interoperable software applications that can properly work together and therefore can be composed into domain-specific systems of systems is a basic objective for developers of business applications. Providing support for such development is, in turn, a major requirement for a domain architecture.

We define composability as a property that, if exhibited by a component, implies the component's potential to offer and consume services in the context of a system of systems, and hence to work safely with the rest of the system constituents.

Composability assumes:

- *engineering or syntactic compatibility*, which implies compatible behavior and consistent data (type) schemes for the components participating in an interaction; and

- *function-&-information or semantic compatibility,* which implies a consistent interpretation of information and correct usage of functions, therefore a consistent application semantics in the context of the overall system of systems.

"Composability" closely relates to the well-known concept of "openness", [58]. As discussed in Chapter 1, the key principle in the process of development of open systems is the concept of conformance, where by conforming to a standard, a system becomes open to all other systems obeying the standard. We apply this general idea of open systems development to the development of a domain-specific set of business applications, required to interoperate in the context of an overall system of systems.

In order to develop such applications as composable systems which can interoperate across an enterprise, a "software applications standard" is required to provide a point of reference and conformance in the process of development and evolution of business applications that are to be integrated into a system of systems. The argument we make is that, as a common design model of the domain's collection

of software applications, a domain architecture can serve as such a standard, where: conformance to the conceptual architecture creates a basis for syntactic compatibility between the components, while conformance to the application architecture helps to ensure semantic compatibility, see Fig. 3.5. In order to successfully serve as such a standard or a domain-specific framework for components "plug-and-play," a domain architecture has to be:

- *commonly accepted,* so as to be applied in the development and evolution support of all constituents of a domain-specific system of systems;
- *strictly applied,* so that the architectural rules are observed, and developed applications comply to the architecture; and
- *rigorously specified,* so that conformance to the architecture can be verified.

All these requirements imply specific activities in the engineering process, and availability of methods and tools to support the architecture definition, representation and application. In the rest of this dissertation we focus on the latter, namely, on development of methods and tools for domain architecture engineering and representation.

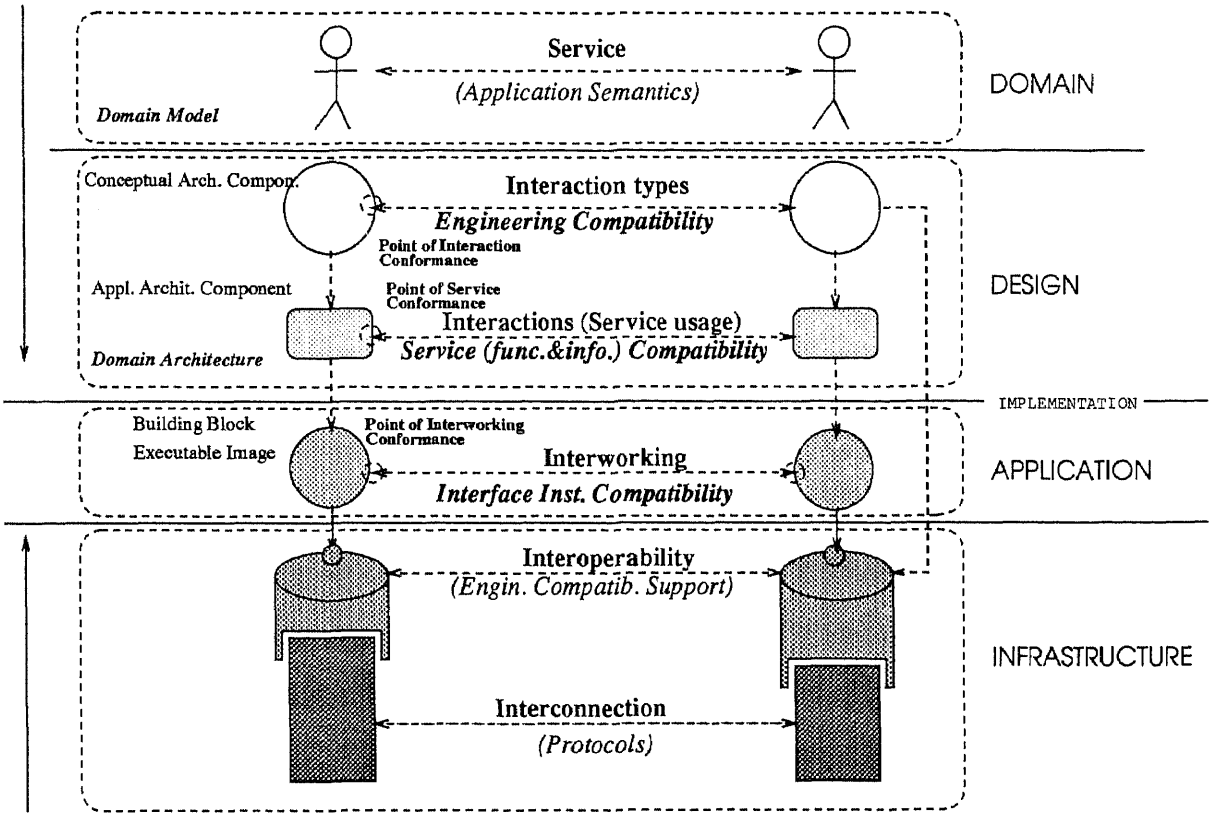


Figure 3.5 The Composability Issue

CHAPTER 4

RELATED WORK

In this chapter we present a summary of related work. We begin with a brief overview of the origins of the field of software architecture and then discuss work in three main categories:

- Foundations for software architectures: conceptual basis and architecture description;
- Architecture in context: a domain perspective and a process perspective; and
- Example architectures: standardized reference architectures and other examples.

4.1 Overview

The evolution of software architecture to its current state of a discipline in software engineering can be summarized as follows. For years software engineers have been creating architectural diagrams to represent the organization of a system under development; to capture the main functional units of the system, its components; and to identify the relations between the components – the interactions that allow the system to function as a whole. This task is performed as part of the system design phase, following the requirements specification, and serves mainly as a transition step. The product of this step is an informal architectural diagram, which serves as a basis for allocating the requirements to the system components. Such a "top level design" or "architectural design" step is present in all traditional design methodologies, [140].

The last decade is also characterized with extensive architectural projects such as APM's ANSA project, [9, 10, 11], and Bellcore's work on $OSCA^{TM}$ Architecture, [4]. In parallel, architectural standardization efforts such as ISO work on a Reference

Model for Open Distributed Processing (ODP-RM) [1], and NIST's Framework for Software Engineering Environments, [2], have been taking place. At the same time the Domain Specific Software Architectures, [95, 151, 149, 148, 150], have been introduced as basis for reuse and component-based development.

As a result of the new demands, the enlarging scope of architectural efforts, and the maturity of the field, a conceptual basis for software architectures started to develop, forming the foundation of the software architecture discipline. In the early work the main question of "What is an architectural design?" and the conceptual building blocks, such as components, connectors and styles, were discussed. The motivational papers, [131, 108, 53], introduce the concepts through examples, and do not attempt to supply specific mechanisms and notations. Rather, they identify the field and call for research in the area. We discuss some of this work in the next section.

4.2 Foundations of Software Architecture

In the context of establishing the foundations of software architecture, the work on architecture definition, architecture modeling and representation, and identification of architectural styles has been of fundamental importance, [108, 53, 3].

4.2.1 Conceptual Background

Consolidating the experience from years of defining architectures and designing software systems, a conceptual basis for software architecture, started to develop. The two main groups contributing originally to this process were Perry and Wolf, and Shaw and Garlan. The problems addressed in this research area fall into three main categories: architectural models, styles, and architectural views.

4.2.1.1 Architecture Models: In [108], Perry and Wolf propose the following software architecture model: *Software Architecture* = {*Elements, Form, Rationale*}. They distinguish between three different classes of elements – processing, data, and connecting elements – and based on them define three corresponding architectural views. The form consists of "weighted properties and relationships", which constrain the choice of elements. The rationale captures the motivation for the choice of an architectural style. The authors do not examine in detail the constituents of the model but rather provide insights into the interdependence among the three architectural views being identified: process, data, and connections. Perry and Wolf also introduce the idea of a style as a property of a class of architectures.

Shaw and Garlan present a model of architecture based on two abstractions: the component, an independent unit of computation; and the connector, an interaction among the components, [53]. The use of components as main elements of architectural models is natural and well-understood. Architectural connectors, however, even commonly used in the research, are still in the process of being accepted by architecture practitioners. Although, the ability to define an explicit connector abstraction is not formally required, there are good practical reasons justifying its use. In [132], Shaw elaborates on a number of reasons for treating connectors separately from components, "connectors may be quite sophisticated," "some information about the system does not have a proper home in any component," "connectors are potentially abstract," and "relations among components may be not fixed."

4.2.1.2 Style: The definition of architectural styles is probably the most thoroughly developed area of the overall architecture-related research, [108, 53, 3, 48]. In [53], Shaw and Garlan identify a set of architectural styles: client-server, black-

board, and event systems, and then introduce components and connectors as the basic units of their representation.

A closely related direction identifies model problems in software architecture and defines a set of corresponding solutions in different architectural styles, illustrating the richness of choices available to an architect, while indicating some of the tradeoffs in choosing one style over the other, [137].

There have been a number of efforts to apply formal methods to the characterization of architectural styles, describing the structure and semantics of single classes of systems. A general treatment of the style formalization problem is presented in [3], where Abowd, Allen, and Garlan provide a formal framework in Z notation [144] for uniform definition of architectural styles. They introduce an abstract syntax based on three entities: components, connectors and configurations, and define semantic models for a pipe-filter style and an event system style. The work establishes a formal basis for defining new architectural styles and style-specific architectural analysis. A further discussion of the issue of formalization is provided in Section 4.2.3.

4.2.1.3 Views: It is also a common understanding now that multiple views (descriptions relevant to a single perspective) such as control and data views, are required to capture adequately the variety of architectural relationships, [54, 46, 159, 77, 133]. Currently, however, even when several system aspects are addressed and several views are created, neither the mapping between the architecture views nor a common semantic basis is established. Most of the existing work focuses primarily on the structural aspects of software systems. The related architectural solutions are reflected into an architectural view, usually expressed in terms of components and connectors.

In the research presented in this thesis, we also use the component - connector[1] structural framework for representing architectures. Components, in our approach, are broadly seen as denotations of the building blocks of a system of systems. The properties of component interactions, their definition and specification in architectural designs, are of fundamental importance in the context of system integration, and therefore a major building block of our model. Components and connectors are, however, just a part of a wider architecture reference model presented in this thesis (GARM).

As the focus of our research is on representing domain architectures, we needed an architecture model reflecting the nature of this type of architecture, the level of abstraction at which they address architectural problems, and their scope and role in an overall development process.

Most of the practical and research efforts, however, have focused on codifying and representing styles at the system level, and on describing specific system architectures at this level. The architecture model introduced by Perry and Wolf provides key abstractions but does not develop a representation system sufficient to support description of domain architectures.

Our model is tailored to the characteristics of reference and domain architectures (see Chapter 5) and has a number of distinguishing features. The model identifies a generic architectural vocabulary of properties and constructive architectural concepts – components, connectors and configurations. It also suggests annotations of patterns and guidelines collected as a feedback from applying a domain architecture. It defines the constructive concepts from several aspects —

[1]Originally, [75], we used the term *contract* to refer to what we now call connector. Our definition relates to the "contract" concept used in ODP-RM , [1], and the "channel" concept defined in ESTELLE, [38]. Later, we adopted the term connector as it became established in the context of architectural vocabulary. We, however, have preserved the term contract for the syntactical element of the architecture specification technique ASPECT used to represent connectors (see Chapter 5).

function (common and application specific), behavior, data, quality, structure, and type – and, thus helps to support the separation between the conceptual and application architecture models, as discussed in Chapter 3. The model also introduces the concept of architectural rules as a means for expressing structural and operational constraints. Rules are a mechanism for representing invariant architectural constraints and organizing them into a "prescription" to be followed when developing particular architectures and systems.

4.2.2 Representing Software Architecture

Suitably representing software architectures, and domain architectures in particular, is an open research problem. The solutions to this problem are, of course, to a large extent preconditioned by the underlying architecture models.

In [135], Shaw and Garlan examine the conceptual aspects of the problem and argue that architecture description can be addressed as a language problem. They outline certain critical properties of architectural descriptions, before all, higher levels of abstraction and distinct semantic entities. Based on this, they define the need to develop specialized languages for describing software architectures and rule out the possibility of successfully using conventional programming languages and module interconnection languages (MILs) for representing architectural designs. We discuss these alternative approaches in more detail below.

4.2.2.1 Non-specific Approaches for Describing Software Architectures:

So-called languages for component description and reuse like LILEANA, RESOLVE, CIDER, surveyed in [154], and MILs, as languages whose purpose is to specify module and system structures, are the closest in objectives to a language for describing the architecture of a software system. MILs, such as MIL75, [35], and INTERCOL, [113], have been developed to support the description of large-scale program structures

independent of any particular programming language or system. They define the structure through definition-use bindings which associate program structures, such as functions, with uses of these structures. MILs increase the independence of system components by creating visibility boundaries around them. By explicitly binding definition to use, they also provide an explicit description of system's structure. They, however, have a common disadvantage from an architecture representation point of view – they assume simple, uniform interactions and concentrate on component descriptions. Further, they do not provide a means of specifying abstract patterns of interaction or of describing families of systems, which makes them not well suited for describing domain architectures.

Module Interconnection Formalisms (MIF) extend MILs with interconnection semantics. They are systems developed to support the composition of software modules based on mechanisms other than definition-use bindings. Different MIFs provide different but specific interaction abstractions as the basis for system composition. For example, the Polylith, [115], system provides support for message passing between components based on software bus abstraction. Each MIF system supports the construction of software systems that use the specific interaction model defined by the MIF; however, it cannot be used to describe or build systems that use other interaction patterns. Therefore, MIFs cannot serve as a general approach for describing architectures.

A similar problem exists with so-called languages for "megaprogramming," [156, 149]. These languages (although defined as implementation, as opposed to design, languages) provide mechanisms for describing system components and interactions at a high level of abstraction and may serve well for representing certain elements and aspects of software architectures, for example, component interfaces, overall system control,and data aspects. They, however, define one specific model of interaction, which is a major constraint when representing heterogeneous sets of

architectural designs constituting a domain architecture (for details see the discussion on megaprogramming in Section 4.3).

A common problem for the integration of systems is the potential for inconsistency in data representations. In order to address this issue, Interface Definition Languages (IDLs) have been developed such as IDL [80, 102], CORBA IDL [85], and ANSA IDL [11]. Like MIFs, these systems define a single model of interaction (typically, remote procedure call or object-method invocation) and do not provide sufficient basis for describing software architectures. An IDL, however, can be used to represent specific aspects of an overall architecture if the systems are to be developed in an environment that supports the respective IDL.

Object-oriented notations are also a class of design (and analysis) notations related to software architecture representation. A number of object-oriented methodologies and notations exist, i.e., Coad and Yourdon [30, 31], Booch [21], Rumbaugh [127], Shlaer-Mellor [139], and Jacobson [66]. Recently, several of these prominent notations have been combined into a Unified Modeling Language (UML) [117, 42]. UML provides a set of graphical primitives for representing the objects in a system design, their attributes, and the static and dynamic relationships between them. The constraints on these objects can be expressed in a separate object constraint language – OCL [116]. The UML diagrams fall into two categories: static structure diagrams and behavior diagrams. Static structure diagrams include class and object diagrams, while behavior diagrams include collaboration, sequence, use case and activity diagrams. Static structure diagrams define the elements of the model – types, classes and objects – their attributes, methods, and their definitional and referential relationships. Definitional relationships include generalization-specialization and instantiation relationships, namely, they include objects which are instances of types or classes, classes which implement types, and types which are subtypes of other types. Referential relationships include part-of and association; they define objects

that are associated with other objects, that are part-of other objects or contain other objects. UML also provides diagrams to capture the behavior of a system. For instance, a collaboration diagram represents a sequence of interactions between the participants in a collaboration. It includes the participating objects and the pathways of communication annotated with the names of the messages or operations. The collaboration diagrams can be combined with state-machine representation of components based on Harel's Statecharts, [62]. Object-oriented notations, however, do not distinguish between the description of a family of systems and a specific system, which makes them not well-suited as a means for representing reference architectures. They also define a specific semantics which does not naturally lend to the description of a heterogeneous set of architectural designs forming an overall reference architecture.

4.2.2.2 Architecture Description Languages: A number of specialized languages for representing architectures of software systems, Architecture Description Languages (ADLs), have recently emerged. Examples include: Wright, UniCon, Rapid, Aesop and ACME. Although all of these languages are concerned with architecture description, each addresses the architectural issues at different level of abstraction and focuses on different architectural aspects. Wright is a CSP-based language which supports formal specification and analysis of component interactions, [7, 8]. UniCon is a general-purpose architectural description language based on components and connectors, [134, 136]. Rapid is focused on event-based systems and provides facilities for simulation of architectural designs, [87]. Aesop is an environment that supports specification and reuse of architectural styles, [48]. ACME is a generic architecture description language intended to serve as a least-common-denominator interchange language for architectural descriptions, [49]. It is being developed as a joint effort of the software architecture research community.

The goal is that a specification written in one language, say Wright, could share a common architectural structure with a specification in another language, say UniCon, and thus the architect could take the advantage of multiple tools for architecture description and analysis. A number of languages have been surveyed recently by the Software Engineering Institute and some results are published in [29]. Below we address the basic characteristics of Wright, UniCon and the Aesop system.

The **Wright** model of architectural description is based on the idea that interaction relationships among components of a software system are directly specifiable as protocols which characterize the nature of the interactions, [7]. The protocols are specified in a subset of CSP, [64]. This formal foundation makes the language especially suited for architectural analysis such as deadlock and composability analysis. A system architecture is described in terms of component and connector types, their instances, and the associations between these instances, called attachments.

UniCon is an architecture-description language intended to aid developers in defining software architectures and in making transition to code. The scope of the language is system development. The objectives of UniCon developers are best expressed in [136]: "Currently, we are more strongly motivated by the practical utility of the model than by formal foundations." The language is based on the component-connector view. Both components and connectors are first class compositional entities (currently the decomposition of connectors is not supported). UniCon supports a fixed collection of interaction types, such as pipe, procedure call and remote procedure call, and provides a number of pre-defined medium-grain components which include module, process, filter, and shared data. UniCon provides a tool for constructing executable configurations based on component types and implementations, and on "Connection experts" that implement specific connector types. Each connector has a collection of roles and each component

additional "interface" construct. This allows grouping of related ports (access points) and representing the component's data and information views, which is of significant importance when describing domain architectures. It supports multiple interfaces, in order to provide facilities for expressing different functional views of a component. In addition, the description of architectural elements in GARM-ASPECT comprises three separate portions (header, body, and external view, e.g., interfaces for a component) which is different from the port-properties representation scheme of the other ADLs. The header explicitly defines the element's place in architectural hierarchies and in this way explicitly reflects the two abstractions addressed by our model, generalization-specialization and aggregation-decomposition. Bodies represent the encapsulated component's or connector's "implementations." The interfaces, ports and roles represent the external view of components and connectors, on a specific level of decomposition. (For an introduction to GARM-ASPECT see Chapter 5.)

4.2.3 Formalization of Software Architecture

Recently there has been increased interest in providing a formal basis of architectural descriptions so that analyses and verification can be applied early in the development process. Our main goal in this thesis is to develop a notation in which architectural structure can be consistently described. However, we also provide a set of "hooks" for relating and linking the structural specifications with external tools and formal systems in which architectural properties and patterns of interaction can be modeled and analyzed.

4.2.3.1 Formalizing Individual Systems and Architectural Styles: Efforts to formalize software architecture began with applying formal methods to individual software systems and, as mentioned earlier, to specific architectural styles. In the

interfaces with its environment by players (ports). At the configuration step players and roles are associated. UniCon also provides a facility for integration with external timing analysis tools.

An approach to describing architectural styles and to providing support for automated architectural design is demonstrated by Aesop, [48]. **Aesop** is a system for developing style-specific architectural development environments based on a generic object model for representing architectural designs. Each environment supports a set of style-specific components and connectors; checks the style-specific compositions of design elements; supports optional specifications of the elements' semantics, by using style-specific languages and external tools; and provides mechanisms for style-specific visualization of architectural information. The approach to architecture representation in Aesop is based on an ontology of generic design objects (components, connectors, configurations, ports, roles, representation and bindings). Style-specific vocabularies are introduced by defining sub-types of the basic architectural classes.

With regard to the general idea of introducing a generic architectural vocabulary and building refinements from it, our approach to architecture representation is conceptually close to those of Aesop and UniCon. The supported vocabularies overlap to a certain extent, as we share the component–connector model. The objectives of the methods, however, are different: describing architectural styles in Aesop, describing system architectures in UniCon, and describing domain architectures in ASPECT. Therefore, the extended underlying representation models, reflecting these objectives, and the representation approaches differ. For instance, the GARM-ASPECT method introduces the concept of architectural rules which make the invariant properties of an overall architecture explicit. Our model examines and interrelates multiple aspects of architectural elements, i.e., structure, behavior, data, function, quality and type. The representation approach introduces an

context of specific system architectures, general formal methods are used to analyze and verify certain critical system properties. As these formalizations are unique and applied only to specific views or parts of an architectural design, they do not create a basis for analysis of the overall architecture for consistency or completeness and they are not directly applicable to other similar systems.

More recently there have been a number of efforts to apply formal methods to the characterization of individual architectural styles. The results are descriptions of the organization and semantics of specific classes of software systems, for instance, event-bus systems [50], or pipe-&-filter style [6]. The notation of choice has been Z [144]. These formalizations apply to classes of systems and establish sets of common verifiable properties shared by all systems in the defined class; in this way, they precisely define the membership requirements for a class. These efforts, however, have been undertaken as individual studies of specific styles and do not provide a common reusable framework for formalization of architectural styles.

A common framework (in Z) for style formalization is defined by Abowd et al. in [3]. This framework introduces three parts of a style definition: style-specific vocabulary, a semantic model, and a set of interpretation mappings from the configuration syntax into the semantic model. On the positive side, the framework provides guidelines for the process of style formalization and in this way creates a basis for comparing styles. On the negative side the practical applicability of the framework is limited by the use of unique style-specific semantic models and style-specific syntax.

A similar approach to style formalization is provided by Mariconi et al. [100, 101] with two major differences: first, they use first-order logic as basis for the definition of the styles and, second, they have further developed a model for style-based refinement of architectural specifications.

4.2.3.2 Formalizing Architectural Connectors: A more practical approach to the formalization of architectural styles and specifically of component interactions is taken by Allen and Garlan in [7]. They define a framework of common notation rooted into a common syntactic and semantic basis. The objective of the method is to define a formalization approach in which architectural patterns of interactions in software systems and architectural styles can be described and analyzed. The selected formal basis is CSP, [64].

There are multiple formalisms based on discrete actions (e.g., state-machines and process algebras) as well as tools that support the analysis of such systems. A number of formal models are based on state-machines, Harel's Statecharts [62], the I/O automaton model of Lynch and Tuttle [88], and Petri nets [109]. In each of these models the execution is a set of traces, where each trace represents a possible path though the state graph. These systems provide basis for analysis and can, for instance, be analyzed for deadlocks or reachability. For the purpose of modeling architectural patterns of interaction, however, these methods lack a key element – they do not provide mechanism for representing the points of decision in a system.

In this respect CSP, [64], has a major advantage, which has made it the formal basis of choice for Allen and Garlan's framework for formalizing architectural connectors and the Wright ADL, [7]. Instead of describing a system as a collection of states combined into a transitions graph, in CSP, behavior is described as an algebraic model of processes. In this model complex behaviors are constructed from simpler ones via a small set of operators which include: sequencing, alternative and interaction. Handling of choices in CSP is an important property. CSP supports two forms of choice: internal (the object modeled by the process controls which behavior will occur) and external (the decision is made by the environment). This distinction is key to the description of such critical architectural properties as the dynamic behavior of intercomponent interactions and the components' responsibility

for making decisions during an interaction. Another piece of work in support of the choice of CSP as basis for Allen and Garlan's framework is the development of the FDR ("Failures, Divergence, Refinement") tool, [145], that automates the test of whether one process refines another. (A process P refines a process Q if the behaviors of P are consistent with, but possibly less general than, the behaviors of Q.) This provides strong analytical capabilities as it provides guarantees of substitutability of one process for another.

The choice of CSP as a semantic basis for Wright, however, limits the type of architectures that can be directly expressed; they have to be static. There are other efforts (e.g., Darwin, [89], and Rapid, [86]) that have addressed certain aspects of architectural dynamism. They, however, do not provide such extensive support for analyzing the dynamic aspects of software architectures as Wright provides for the static ones.

4.3 Architecture in Context

In this section we discuss related work addressing the architecture as a tool in system construction. These efforts fall into three different categories: domain-centered development, systems integration, and development process factors and economics.

4.3.1 Domains, System Integration and Software Architectures

Software architectures are major elements in domain-centered approaches to software construction. As the definitions of domain differ, so do the roles and the scopes of the corresponding architectures. For instance, a distinction should be made between the concepts of an application domain and the corresponding (application) domain architecture, and a business domain and the corresponding (business) domain architecture. The intuitive difference is as follows: an application domain architecture is an architecture for a family of systems with characteristic functional and structural

roperties, while a business domain architecture (discussed in this thesis) is an chitecture for the class of integrated systems of systems (and more specifically, tegrated business applications). A business domain architecture goes beyond an plication domain architecture, and defines how systems from different families (or plication domains) interoperate and form a business' full ensemble of software stems. These architectures have also a number of common properties. They e reference architectures and have similar roles in the development process; for stance, both are required to support reuse and component-based development. he difference is mainly in the scale and the heterogeneity of software systems they ldress.

An architecture for systems integration, can be scaled to address interactions ross business domains' boundaries. This view is presented by the megaprogramming approach, introduced in [156]. The issues of system integration are also ldressed by enterprise architectures. Both will be discuss later in this section.

3.1.1 Application Domain Architecture: Application domain engineering ameworks have resulted from years of research on software reuse, [60]. This research as been based on the conjecture that there exists no universal characterization of le information that could be of value to a reuse algorithm, and on the belief in le need for domain-specific approaches to reusability, [60]. Fundamental to this proach, as to our own research, is the concept of a domain. The understanding what an application domain is, the structure of the domain analysis process, and le results expected from it vary in different approaches. However, experimental ork done by different schools has shown a common trend – "dramatic improvement the amount of reuse when the domain analysis is successful in deriving common chitectures, generic models or specialized languages for a specific problem area", 11].

Triggered mainly by benefits from the reuse incentive, a substantial amount of work has been done in identifying software architectures associated with particular families of systems each solving specific problems, for instance, office systems or network management systems, [36, 108]. These architectures are defined to serve as reusable reference models in the development of systems from the specific family. They serve as frameworks for component-based development and, therefore, re-use of application-specific components.

In order to consolidate this work, to establish common theoretical grounds, and to stimulate the development of reference architectures for several domains of special interest, such as Command and Control Systems, [22], ARPA has sponsored multiple projects addressing the issues of Domain Specific Software Architectures (DSSA), [95, 151, 149, 148, 150].

The pair "domain–domain architecture" is fundamental for the DSSA initiative. The definition of the (application) domain concept in the context of ARPA's DSSA program is based on Prieto-Diaz and Cohen's technique, [111, 112], and can be summarized as follows: An application domain is: (1) a collection of problems (where the domain analysis result is a theory of problems in the domain) and (2) a collection of existing or future applications perceived to be similar (where the result of the domain analysis is a taxonomy of system components and relationships), [149]. In this approach the domain analysis process is seen more as an abstraction of system design, leading to the specification of components and characteristic patterns of organizations representative for the particular domain, [77]. The resulting interconnection structure (of components) for the class of systems in the domain is referred to as a Domain-Specific Software Architecture and is seen as a framework for software reuse.

There are three distinct directions under the overall approach to architectures within the DSSA program. The Teknowledge project is based on a particular

software architecture, associated with the work of NIST in robotics control, [2], which proposes a multi-level hierarchy of controllers. The Honeywell and ORA projects' approach to software architecture is quite different. Honeywell proposes multiple formal engineering models (e.g., for descriptions of control laws, for schedulability analysis, for determination of reliability) for the domain of Guidance, Navigation and Control. Software architectures are derived from these models, [5, 17]. The IBM and GTE projects are based on domain modeling approaches, [112]. This approach identifies the critical aspects of a domain or class of problems as the experts in a domain perceive them and captures them in a domain model. The model is independent of any implementation. A software architecture is derived from the domain model. The model describes a family of problems, while the architecture describes a family of solutions, [130]. The view taken by the IBM project is closest to our approach, in that the domain modeling is only concerned with the problem domain while the architecture represents the (software) solution space.

Reuse opportunity (from a market perspective) is also the driving force behind the process of defining so-called "conventional software architectures" for classes of related systems or subsystems, [20]. The goal of this "product line approach" could be stated as fast (component-based) development of high quality software for well-established market niches, with software architecture as the underlying vehicle (we will discuss this approach in more detail in the next section).

A practical experience from development and application of a "product line" architecture is reported in [33]. Crocker and Engelsma describe the efforts of Motorola Cellular Infrastructure Group toward building organizational competence in software architecture. They summarize the company's initial efforts at developing an organization-wide understanding of architectural issues and on the difficulties they have had experienced in evaluating a third-party's software architecture specifications. A number of problems they report, such as "lack of preciseness in the

specification," ambiguity, and lack of established commonly understood vocabulary, match the problems we have identified, [69], and which triggered our work on architecture modeling and representation.

The "application domain – domain-specific software architecture" methodology exemplifies the developers' point of view and is driven by the benefits of "reuse during development". Our "business domain – domain architecture" approach, in addition, focuses on the interoperation and system integration issues. In both cases, however, the architecture serves as a reference framework for a unified design, development coordination, and technology standardization in a domain (application and business, respectively).

4.3.1.2 Enterprise Architectures: Closely related to the (business) domain architecture is the concept of an enterprise architecture, [19].

Originally, the work on enterprise architectures has been concentrated on establishing conventions that enable interoperability between the tools within and across software environments, or, in our terms, on standardizing the domain's infrastructure. Typical examples are the HP-NIST-ECMA "toaster" model, [2], and the AT&T's BaseWorkX, [36].

Recently, a different type of enterprise architecture has emerged, addressing information technology needs of an enterprise, e.g., the Air Force Information Architecture, [39], OSCA, [4], and "3-tier architecture", [128]. This change of focus in enterprise architecture-related work goes together with the intensive work on providing engineering and technological support for development, integration, and operation support of enterprise-wide information systems. Most of the new interest has been stimulated by the acceptance and the application of object-oriented technology in this area, [91, 128, 147, 32].

The (business) domain architecture, as defined in this thesis, integrates both of the above views. The conceptual portion of a domain architecture is a means for standardization of the technology and the tools in a business domain and corresponds to the traditional enterprise architecture concept. The application architecture corresponds to the second view – it captures and imposes conventions about the provisioning of "business-aware," application-specific services across an enterprise and establishes a bridge between the knowledge of the specific business domain and that of the domain of software.

4.3.1.3 Megaprogramming – Beyond the Business Domains: The issues of integration and service sharing across the boundaries of business domains is addressed by Wiederhold, Wegner, and Ceri in their paper on megaprogramming, [156]. The authors propose a framework for megaprogrammming as a technology for programming with large modules, called megamodules. Of primary concern are the computations that span several software components. The approach encompasses not only largeness in size but also persistence (largeness in time), diversity (large variability) and infrastructure (large capital investment). The megaprogramming paradigm models real-world services and supports parallelism, distribution, and dynamic changes of the interface behavior. It also addresses the structure of the constituent megamodules.

Megamodules are defined as entities that capture the functionality of services provided by large organizations like banks, airline reservation systems, etc. They are perceived as internally homogeneous, independently-maintained software systems managed by a community with its own terminology, goals, knowledge, and programming traditions. Each megamodule describes its externally accessible data structures and operations and has an internally consistent behavior. A megamodule's capabilities are provided at its interfaces. Megamodules realize greater abstraction power

than traditional modules by stronger encapsulation mechanisms: they can encapsulate not only procedures and data, but also types, concurrency, knowledge, and ontology. Programming within the megamodule can be handled by traditional technology, while computations spanning several megamodules are specified by megaprograms in a megaprogramming language. A megaprogramming language (MPL) is introduced which supports flexible composition of megamodules – both synchronous and asynchronous invocation schemes, decentralized data transfer, parallelism, and conditional execution. Megaprograms, written in MPL, provide the "glue" for megamodule composition.

Most importantly the work establishes a high level interaction framework based on the following:

(1) The traditional CALL statement (for invoking remote procedures and passing parameters) is segmented into three distinct statements: SUPPLY (to provide global arguments from a megaprogam to a megamodule), INVOKE (to cause the actual processing), and EXTRACT (to extract results from a megamodule).

(2) Data interfaces are provided for SUPPLY and EXTRACT. The data structures supported by a megaprogramming declaration enable an application to provide input and output parameters of arbitrary complexity. Input and output parameters of megamodules are presented through database-like schemes. Megamodules provide self-describing EXPORT data structures that should be compatible with the MPL type system. The megamodule, however, is not required to handle all of the MPL type system, but only to support all of its EXPORT data structures. (The megamodule's internal type system is of no concern.)

(3) Flexibility and asynchrony are enhanced by allowing autonomous execution of the megamodules, while the invoking megaprograms retain great flexibility as MPL statements can monitor, inspect, and constrain the execution of megamodules and can accept partial results.

The main objective of megaprogramming, as defined in [156], is to provide flexible support for construction of large and complex systems. The supported view of a megamodule is similar to what we call a constituent system of an integrated system of systems. The difference is that in the context of megaprogramming, the megamodules provide services across business domains, while in the context of our framework the services are shared inside a business domain. Apart from that, the megaprogramming framework addresses the issue of system integration and suggests an approach for achieving it. From our perspective, it creates an architectural framework, as it defines the organization of the target composite systems and precisely defines an interaction model. To this end, we have examined the megaprogramming approach as a possibility for organizing an integrated system of system (this aspect was briefly addressed in Chapter 2).

4.3.2 Architecture and Software Process

In this section we summarize work addressing the relationship between an architecture and the rest of the development process artifices, the architectural processes themselves (architecture recovery, architecture engineering, etc.) as well as the organizational structures and technology required to support an architecture-based system construction.

4.3.2.1 Megaprogramming – Software Process Perspective: Boehm and Scherlis, in their paper "Megaprogramming," [20], address the development process aspect of megaprogramming[2] in the light of economic factors and incentives and,

[2]In [20], megaprogramming is defined as a coordinated approach to a broad range of management and technology facets of a software engineering practice that suggests an unified use of different technologies, including software reuse, software engineering environments, software architectures, and application generation, in order to provide a component-oriented "product-line" approach to software development (compare with the view presented in Section 4.1.2.1).

among other elements, identify the role of an architecture in the development of (application) domain-specific software systems.

In the paper, four principal features of a successful reuse effort are outlined and the coordinating role of an architecture in the context of a development process is clarified:

Product line approach: The "product-line" approach is introduced as a methodology that encompasses the aggregate return on investment in a set of related software products (components). The paper substantiates the approach by arguing that in the context of a marketplace, where customers seek flexibility and reduction of adoption risks, and producers seek market expansion and new niche opportunities, production based on components and conventionalization of architectures becomes a natural process. (Engineering conventionalization is a process of developing conventional approaches – approaches for which there is a supporting consensus.) The result should be a facilitated marketing of components aimed at particular architectural niches common across families of systems or subsystems.

Domain oriented software architectures: The authors make the argument that a component oriented development and reuse opportunity creates a natural incentive to achieve conventionalized software architectures for classes of related systems or subsystems, where architecture specifies the overall interconnection structure of components constituting the system.

Technology support: A view is presented that a component-oriented approach, supported by the definition and use of architectures, requires effective technology support, as methods and tools are required to resolve differences and achieve compatibility of data, control, timing, and, most importantly, shared assumptions.

Reuse roles: An argument is made that reuse will meet its potential only if the organization can orchestrate and combine the full range of roles – product line managers, component producers, component brokers, and component users. Again,

the point is made that the challenge is to understand what the assets are – as the actual assets in software engineering are conceptual designs (conceptual architectures) rather than physical product elements.

Further, five major elements, contributing to a successful adoption of the megaprogramming approach, are identified:

(1) *Architecture determination*: an ongoing architecture engineering process, where domain analysis is used to identify and conventionalize the structure according to which components interact in a system, in order to agree on common interfaces that enable and encourage sharing, and effective interoperation of components from diverse sources. The goal of this continuing process is an evolving "market structure" for component interchange in specific domains.

(2) *Architecture and component description* (in order to facilitate acceptance of externally produced components and architecture elements): This element is seen as a mix of formal and informal descriptions, both of individual component interfaces and of software architectures. (Notice that the architecture description here is only the description of the configuration, which is different from the view we present in Chapter 5, where an architecture description includes not only the description of the configuration but also the description of the elements.) It also includes tools to support reasoning about the descriptions.

(3) *Component construction, specification, verification, and adaptation*: The paper points out that, although megaprogramming emphasizes the assembly of components into systems, adoption of the approach has a clear influence on the manner in which individual components are constructed. Components within a family of related components (effectively a product line) must be specified using the domain-specific architecture conventions enabling sharing of elements.

(4) *Component assembly or composition* (including assessment, acceptance-validation, support, and evolution of components): Megaprogramming is seen as

a methodology that supports construction of large systems by assembling them from powerful compositional units according to architectural concepts that themselves provide a degree of a priori validation. Hence, in the context of system composition, the descriptive specifications supporting the architecture design become prescriptive specifications supporting component integration. Subsequently, component composition and assembly is seen as a consumer activity in megaprogramming. To achieve these goals, new languages are needed to specify fully the structure and characteristics of such a composition of components.

(5) *Component interchange*, including search for and identification of candidate components, evaluation, and incentives to place components into an interchange or marketplace. Producers and consumers of architectures and components find each other by means of brokerage functions.

In summary, the paper takes the software process and its economics point of view, when introducing megaprogramming approach. It argues that the high short-term cost associated with the introduction and support of conventionalized architectures is a "correct" solution, in that the overall long-term cost is lower. The views expressed and the conclusions made in this motivational and directional paper, regarding the role of an architecture, the benefits from it, and the requirements imposed by it on the engineering process, are applicable to (business) domain architectures as well.

4.3.2.2 Process and Integration Architectures: The process[3] aspects of applying architectures to support system integration is addressed in [159, 124]. In [124], Rossak, Zemel and Lawson define a meta-process model for planed development of integrated systems: an architecture, called Integration Architecture, is

[3]This work is the basis from which the research presented in this dissertation originated. The two aspects: process, addressed in the previous work; and architecture, addressed in this dissertation, complement each other into a framework for complex systems development; see section 4.2.3.

the core controlling element of this model. In [159], Zemel introduces an extended process model for development of Mega-systems – large-scale software systems. The Mega-system architecture is one of the major building blocks of the model, which guides the rest of the process activities. The model defines in detail the role of the architecture in respect to the rest of the process artifacts (for details see the next section on GenSIF). Recently, in [107], Perry has also acknowledged the specifics of a business domain architecture engineering process: problem vs. implementation focus; and populating of architectural models with domain elements.

4.3.2.3 More Process-related Issues: The influences of an architecture on the software process and organizational culture are addressed in multiple publications.

Adopting an architecture-based development requires changes and new elements in the development process. For example, the need to accommodate "architectural reviews" in the development process and the role and influences of these reviews are addressed in [90]. Marazano *et al.* discuss AT&T's experience with conducting architectural reviews to evaluate and validate architectures. AT&T Bell Labs have developed checklists and methods for the formal review of architectures in development organizations. The checklist identifies common, critical issues for the type of systems being developed by AT&T. These issues include administration and maintenance, error recovery, system integration, and overall system performance. The method defines how to address this issues in the context of an entire development effort. In particular the method defines two formal reviews, a "discovery" review to air issues that have to be addressed by an architecture before the architecture is developed, and a "validation" review to evaluate an architecture and discover problems before the architecture is applied and the detailed design and implementation take place.

It is interesting to note that the approach pays a special attention to the interactions between different system components and subsystems, due to the fact that more than 50% of the reported problems are related to poorly-defined communication interfaces. This strongly supports our approach by demonstrating the need for explicitly defined interactions between different constituents of an integrated system

Organizational issues (such as lack of common understanding of architectures, and lack of organizational structures to support the architecture-based development) related to the adoption of organization-wide software architectures are addressed in [33]. In [28], Clements provides some initial results identifying correlation between architectural influences and decisions in large system projects – the conclusions are based on information extracted from a number of case studies conducted as part of system development projects in multiple domains, including air traffic control, shipboard fire-control, military command centers, machine controllers, database management systems, and flight simulators.

Even though the process aspects of architecture-driven development are not in the main focus of this dissertation, a clear definition of (domain) architectures role in the development process is required, as it imposes requirements for an architecture's properties and the form of its representation.

4.3.3 GenSIF

GenSIF[4] has been proposed as a framework for studying the issues of software development and integration in large-scale domain-specific development efforts. It

[4]Special attention is given to GenSIF in this survey as this framework is the foundation for the research on domain architectures presented in this thesis. The research on domain architectures complements the previous work on process models for software construction and on domain analyses. All three elements together form a framework for system construction and integration, called GenSIF, [121, 124, 159, 122, 75, 76, 81]. Even though our focus in this section is on the previous process-related work, we also show how the elements (previous and current) fit together. The process-related figures, included in this summary, are adapted from previous publications but have been changed to reflect the domain architecture, discussed in this thesis.

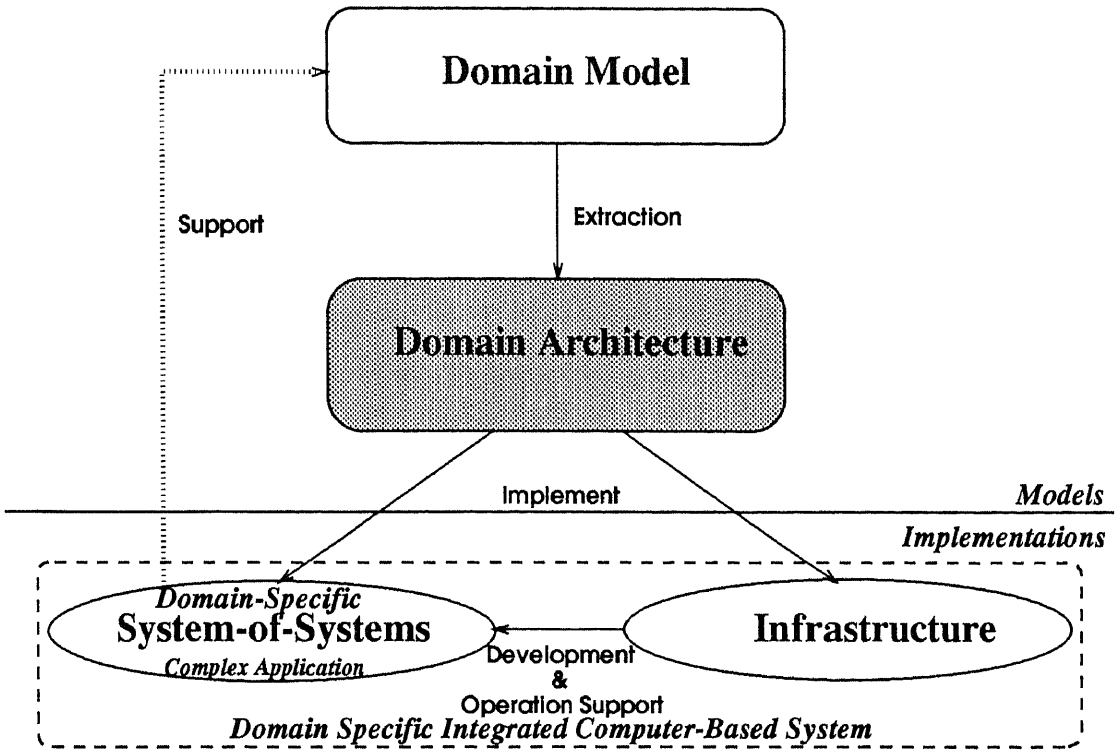


Figure 4.1 The GenSIF Framework

introduces a discipline for domain analysis, architecture engineering and infrastructure acquisition. The framework unifies a variety of views and experiences in software and knowledge engineering disciplines and provides a flexible generic model for systems integration.

The GenSIF approach to system construction can be described best as "planned-for-integration" [159] development. To achieve the goals of systems integration GenSIF relies on new control structures and new artifacts – domain model and architecture – in the development process. The framework also suggests the development or acquisition of domain-specific infrastructure. The major elements of GenSIF are as follow (Fig. 4.1)

- **Domain model**

  A domain model is a description of the domain that reflects the understanding of the domain experts and the needs of system developers, [122];

- **Domain architecture**

  A domain architecture, as defined in this dissertation, is a reference software architecture, tailored to the specifics of a (business) domain, [76, 81];

- **Infrastructure**

  A domain infrastructure is a set of domain-specific tools and common services which form an uniform development and operation support environment, [159, 71].

- **Domain-specific System of Systems (S^2)**[5]

  Enterprise-wide integrated software systems, in GenSIF, are systems of systems.

The above elements are pulled together by the GenSIF process model, discussed next.

4.3.3.1 The Extended Development Process in GenSIF: A fundamental result of the previous work on GenSIF is the introduction of a process model for development of domain-specific large and complex software systems (in [159] called Mega-Systems). The model binds together the elements discussed above – the domain model, domain architecture[6], and infrastructure – and relates them to the elements of a traditional project/system level development process, see Figure 4.2.

[5]Originally, the "system of systems" concept was defined by Eisner, Marciniak, and McMillan, [41], as a set of independently-acquired systems, each under a nominal systems engineering process. These systems are, however, interdependent and form in their combined operation a multi-functional solution to an overall coherent mission.

[6]In the previous work two other terms were used: Integration Architecture and Mega-System Architecture. In this thesis the semantics of the architecture concept has been changed to also address the issues of integrating heterogeneous sets of business applications with (possibly) different architectural designs and information views, and is referred to as a domain architecture.

The Two-leveled Development Concept:

Underpinning GenSIF's model of software development and integration is the concept of *"two-leveled development,"* introduced by Rossak, [121, 124]. The two levels, as portrayed and related to each other on Fig. 4.2, are respectively a *domain-wide level* and a *project/system level.* The domain level is focused on providing global solutions for an entire domain, while system development is concerned with solving a particular problem in the domain. The elements of the domain level establish a reference framework of standardized concepts, design rules, tools and methodologies that constrain but also coordinate and guide the development efforts at the project/system level.

This separation of concerns between the two levels helps to meet the objective of the framework: to leave the project teams as much freedom as possible, but also to provide common measures for project coordination. Each project can then pursue its own organization as long as it complies with the requirements of the domain level. In this way the delivered product is in conformance with the standards established by the domain level, and can be integrated into a target system of systems.

The Model:

The so-called *GenSIF process model,* introduced by Zemel and Rossak, [124], implements the two-leveled development concept[7]. It upgrades the currently used software development life cycle models, such as Waterfall or Spiral models, in order to overcome the problems encountered during the development of large and complex systems. "The major point is not to revolutionize the way systems are developed but rather to change gradually ..., preserving knowledge, tools and environments available for traditional development processes and at the same time introducing an additional level of reasoning and control above the single project", [123]. Figure

[7]A detailed description of the model can be found in Zemel's dissertation, [159]. The focus in this dissertation is on mega-system development and the process is referred to as MegSDF process model.

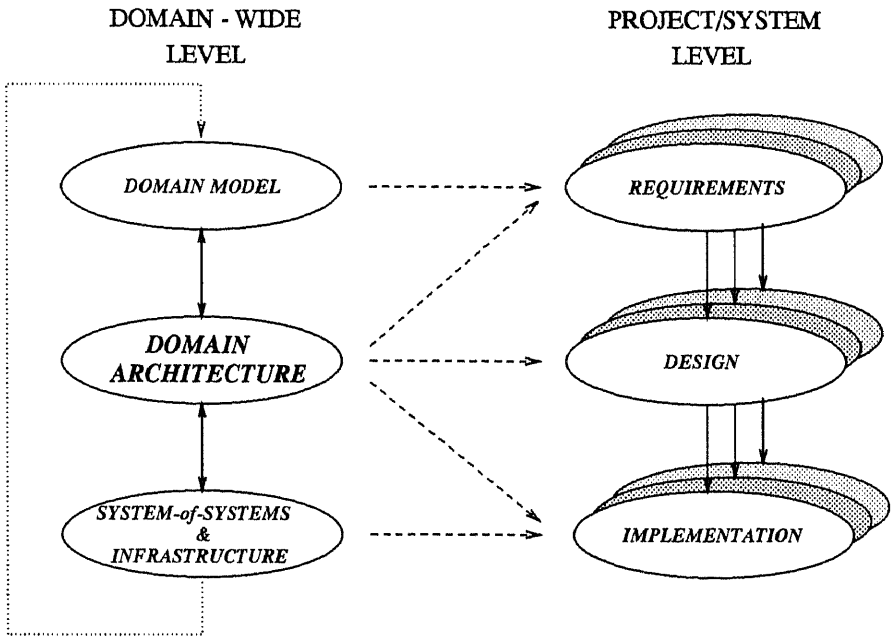


Figure 4.2 The Two Levels of System Development in GenSIF

4.3 illustrates the model (a variant of SADT notation is used). For simplicity the feedback loops are not shown.

According to the model, multiple SYSTEM TASKS (projects) can be active at the same point in time. The internal structure of the system tasks corresponds to a traditional system development process, transforming specific requirements into software systems (see project/system level in Fig. 4.2). In order to coordinate the system tasks (the single projects) and to help guarantee results that can be integrated into a system of systems (and not a set of unrelated systems), a DOMAIN TASK is introduced.

The domain task is constituted by three activities or sub-tasks: domain analysis, mega-system (domain) architecture design, and infrastructure acquisition, which are respectively responsible for the three artifacts: domain model, domain architecture, and infrastructure (see Fig. 4.4). These three sub-tasks complement the system tasks by establishing an additional level of coordination in the domain, where: (1) the domain model allows project teams to handle requirements and to

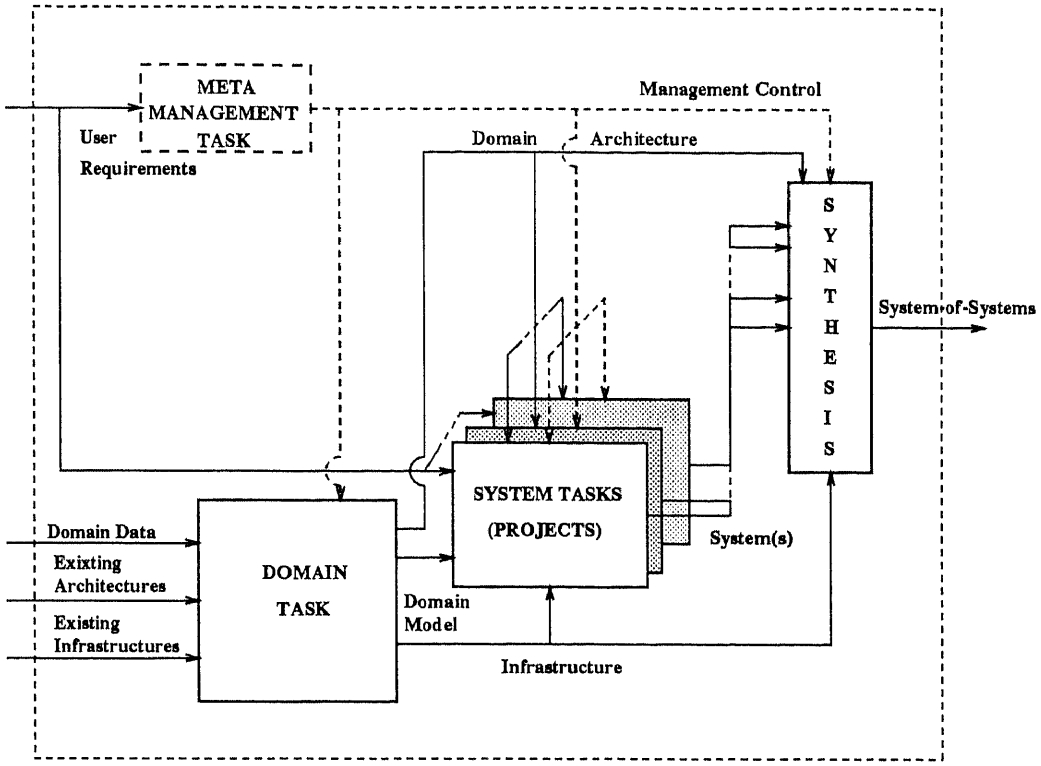


Figure 4.3 The Basic Process Model in GenSIF

communicate knowledge about the domain on the basis of a standardized common model; (2) the domain architecture provides a design reference model used to guide projects during their internal design activities; and (3) the infrastructure provides a standardized set of tools and environments, shared by all projects.

Underlying the model structure, again, is the goal to keep projects as independent as possible but also to coordinate them by the elements of the domain task so as to enable the production of composable systems that can be *packaged* by the SYNTHESIS activity (see Fig. 4.3) into a consistent system of systems.

The process model suggests that domain modeling, domain architecture design, and infrastructure acquisition remain active during the lifetime of the domain and the related systems. By steadily adapting to the changing environment and by reacting to changes in the domain and in the system architecture itself, they keep the framework up-to-date.

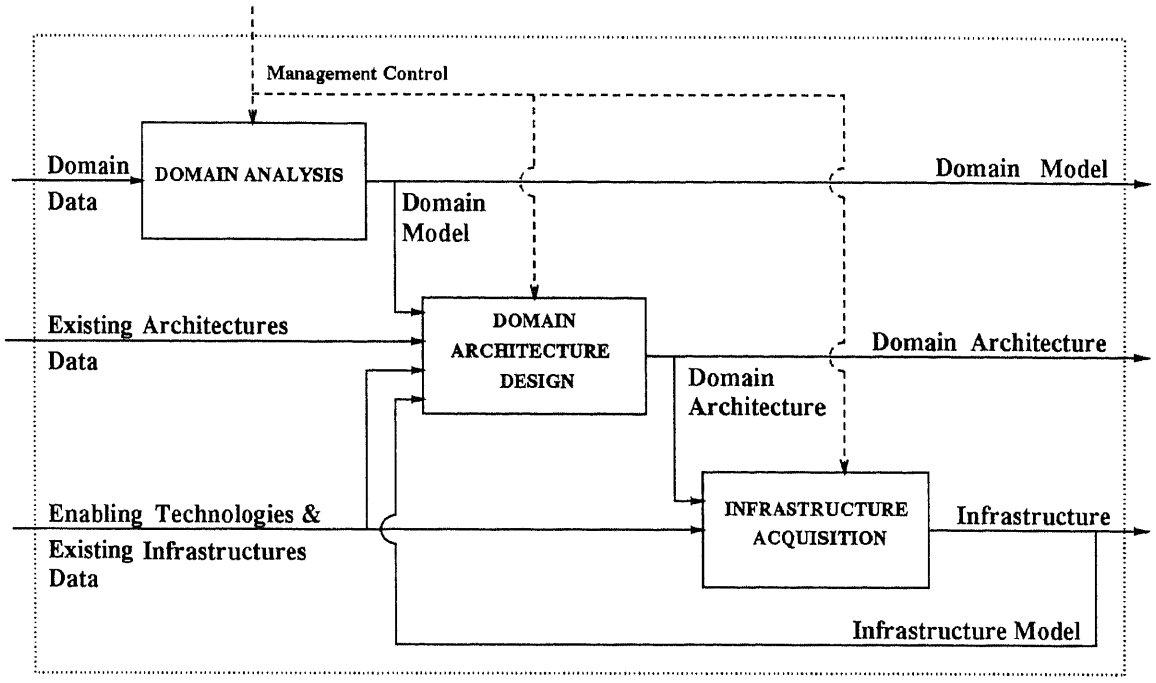


Figure 4.4 The Domain Task in GenSIF

From the project management perspective, all tasks are controlled by a META-MANAGEMENT TASK, Fig.4.3.

The GenSIF process model, together with the underlying two-leveled development concept, constitutes the skeleton of the framework, and are the most significant contribution of the previous work on the GenSIF project. This model also served as a foundation for the research presented in this dissertation which is focused specifically on Domain Architectures and their modeling and representation.

4.3.3.2 Influences of GenSIF Process Model on GARM-ASPECT: The GenSIF process model defines the relationships between the domain architecture and the rest of the development process elements. This has helped:

- to establish the architecture's role, objectives and stakeholders;
- to define the architecture's scope, and

- to derive requirements for the form of its representation, addressed by GARM-ASPECT.

4.4 Example Architectures

In addition to the research investigating the issues of architecture modeling, representation and application, there have been multiple practical efforts, as mentioned in the overview section, focused on defining architectural solutions for specific problems.

Although our objectives do not include definition of architectures, but rather the development of tools to support the definition process, we have examined a number of existing architectures, with the following two objectives: to develop an architectural taxonomy, and to evaluate the feasibility of the modeling abstractions of our representation model.

The following five reference architectures as well as the models for component cooperation, discussed at the end of this section, have influenced our thinking in a considerable way. They all address aspects such as distribution, interoperation, and heterogeneity, which are fundamental to business domain architectures.

4.4.1 ODP-RM: An Architecture Reference Model for Open Distributed Processing

The ISO's initiative for standardization of Open Distributed Processing (ODP) is focused on extending the concept of "openness" from communication to distributed processing, [1, 146, 84]. The result is the definition of a generic framework and the specification of an architecture reference model for open distributed processing (ODP-RM) within which support for distribution, interworking, interoperability and portability is provided. The reference model characterizes the class of open distributed systems whose components, if designed in conformance to the ODP standards can interoperate with other systems, also developed in conformance with the ODP-RM. In this way multi-vendor interoperability of distributed applications

is supported, extending the multi-vendor interoperability of OSI communication software. The model is organized in five parts:

Part 1 provides an overview of ODP.

Part 2, known as the Descriptive Model, introduces concepts for description of distributed processing systems.

Part 3, known as a Prescriptive Model, contains the specification of the required constraints, that qualify distributed processing as open.

Part 4 defines the user's perception of the ODP.

Part 5 examines a set of formal description techniques for their applicability to formalize the concepts in part 2.

Part 2 and 3 constitute the core of the reference model. Part 2 provides a vocabulary, while part 3 defines and constrains the meaning of open distributed processing. Part 3 is structured in five models which correspond to five important viewpoints: enterprise viewpoint (why?), information viewpoint (what?), computational viewpoint (how to do?), engineering viewpoint (how to support?), and technology viewpoint (build or select?). (These viewpoints have been adopted from the ANSA architecture and are discussed in more detail latter.) The overall framework is extended with specifications of a set of components important for the computational and the engineering viewpoints which, for example, include components to support transparency and security mechanisms.

The ODP-RM had many-fold influences on our research and served as a testbench for our approach to architecture modeling and representation. Being a well-documented reference architecture which supports interoperabilty of software applications, ODP-RM made a well-suited example to be studied and generalized. For instance, we have found the idea of common components or services (e.g., a trading service) defined by the architecture very important and reflected it in our architecture model. The vocabulary provided by the architecture, even though

constrained to the area of open distributed systems, has influenced our work on the architectural concepts defined by GARM (see Chapter 5).

4.4.2 Distributed Object Management and CORBA

The Distributed Object Management (DOM) framework, outlined by the Object Management Group (OMG), attempts to define standards for open object interfaces, [103]. The core of this effort is the development of the Common Object Request Broker Architecture (CORBA), [104]. CORBA defines language-neutral, transparent interfaces that permit client object to request services from a server object without requiring the client object to have specific knowledge of the identity or of the location of the server object. The interchange of messages between objects residing across heterogeneous platforms is managed by a CORBA-defined mechanism, referred to as Object Request Broker (ORB). Objects residing on remote and disparate platforms are made transparently accessible to each other through the services of the ORB which supports the implementation of intercommunicating objects distributed across an enterprise. By permitting objects to be language neutral, location transparent, operating system independent, and machine independent, the approach significantly expands the applicability of object technology.

The DOM framework relies on several critical success factors. First, it supports open application architectures. A CORBA-based application will be able to access and utilize objects, services and components that are architecturally diverse and operate on different platforms. The ORBs help to open the applications up for shared access within an enterprise, while CORBA provides the conceptual "glue" that binds disparate applications together. In result, organizations can gain profits from a community of cooperating objects distributed across the business (domain). Second, DOM technology supports open application interfaces. By maintaining a language-neutral interface, DOM permits the connection of computing functions

coded in different languages. The ORB provides a common media through which messages from dissimilar object-environments are transformed into the receiving object's native language context. The language used to construct the server object is transparent to the client object when using the CORBA standard interface definition language (IDL). This language-neutral interface capabilities should, in time, stimulate the emergence of "commodity object" products in the market place, which in turn will reduce the boundaries between in-house applications and vendor products. Third, the ORB technology provides access and location transparency to the objects distributed across the network of the enterprise. It insulates the developers from the need to code low-level networking and allows them to concentrate on providing solutions to business problems, increasing the quality and maximizing the return on development investment. Fourth, redundant data and functions can also be greatly reduced as core objects representing common services and facilities (e.g., security, printing, etc.) as well as functions shared in the business domain (like "customer loan" in a banking domain) could be built once and made reusable and accessible across the distributed object environment. This will further simplify and lower the cost of the development, increase the return on investment, and enhance quality by reusing "veteran" objects that have been already developed, debugged and tested.

The CORBA standard marks a "step" in object-oriented technology, [103]. It provides a conceptual basis for interconnecting decentralized business units into an enterprise-wide network of objects. By enabling objects to interact across heterogeneous platforms within a distributed processing environment, it presents a great opportunity for maximizing the computing potential of every business domain by providing solutions for problems such as closed application architectures and incompatibilities between heterogeneous computing environments.

4.4.3 ANSA the Architecture

The Advanced Networked System Architecture (ANSA), [10], is an architectural framework for ODP. In our vocabulary, ANSA is a generic reference architecture which supports the design, construction, deployment, and maintenance of open distributed applications. The process of defining the architecture has been documented by the team and is valuable by itself. To derive the architecture, the ANSA team has studied current practice and research in distributed computing and system design techniques. The crucial concerns that make up "distributed processing" have been identified as corresponding to five dominant viewpoints. Subsequently, the description of ANSA has been structured as a set of projections of the architecture onto models representing these five view points. The projections that make up the architecture are as follow:

1. Enterprise projection – the purpose of this projection is to explain and justify the role of a computer system within an organization. An enterprise model describes the overall objectives of a system – "What is a system to do and who is it doing it for?".

2. Information projection – the purpose of the information projection is the identification and location of information, and the description of information processing activities. An information model describes the structure, flow, interpretation, value, timeliness and consistency of information held within a system.

3. Computational projection – the purpose of the computational projection of ANSA is to demonstrate the organization of a distributed system as a set of linked application programs. A computational model is a framework for describing the structure, specification and execution of programs with a special attention paid to the constraints that arise directly or indirectly from

distribution. The ANSA computational model defines the facilities required in a programming system which permit the implementation of distributed applications for an ANSA infrastructure (e.g., an infrastructure conforming to the ANSA engineering model). The model introduces a set of fundamental concepts, e.g., client, server, interface, and action. It is structured in two parts – an interaction model that defines permitted forms of interactions and a construction model that defines the elements from which the interacting objects are constructed.

4. Engineering projection – the purpose of engineering projection is to make extra-functional aspects of the distributed system explicit. An engineering model is used by a designer to make decisions that concern trade-offs between quality attributes such as performance, dependability, and scaling.

5. Technology projection – its purpose is to describe the physical components, that make up the distributed system, so they can be related to the architectural concepts in the other projections. The technology models serve as blueprints of systems during their construction and maintenance, showing how architectural concepts have been implemented in practice.

ANSA, the architecture, has been implemented as an infrastructure – ANSAware, [11]. ANSAware is a development and run-time environment which supports and enforces the ANSA architectural principles in development of distributed applications. As our experience from several case studies shows, this infrastructure has made the application of the architecture well-understood and easy to follow process.

The work on ANSA have served as a foundation for the ODP standardization initiative of ISO, discussed earlier in this section.

4.4.4 OSCA Architecture

The $OSCA^{TM}$ architecture, as described in [4, 14], is an implementation-independent system design framework intended to give Bellcore Client Companies the flexibility to combine software products in ways which best satisfy their business needs and to provide access to corporate data by all authorized users. It promotes interoperability between heterogeneous software products residing on a variety of computing systems, which if developed within the OSCA framework, will behave as a cooperating whole in a loosely coupled, distributed configuration. It also supports the "operability" of software products that consist of large numbers of programs, transactions, and data bases in order to meet the performance, availability, reliability, and security objectives of the business. Fundamental to the OSCA architecture (in achieving the above goals) is the notion of separation of concerns, where the overall systems functionality is organized into layers or categories which include corporate data management functionality (data layer), business operations and management functionality (processing layer) and human interaction functionality (user layer), so the functions of one layer are de-coupled from the functions of another layer. The software which implements the functionality of the layers is partitioned into "building blocks" which must adhere to specific building block principles – a set of structural and operational constraints. In addition, interfaces provided by the building blocks for other building blocks must meet certain well-defined criteria in order to support interoperability. This elegant structure and its high-level of abstraction make the OSCA architecture easy to understand and, one should expect, to apply. However, the lack of an infrastructure that implements, supports and to acertain extent enforces the architecture principles; and the lack of rigorous and easy-to-manipulate architectural specifications (instead a huge volume of textual descriptions is provided), as well as somewhat weakly defined interaction model have reduced OSCA's value as an architecture directly applicable in a development

process. This, however, does not reduce the OSCA architecture's value as a research experience and as a reference architecture that can be refined, customized and then applied in the development of complex integrated systems.

4.4.5 DORIS

The Data Oriented Requirements Implementation Scheme (DORIS) is an architecture-based framework for development of complex, embedded, real-time, distributed, multiprocessor systems, [143, 142]. The term architecture, in the context of DORIS, is used in very general sense to indicate any well defined form of system description in terms of components, connections, and composite structures.

DORIS has been developed at British Aerospace, Stevenage, (BAe) for use in high performance guided weapon products. The primary objective for its development has been to provide a means of handling large-scale system development efforts (as prime contractor) with the following facets to this: (1) the need to be responsive to the customer in terms of change requests and associated schedule, cost and technical implications; (2) the need to manage the work done by sub-contractors (often for major subsystems); and (3) the need to keep track of the current status of the system and to be able to perform "trade-off" and "what-if" analyses at any stage from system definition through into operational deployment.

The fundamental approach, adopted in DORIS, is based on a layered set of coherent system representations which cover the system development lifecycle in a consistent traceable way, that preserves the integrity of the concepts throughout the entire development. There are three central elements:

1. Functional definition, presented as a network of processing functions, information retention functions, and communications, any of which may contain any of the others.

2. Design – which produces a "real-time network," [141], and lower levels detailed designs.

3. Implementation – which covers the integration and the commissioning of the target systems. At this level DORIS defines a set of principles for guiding the implementation of the real-time network, such as the following example "... minimization of dynamic resource scheduling that, in turn, favors the use of multi-processor configurations to reduce process contention, and distributed shared memory to eliminate memory access contention," [143] .

DORIS, which takes its name from the strong and explicit representation of data at all levels and stages of the development process (more properly, shared information at the functional level and shared memory at the implementation level) provides a detailed specification of the interaction model (a set of connectors, or protocols), decomposition principles, and the use of hierarchies. It is a rigorously defined, multi-layered architecture providing support to an established application domain. DORIS has the merits of being well documented and successfully applied in many development projects and provides valuable experiences justifying the architectural concepts.

4.4.6 Architectural Models for Component Cooperation: The Intelligent Agents Approach

Underlying any integration framework is the issue of component cooperation. The general problem of cooperation (including the identification of architectures conducive to cooperation) is given a substantial attention by the intelligent agents research community, [56, 55].

Genesereth and Ketchpel, in [56], define agent-based software engineering as a methodology for creation of software capable of interoperating in the presence of heterogeneity (different languages, times, developers and interfaces) and dynamics

(additions, rewritings, removals of components) in the software environment. Applications in this approach are written as "software agents" – software components that communicate with their peers by exchanging messages in an "expressive agent communication language". The size and complexity of an agent is not restricted – they can be as simple as subroutines, but typically they are larger entities with some form of persistent control.

The architectural problem in the context of intelligent agents cooperation is, typically, addressed form an operational point view, as opposed to the structural one. In [56], two very different approaches for agent interaction are discussed: direct communication, in which agents handle their own coordination (in a distributed fashion), and assisted communication, in which agents rely on special common services (system programs) to achieve coordination. Further, two architectures (systems organizations) for direct communication are presented: "contract-net" and "specification sharing". In the contract-net approach to interoperation, agents in need of services distribute requests for proposals to other agents. The recipients of these messages evaluate those requests and submit bids to the originating agents. The originators use these bids to decide which agents to task, and then award contracts to those agents. In specification sharing, agents supply other agents with information about their capabilities and needs. The agents then use this information to coordinate their activities. Two basic disadvantages of the direct communication approach are noticed – cost and implementation complexity.

In order to eliminate these disadvantages, a "federation" is suggested as an alternative for agent organization. In this organization, agents do not communicate directly to each other. They interact only with system programs called facilitators (or more generally, mediators, [155]) while these programs communicate between themselves to satisfy the agents' needs. Agents document their abilities and needs with their local facilitators. In addition, they send to and receive from the local

facilitators application level information and requests. The facilitators transform and route the necessary messages to the appropriate locations. In effect, agents form a "federation" in which they surrender their autonomy to their facilitators, and the facilitators take the responsibility to fulfilling their needs.

The concept of using common services to support interoperation is well known. Popular examples include directory services, distributed object managers, subscribe capabilities, and automatic brokers. The primary difference between these approaches and agent-based software engineering is in the sophistication of the processing done by the facilitators, [155].

Most of the architectures we have examined in our studies, and especially the ones we have selected for the above discussion, have had an impact on our understanding of the architectural issues. They have contributed to the identification of a set of important architectural aspects and have supported our work on the architecture reference model, presented in the next chapter.

CHAPTER 5

ARCHITECTURE TOOLBOX

In this chapter we present our approach to architecture modeling and representation. The chapter resembles a toolbox of modeling concepts and notations for expressing architectures of software systems. The balance of the chapter is as follows:

- Section 5.1 *GARM: A Generic Architecture Reference Model.* In this section we define the software architecture model (GARM) underlying our approach to architecture engineering and representation (GARM-ASPECT). We discuss the modeling abstractions defined by the model and its implications for the architectural process.

- Section 5.2. *ASPECT: An Architecture Specification Technique.* This section is an introduction to ASPECT – a generic architecture description language, designed for expressing architectural structure of software systems. It permits the description of reference architectures and architectural instances. We define the elements of the language and illustrate them with examples.

5.1 GARM: A Generic Architecture Reference Model

Underlying any discussion of software architecture is a model of what software architecture is. The model has implications for both the architectures themselves as well as for the architecture engineering processes and its understanding is fundamental to the proper treatment of the various issues of architecture description, analysis and application.

The Generic Architecture Reference Model (GARM) is a set of architecture modeling abstractions. Even though our main focus is on reference architectures and, especially, on domain architectures, the model is general and can be applied to reference architectures as well as to individual system architectures.

GARM defines a set of modeling elements, their relationships, respective semantics, and general rules of how to combine them into architecture specifications. The following are the basic elements of GARM (see Figure 5.1):

- **concepts,**
- **rules,**
- **patterns and**
- **guidelines.**

Concepts and rules are fundamental to the description of any reference architecture. They are the modeling primitives, the goals, and the constraints in terms of which an architecture is described. Patterns and guidelines, in turn, are helpful extensions, annotating the core description with verified solutions, design rationale, and recommendations, typically collected as feedback from applying an architecture.

GARM supports the concept of multiple architectural views which reflect system aspects such as structure, behavior and security. It recommends the definition of aspect-specific properties and constraints (rules) that support them. It also defines multiple views of constructive elements: components and connectors which will be discussed next.

5.1.1 Concepts

Concepts establish an architectural vocabulary. There are two groups of fundamentally different concepts addressed in GARM:

- *property-related concepts* in terms of which the objectives and the constraints for/of the architecture can be expressed. They represent the "basic concerns" of an architecture: a set of extra-functional – quality-related, structural and operational – requirements that have to be met by software systems developed according to the architecture.

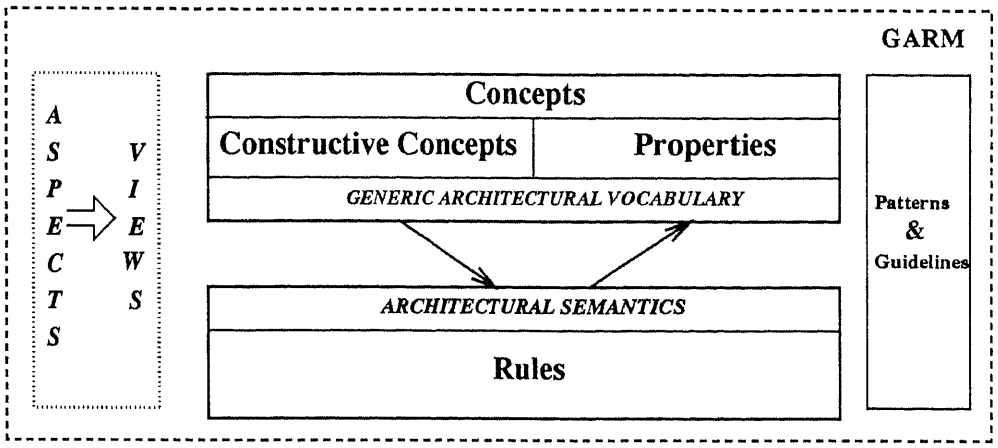


Figure 5.1 The Generic Architecture Reference Model

- *constructive architectural concepts* in terms of which the architectural solutions are expressed. These are high-level design elements that constitute the "building material" available to the architects when describing the architecture of software systems.

5.1.1.1 Property-related Architectural Concepts: Reference architectures define properties and prescribe design models that support them. The architectural descriptions comprise definitions of properties which are "goal statements" that initiate, direct, and constrain the design process. Typically, these concepts relate to so-called extra-functional, or *quality aspects of software systems* such as security, performance, and reliability. For instance, a reference architecture, which addresses the security aspect of software systems in a domain, may include definitions of security-related concepts (as well as constructive design specifications that implement them) like access control and authorization. A reference architecture description may also include definitions of reliability and fault-tolerance-related concepts like error, failure and fault. Property-related concepts also define *structural and operational goals*, related to the way software systems are organized and function. Examples

include transparency (access, location, replication, concurrency, migration, etc.), autonomy, and scalability.

Property-related requirements and objectives affect architectural solutions and can be enforced in a number of different ways. Certain properties can be addressed by constraining the structure and the behavior of the design elements, e.g., components and their interactions. Such constraints, as it will be discussed later, can be captured as architectural rules. Assume for example, that an architecture supports "secure interactions", defined in terms of access control to a certain type of component (e.g., DLBB in OSCA architecture, see Chapter 4). The description of such an architecture will then contain an architectural rule that constrains the behavior of this type of component and its environment (the other components participating in the interaction) in a way that enforces this architectural property. Such a rule would state informally that: "It must always hold that DLBB building blocks check the access privileges of the service requester before servicing a request." This rule obligates the DLBB architect, designer and developer to develop access control procedures, for example, to provide support for ACLs (access control lists). It also implies that any component, requesting a service, has to submit an identification parameter with every service request placed to a component of type DLBB.

Supporting certain properties in an architecture can also lead to the identification of "specialized" design elements. The services provided by these elements (called architectural services, [11], or common functions, [1]) are independent from the specifics of the problem domain in which the target software systems are developed. Such components are defined as part of a conceptual architecture, and are typically implemented as major elements of the infrastructure supporting this architecture, [76]. Consider, for example, the *location transparency property.* A typical architectural solution would be to define a "directory service" and to allocate this service to a "specialized" component such as the "Trader" component defined

in ODP-RM, [1]. The functions and behavior of this component will then be defined in the conceptual architecture view of the overall reference architecture.

Certain properties can also affect the entire system organization and the selection of interaction models. This, for example, is often the case with the overall system performance.

Conforming to an architecture which supports a set of properties helps to provide guarantees that an actual software system developed according to the architecture (or an instance of it) will exhibit these properties, given that the architecture is not violated throughout the rest of the development process. Of course, it is the responsibility of the designers and developers to verify that the transformations are correct and that the implemented system meets its architectural specifications.

5.1.1.2 Constructive Concepts: Constructive concepts or architectural elements are means for modeling the overall organization of software systems. The following are the main constructive elements of GARM:

- two basic modeling concepts:

  (1) Component[1] and

  (2) Connector, and

- an organizational concept that wires components and connectors together:

  (3) Configuration (static and dynamic).

Components:

Architectural components denote objects with independent existence, such as systems (of a system of systems), modules, processes, and procedures[2]. Provided our focus on system integration, in the context of GARM, components should be broadly

[1]Component–connector model is commonly applied for describing structural aspect of software architectures. For details, see Chapter 4.

[2]A list of architectural component types can be found in [29].

interpreted as large-scale components, subsystems or systems. Components represent the nuclei of computation and allow designers to aggregate functions, data and state into higher order units. From a dynamic perspective, components represent the sources of activity in the system and form the designer's mental model of a system in operation. The focus of architectural designs is on externally observable properties of components separated from the technology and algorithms of their implementations. Implementation details are suppressed in architectural descriptions and, therefore, different component implementations are indistinguishable from an architectural perspective (at every level of system decomposition).

Architectural components in GARM are modeling concepts with multiple aspects, namely function, behavior, data, quality, structure, and type.

The Functional Aspect

From a *functional perspective* architectural components represent information handling functions with signatures describing their interfaces. In GARM, a distinction is made between two basic types of functions:

- *application-specific functions* and
- *common or architectural functions,* [1].

The difference between application-specific and common functions is a pragmatic one. In general, the separation corresponds to the two levels of architectural thinking (conceptual and application), discussed in Chapter 3. Functions that characterize the component as part of a problem-specific solution are defined as application-specific, while functions that support the operation of the system itself are defined as common functions (see also [1]). Common functions may depend on other common functions. Application-specific functions may depend on common functions and other application-specific functions to provide required functionality. Components that offer only common functions are fully described in the conceptual architecture, and

are, typically, implemented as part of the infrastructure. Components that offer application-specific functions are only templates in the conceptual model and will be further defined in the application architecture, where application-specific functions will be assigned to them (see also Chapter 3).

As an example, let us return to the Trader component, discussed in the previous section. A typical architectural function or service provided by this component is an "Interface Reference Trading" service, [1]. The Trader, as we have seen, is an "implementation" of the location transparency property. It offers only common services and as such it is fully defined in the conceptual architecture and consequently can be implemented as part of an infrastructure that supports the particular architecture, for example, ANSA (the architecture) and ANSAware (an infrastructure) that implements and supports the architecture, [11]. Remember that extra-functional properties (reliability, safety, security, etc.) often lead to the identification of common functions that are typically implemented as components being parts of an infrastructure, [9].

As another example consider a "Calculate Quota" function defined in the context of an insurance domain. It is a typical problem-specific or application-specific function which have to be provided by a component that is a part of an Insurance application. This component can be defined only in the context of an application architecture and will, typically, be derived from a component type of the underlying conceptual architecture, [69].

The Behavioral Aspect

As a part of a system in operation, components are interacting entities, identified by the roles they play as part of the overall system. During this process components exhibit certain characteristic pattern of *observable behavior.* A component's observable behavior is defined as a set of observable (from the environment) actions,

as a reaction to the actions of the environment or initiated by the component itself. (Environment refers to the other components interacting with the component.)

The component's behavior is observed at certain access or interaction points, modeled as *ports*. A port represents an abstract "location in time and space" at which interactions between the component and its environment occur. Any subset of (functionally) related ports is captured in an abstraction, called *interface*. Taken together, the set of all interfaces related to a component account for all interactions the component may participate in. Apart from expressing functional and behavioral aspects of components, interfaces capture their data aspect as well.

The Data Aspect

In order to reflect the data aspects of software systems, architectural components describe externally observable information models, data structures and types as part of their interface specifications. Typically, a data-type-model (for external data representation) is defined in the context of a conceptual architecture (as an aspect of the interface types defined in this architecture). Describing the externally observable organization of the data encapsulated in a component is also an important aspect, especially when a data-capsule type of components, such as directory component (ISO X.500) or an OSCA's DLLB, have to be described in architectural designs.

The Quality Aspect

In GARM, extra-functional properties of software systems are considered as fundamental architectural concerns and their representation is made an important aspect of architectural descriptions. Architectural components are characterized by *quality attributes*, reflecting the extra-functional properties of software entities and, in their interface specifications, quality attributes are as important as functional and data-oriented elements. For example, qualities, such as response time, rate of event release [67], and level of security, observed at a particular port, determine the quality attributes of the port and have to be reflected in its overall description.

The Structural Aspect

Architectural components model complex entities, such as software systems. They are regarded as possibly composite entities with an internal structure. Therefore, in GARM, two types of components are defined: (1) atomic components, called *building blocks* and (2) *clusters* encapsulating other components. Composition-decomposition can be applied recursively to architectural components.

Composition (aggregation) is a powerful modeling concept that permits an organization of interdependent components to be regarded as a new, higher-order component. The characteristics of the composite component are determined by the characteristics of the components that are combined and the way they are combined, i.e., the types of connectors between them (see below). Thus, complexity can be reduced by modeling sub-systems or systems as architectural components.

At every level of architectural description, components are represented by their interfaces. Their internal organization is encapsulated and isolated and, therefore of no consequence, as long as the rules of interaction and structuring are observed on the interface level. This, for instance, allows different architectural styles to be employed within different components, provided that all of the lower-level components are aggregated and encapsulated in such a way that they form valid components on the next higher level. In this way heterogeneous architectural styles can be combined into a coherent, higher-level architectural model.

Decomposition is also a well known design tool, routinely used in techniques such as Data Flow Diagrams, Structured Analysis and Design Technique, and Object Oriented Analysis and Design. It permits a complex component (e.g., a model of a system) to be decomposed into a number of simpler entities. Obviously, there are limits beyond which it is not practical to decompose a component to more primitive components. These limits are set by the nature of the problem and the level of granularity of the design (systems, modules, processes). To model the elements at

the lowest level of granularity, in GARM, the building block concept is introduced as an atomic architectural component.

The application of composition-decomposition abstraction leads to hierarchical architectural descriptions characterized by "PART-OF" relations.

The Type Aspect

Components in a reference architecture come in a variety of types and specializations, as determined by the problem, system aspects addressed, architectural style or the level of granularity of architectural designs. Examples for types of components are: specialized building blocks in a system of systems - GenSIF [121], objects - CORBA [152], guardians - Argus [82], filters - Pipe and filter [53], etc. In general, types are components with the same basic layout but different capabilities and specialties. A component (or any element) type defines the structure and properties that a component (element) instance must posses in order to satisfy the type.

The identification of component types can be based on different principles and guided by different objectives. For example, the principle of separation of concerns has guided the identification of three types of building blocks in the OSCA architecture [4]: user interface building blocks, specializing in handling the user-interface functions; process layer building blocks, responsible for business-aware processing; and data layer building blocks, encapsulating and stewarding the data repositories.

Reflecting this property, our model views architectural components as typed entities, where classification hierarchies can be used to relate them and to organize architectural descriptions. The foundation of this type of hierarchy is the generalization-specialization abstraction (or so-called "IS-A" type of relationship) and a refinement process, which is discussed in detail in Chapter 6.

Thus, our architecture model implies that both "PART-OF" (discussed earlier) and "IS-A" relationship types have to be supported by an architecture description technique, and that a reference architecture description will include hierarchical

descriptions. (It is, however, very important to appreciate that these hierarchies are entirely orthogonal.)

Connectors:

A software system is a collection of interacting components. The interactions are the glue that binds the parts of a system in operation together into an overall system and deserve special attention in architectural designs, [132]. In GARM components' interactions are explicitly modeled as *connectors*. A connector is a modeling abstraction that represents functions and structures for data and control interchange in a software system.

Connectors are abstract channels relating architectural components. Consider, for instance, the following well-known types of relations between pairs of interaction points: (1) client - server, where components using a service are called users or clients and the components providing a service are providers or servers; and (2) producer-consumer – a different interaction model, where the components acting as a source of a data flow are called producers while the components acting as a sink for the data are called consumers. Notice that components participating in an interaction play specific *roles* (e.g., a producer and a consumer) which are synchronized by a set of rules underlying the interaction model, referred to as interaction *protocol*. Thus, like components, connectors have interfaces, that are defined by a set of roles. Each role identifies a type of participant (component port) of the interaction modeled by the connector. Binary connectors have two roles, such as caller and definer of an RPC connector, the sender and receiver of a message passing connector or, for example, the abstract roles discussed above. Other kind of connectors may have more than two roles. For example, an event-broadcast connector instance might have a single event-producer role and multiple event-receiver roles. Also, one may define a

monitored-pipe connector with three different type of roles: a source, a sink and a recorder roles.

Therefore, a notation for an architectural connector has to provide means:

- for expressing the obligations of any of the participating ports, describing their roles,
- for reflecting the obligations of the roles in regard to each other, e.g., ordering, synchronization, etc., hence, describing the interaction protocol, and finally
- for capturing the properties of the interaction channel, such as delays or possible structure.

In GARM, connectors are first class entities, i.e., they are named, typed, and refinable. They are further characterized by the same aspects as architectural components. The type and classification aspect is, in principal, identical to what has been presented for architectural components. Therefore, it will not be discussed any further. The rest of the aspects – functional, data, behavioral, quality, and structure for connectors and components vary to a certain degree and are addressed below.

The Functional Aspect

From a *functional perspective*, a distinction can be made between a "data-carrying" aspect of a connector, modeling the flow of data between the interacting components and a "control-passing" aspect, reflecting the flow of control in the system. For example, the producer-consumer relationship, discussed above, lends itself to a "data-carrying" type of connector, while a client-server relation implies a connector that demonstrates both aspects: "data-carrying" – the parameters and the results passed, and "control-passing" – a client triggers the execution of a requested service.

The Data Aspect

The *data aspect* of a connector is expressed in terms of data structures and types defined in the context of a connector. For instance the buffer of the "buffer-type"

connector defined in DORIS architecture, [143], may have a well-defined structure; or the pipes of a Pipe-&-Filter style-based architectural design may be defined to be of type character-stream, [3]. Data cannot be transmitted over a connector unless their types are compatible with the types defined in the context of this connector.

The Behavioral Aspect

A connector's *behavior* is defined and restricted by the behavior of the ports (hence components) participating in the interaction. The roles part of a connector description prescribes the expected behavior of the corresponding ports participating in the interaction. The protocol of the connector describes the "process of interaction", imposing order on the observable actions of the participating ports.

The Quality Aspect

The quality aspect of connectors is concerned with the *quality of interaction.* It addresses the quality of the collective operation of the ports participating in an interaction, thus it encompasses the quality attributes of participating ports (prescribed by the roles) and the quality of the interaction channels. For simplicity, however, we have assumed that the quality of interaction depends only on the quality attributes of the participating ports, hence the interaction itself is "perfect" – delays or communication faults, for example, are not considered in the current GARM model.

The Structural Aspect

From a structural point of view in GARM a distinction is made between primitive and composite connectors. Hierarchical decomposition is applicable to complex connectors which may be decomposed to connectors and components, with the restriction that the involved components support only common functions (see the discussion on components). For example, a high-level, abstract, composite connector that describes the interactions between two intelligent agents in a "federated architecture" [155, 56], may be decomposed into simpler connectors and a component, "facilitator", which offers only common functions. Respectively, composition is also

applicable to connectors in a way that several components and connectors, "implementing" an interaction, can be combined into a composite connector. This reduces the complexity of architectural designs by hiding complex interaction processes into a higher-level abstract connectors.

From a structural perspective we also distinguish between connectors with different "topologies", such as binary connectors (port-to-port type of connectors) and connectors with a higher-order of connectivity (analogous to "broadcasting" or "multicasting" structures in networking). An RPC-based client-server type of connector is a typical example of a binary connector, while an "event-bus" connector defined in the context of an "event-bus architecture," [48], has a higher order of connectivity.

Architectural Configurations:

Architectural configurations define associations of components and connectors. They name the result of a set of components and connectors wired together and, thus, provide a basis for addressing global properties, relevant to the overall system, such as end-to-end behavior, and global control and synchronization.

A configuration description goes beyond the description of individual components and connectors, and represents a collection of components, structured by the identification of their "bound interfaces." Configurations are "snapshots" of the structure of the collection, as it should be observed during system operation. They may represent static structures invariant of system execution. Alternatively, they may represent dynamic structures, established by dynamic mechanisms such as object binding and unbinding in a distributed system, [1]. Finally, any configuration aggregates participating components and connectors into an instance of an equivalent composite component or architecture.

Thus, any configuration is an expression over the basic (primitive) constructive architectural concepts. The form of the expression is defined by a set of constraints, referred to as architectural rules and discussed in the next section.

In order to create a basis for more expressive architectural specifications, in the future, GARM may be extended to support additional organizational concepts, such as groups and domains. (This will make the description of architectures, such as ODP-RM, [1], which support them more natural.)

Components, connectors, and configurations are the constructive concepts for representing architectures of software systems on one layer of abstraction. In order to handle more complicated architectures, as discussed above, generalization-specialization and aggregation-decomposition abstractions can be employed, building layers of interrelated architectural designs.

5.1.2 Rules

Architectural rules are a mechanism introduced in GARM for gluing together property-related and constructive architectural concepts. Rules specify design decisions made in order to support a set of architectural properties which reflect technology constraints and domain-specific requirements. They define constraints on constructive architectural elements: components, connectors, their constituents as well as their static and dynamic (structural and operational) compositions, identified as configurations (see Fig. 5.2). A constraint system will allow architects to express additional information about the constructive element types and to increase the precision of their descriptions. For instance, a component type may be defined which have a specific type of interface; then a rule can be added to explicitly define the allowed range of replication of the interface in component instances.

Rules are the basic mechanism employed for representing the configuration concept at the conceptual architecture level. By constraining the allowed config-

urations and their operational meanings, the rules of a conceptual architecture prescribe the structure and the operation of software systems developed in compliance with the architecture without assuming any specific application or problem domain.

On the application architecture level components are further functionally differentiated as problem-specific functions are allocated to the component types of the conceptual architecture. This leads to explicit structures and hierarchies, built in compliance with the rules of the conceptual architecture and described as part of the application architecture.

With respect to the relationship between an architecture description and its application, rules are assertions that must be demonstrated by the user of the architecture to hold of the implementation. They constitute an invariant of the development process architectural order[3]. Some rules may be enforced by tool checking.

To provide some intuition behind the concept of architectural rules, let us consider as an example a toy "layered system organization" *(LS)*, [97]. A conceptual architecture reflecting such a system organization may include the following (informally described) rules:

> *R1: Components of (LS) are grouped into layers – L1, L2, ..., Ln.*
> *R2: The components that belong to a given layer can directly interoperate only with components from neighboring layers and their own layer.*

An application architecture will translate these rules into explicit configurations of connector and component instances, exchanging control and data. Consider, for example, the layered organization of the OSI Networking Reference Model ([146]). Knowing the exact services provided by the individual protocol-layers (components) the precise interfaces between the layers can be specified following the rules of the conceptual architecture.

[3]In Webster Dictionary, [93], "architectural order" is defined as: a. any of the five classical styles of architecture (Doric, Ionic, Corinthian, Tuscan, and Composite) based on the proportions of columns, amount of decoration, etc.; b. any style or mode of architecture subject to uniform established proportions. Here, we use the term to refer to established structural and behavioral principles as well as quality attributes.

As discussed above, rules in GARM are a means for capturing of architectural constraints and are referred to as *Constraint Rules.* As part of an architecture description they are axioms that specify restrictions about the structure and behavior of software systems developed according to this architecture. The Constraint Rules are further subdivided into two categories: *Structure Constraint Rules* and *Behavior Constraint Rules.*

5.1.2.1 Structure Constraint Rules: Structure Constraint Rules (SCR) specify policies or conditions about the organization of element types and their associations. They are invariant throughout system operation, as they must hold under any operational circumstances.

We group SCR into the following three categories: *element-structure-constraint-rules* which deal with the structure of architectural elements, components and connectors as well as their constituent elements; *configuration-constraint-rules* which prescribe the allowed compositions of components and connectors on one level; and *layered-structure-constraint-rules* which constrain hierarchical organizations.

Consider the following informally defined SCR examples:

1. An example element structure constraint rule:

   - A rule restricting the organization of a component element type in MyPipe-&-Filter architecture:
     R: *MyFilter component type may have one and only one output port.*

2. Example configuration constraint rules:

   - A rule restricting the allowed associations between selected types of components and connectors in an OSCA-based architecture:
     R: *(1) Interface Building Blocks and Processing Billing Blocks interact by message-passing; (2) Processing Building Blocks and Data-handling*

Building Blocks interact by message-passing; (3) No direct interactions (message passing) are allowed between Interface Building Blocks and Data-handling Building Blocks. (The component types and the connector types are described separately from this rule which only specifies the allowed associations between them.)

- Event-bus architecture – an example configuration constraint rule:
  R: *It must always hold that the event bus connects only to components that announce or receive events, [48].*
- Event-bus architecture – a dynamic configuration constraint rule example:
  R: *An announced event can be received by zero or more other components (unlike, for example, the typical rule in a pipe-&-filter architecture, where the written data can only be read by one other component – the sink).*

3. Layered structure constraint rules can, for example, be used to restrict the use of multiple inheritance in an object oriented system or to ban the aggregation of certain components into clusters because of reusability constraints or performance reasons.

5.1.2.2 Behavior Constraint Rules: Behavior Constraint Rules (BCR) are policy or conditions that constrain the behavior of constructive elements, and are defined in the context of an action or interaction[4]. We distinguish between two types of behavior constraint rules: *action constraint rules* and *stimulus-response rules.*

Action Constraint Rules (ACR) – specify conditions that must hold before and after an action (operation) to ensure its correctness. ACRs can be defined as

[4]Structure, behavior and quality are three basic views of any architecture. In order to provide a comprehensive description of an architecture, all three views have to be addressed. As our representation approach is focused primarily on the structural view and only provides pointers to descriptions of behavior, the BCRs are addressed only in this overview of GARM.

pre- and post-conditions or obligations ([106]) guarding the behavior observable at an access point (port). ACRs can be used to specify dynamic dependability constraints between the elements of an interface.

As an example, consider the following informally defined rule constraining the behavior of a filter component (pipes-&-filters architecture),

R: *Data can be written on a filter's output port only after data have been read on its input port.*

And while this rule is general, defined in the context of a conceptual architecture and applicable to any filter component type, ACRs can also be used to capture the behavior of component instances. The following rule prescribes the behavior of a "Trader" component [1],

R: *A request for an interface reference submitted to a Trader component will be satisfied only if the service-offer has been already registered with the Trader's offer space. (A precondition, such as* OfferExists, *restricts the outcome of the operation.)*

Stimulus-Response Rules (SRR) – constrain behavior and define obligations in the context of an observable event (such as *service request*), triggered by component's environment. They affect both the behavior of components and connectors, and de facto define constraints on interaction protocols. SRRs can be defined at different levels of abstraction, as part of a conceptual architecture, application architecture or specific system architecture as well as at different levels of generality. They can, for instance, be used to constrain the order in which messages between components are sent. (For example, an OPEN or INITIATE call may have to precede a READ call to a component port.) Such rules have to be documented as constraints on component interface, [157]. SRRs, for example, can also be used to establish system-wide security mechanisms by specifying constraints on the interactions between data-capsule type of components and their environment,

R: *In order to use a service provided by a data capsule, the service requester has to*

submit an access-permission parameter which is to be compared with the entries of an ACL supported by the data capsule. If access is "granted" the request is serviced, and the result is returned to the requester; if not the requester is informed about the reason for denying the request.

Another example of this group of rules is the following less general rule defined in the scope of a specific component,

R: *When an interface reference is requested from the Trader (ODP) and if the requested service matches an offer in the trader offer space and if the "authentication parameter", submitted by the requester, matches the one required by the offered interface reference, then the interface reference is passed to the requester, else a negative response is returned.*

Rules, similar to the examples above, are often used in requirements specification, [126]. During the process of architecture definition the requirements rules have to be allocated to elements of the architecture.

There is an important issue involved with the evaluation of rules, namely their scope. The scope of a rule is said to be the constructive element type (e.g., component, connector or architecture) for which it is defined. Thus, the rules of an architecture are design rule predicates that must hold on any element that is of the type for which the rule(s) is defined.

Rules form an "architectural law[5]" that is explicit and strictly enforced throughout the development process. This law provides some a priori knowledge about the structure and the behavior of the system, that makes the system easier to understand, safer to implement, and simpler to manage. They also create a consistent reference for maintenance and controlled evolution.

[5]A similar concept, referred to as "system law", is used in [97]. The focus in this work, however, is primarily on component interactions where the only underlying interaction model is message-passing, while the scope of our rules is broader. The "system law," on the other hand, incorporates aspects of system evolution and development process which are not part of architectural law as presented in this thesis.

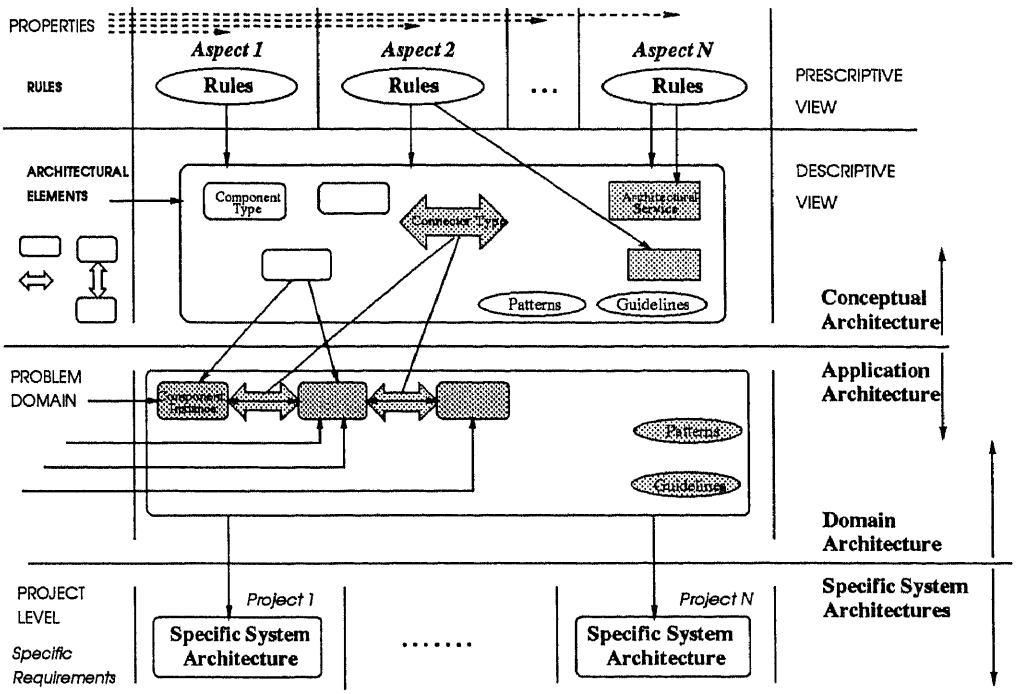


Figure 5.2 The Generic Architecture Reference Model

A system which violates an architectural law, hence whose implementation does not comply the architecture, may still operate as required and meet its obligations, but could be more difficult to enhance or modify than one which did not; or it may not work at all. (In this context, Perry and Wolf , [108], discuss two architectural problems: architectural erosion and architectural drift, due to violations of or insensitivity to the architecture.)

As portrayed in Fig 5.2, rules reflect required system properties and define constraints on architecture-specific constructive architectural elements. Together, the rules, defined in the scope of an architecture, constitute the prescriptive view of the architecture. The architecture-specific architectural elements (constructive concepts), constrained by the rules, comprise a descriptive model. The constituents of the descriptive model may undergo series of steps of refinement and instantiation until a specific system architecture is described.

5.1.3 Patterns and Guidelines

In addition to the concepts and rules, two types of "annotations" – patterns and guidelines – are envisioned by GARM as supplementary parts of an overall reference architecture description. They capture design rationale and codify experiences from applying the architecture.

5.1.3.1 Patterns: The term pattern, is used in software engineering to refer to *established structures & behaviors* at different levels of granularity and abstraction, from architectural styles, like pipe-&-filter, to object oriented patterns such as model-view-controller (MVC) pattern, to patterns related to specific problem domains, [44, 45, 48].

Patterns, as part of a reference architecture description, are captured design experience compiled in result of applying the architecture. They codify useful structures, for meeting particular needs while staying within the boundaries of architecture concepts and rules. Patterns offer solutions to specific problems in a generic, adaptable way. They describe what is considered to be "good practice," a notion used in many other engineering disciplines. Following patterns is not mandatory; their (re-)use, however, reduces design effort and guarantees practical solutions of good quality. In the context of an architectural description, patterns form a "library" of reusable parameterized configurations, suitable for reuse on the level of system design.

As an example consider the Model-View-Controller (MVC) pattern, [45]. The MVC pattern defines three types of objects: "Model" objects, which maintain the application's data; "Controller" objects, responsible for the interactions with the user through input devices; and "View" objects which display the state of a Model object in different visual formats. The consistency is achieved by a dependency mechanism which constrains the communications between the different parts and ensures that

the View components reflect the current state of the Model components. The effectiveness of the MVC pattern is based on the "separation of concerns" between problem oriented components and the end-user oriented external interfaces. This pattern is widely used in development of object-oriented systems and can be an extension to any general object-oriented architecture.

As another example, let us examine a pattern for introducing transactional behavior to an object-oriented architecture. As the basic rules of a typical object-oriented architecture do not cover the problem of transactions, i.e., a set of messages that behave as an atomic unit (either all of them are executed or none of them), to facilitate a transaction-like behavior a special type of processing building block can be introduced – a transaction manager. This component takes responsibility for supporting atomic transaction capabilities in systems with otherwise distributed control structure and has the sole purpose of coordinating messages so that the system exhibits the required behavior. Notice, however, that if the transactional behavior was an originally-prescribed architectural property, the transaction manager component – the solution provided by the pattern – would be a part of the original architectural description. Obviously, in that case the pattern supporting transactional behavior will not be needed. Collecting and reusing patterns throughout the life of an architecture saves design efforts, reduces the cost, and increases the quality of the systems developed according to the architecture.

5.1.3.2 Guidelines: Guidelines are also captured design experience. They provide explanations to well-known problems encountered in architectural design (e.g., in the boundary of a given architecture or architectural style) and suggestions how they can be avoided. They help in "staying out of trouble" when using the concepts and rules of an architecture, by providing hints and techniques for building useful models of systems. Guidelines are of a less constructive nature than

patterns and in many times read like the well-known "problem-reason-solution" tables, although the problems discussed and the suggested solutions could be of very general nature. As an example, consider the following paragraph (see [76]):

"Problem: messages between building blocks get lost during times of high system load. Possible reason: Sender authorization checking in building blocks is a bottleneck in most implementations and might lead to an overflow of the queue for incoming messages each building block has to handle. Solution: Increase the size of the queue, replicate the sender authorization checking function in the problem building blocks and provide for parallel execution of the authorization procedure. (Remark: It is of course not possible to drop authorization, as it has been stated as a rule and is, therefore, mandatory!)"

If compared with the concepts and rules, which are more top-down imposed elements in an architecture, patterns and guidelines are bottom-up, reverse-engineered information that represents feedback from different applications of the architecture and relate more closely to implementation details. Therefore, the log of patterns and guidelines related to an architecture will most likely grow over time, with the use of the architecture they belong to.

5.1.4 GARM: A Concise Summary

The set of extended BNF-like statements, shown below, summarizes the above discussion and is intended simply as a concise illustration of our generic architecture reference model – GARM. It identifies the architectural aspects addressed in GARM and underlines their relationships. The following meta-symbols are used [34, 96]:

[...|...|...] indicates alternatives

{...}* denotes zero or more iterations

{...}$^+$ indicates one or more repetitions

; indicates sequence

"..." indicates a literal string

According to GARM, an architecture is described in terms of four types of elements: concepts, rules, patterns, and guidelines; and while concepts and rules are mandatory, patterns and guidelines remain optional:

$Architecture => \{Concept\}^{+}; \{Rule\}^{+}; \{Pattern\}^{*}; \{Guideline\}^{*}$

In GARM, a further distinction is made between two types of architectural concepts: system properties, and constructive (modeling) concepts. The last are defined from multiple aspects.

$Concept => [Property|Constructive\_Concept]$

$Property => [Quality\_Related|Structural\&Operational]$

$Constructive\_Concept => [Component|Connector|Configuration]$

$Component => Functional\_Aspect;\ Behavioral\_Aspect;\ Data\_Aspect;$
$Quality\_Aspect;\ Structural\_Aspect;\ Type\_Aspect$

$Functional\_Aspect => \{Function\}^{*}$

$Function => [CommonFunction|Application\text{-}SpecificFunction]$

$Data\_Aspect => \{Data\_Description\}^{*}$

$Behavior\_Aspect => \{Comp\_Obsv\_Action\}^{*}$

$Structural\_Aspect => \{Part\_of\_Relation\}^{*}; \{Configuration\}^{*}$

$Configuration => \{Component\}^{+}; \{Connector\}^{+}; \{Rule\}^{+}$

$Quality\_Aspect => \{Quality\_Attribute\}^{*}$

$Quality\_Attribute => [Performance|Security|Timeliness|...]$

$Type\_Aspect => \{Is\_A\_Relation\}^{+}$

$Connector => C\_Functional\_Aspect;\ Data\_Aspect;\ C\_Behavioral\_Aspect;$
$C\_Quality\_Aspect;\ C\_Structural\_Aspect;\ Type\_Aspect$

$C\_Functional\_Aspect => \{C\_Function\}^{*}$

$C\_Function => [Data\_Passing|Control\_Passing|Hybrid]$

$C\_Behavioral\_Aspect => \{Env\_Action\}^*$

$C\_Quality\_Aspect => \{Interaction\_Quality\_Attribute\}^*$

$C\_Structural\_Aspect => Structural\_Aspect;\ Topology$

$Topology => [Port\_to\_Port|N\_Port]$

Rules constrain the structural and the operational properties of constructive elements and are referred to as Constraint Rules. The Constraint Rules are subdivided into two groups: Structure Constraint Rules (SCR), and Behavior Constraint Rules (BCR), where the Behavior Constraint Rules themselves are of two types Action Constraint Rules (ACR) and Stimuli-Response Rules (SRR):

$Rule => [SCR|BCR]$

$BCR => [ACR|SRR]$

Patterns are instances (which may be parameterized) of architectural configurations. All patterns together form a library of reusable configurations.

$Pattern => \{Configuration\}^+$

Guidelines list a set of possible answers for an architecture-related question:

$Guideline => Question + \{Answer\}^+$

5.1.5 GARM: Conclusions

The Generic Architecture Reference Model provides an architectural ontology and establishes a conceptual basis for describing and evaluating (reference) software architectures.

The model introduces an architectural vocabulary in terms of architectural elements (constructive concepts) and properties. It also defines an approach for

capturing of architectural constraints as a collection of structural and behavioral rules. Finally, it sets requirements for the design of techniques and tools for architecture specification.

The view of a reference architecture description, established by GARM, has served as a basis in the development of ASPECT – the architecture specification technique, discussed next.

GARM and ASPECT (GARM-ASPECT) together represent our method for architecture engineering and representation. GARM defines *what* is an architecture and what has to be specified, and in this way provides guidance to the process of architecture engineering; the accompanying architecture specification language, ASPECT, is a notation for representing architectures which gives an answer to the question *how* the elements of a software architecture can be specified.

5.2 ASPECT: An Architecture Specification Technique

ASPECT is a language for describing software architectures [72, 73, 74]. It is based on the Generic Architecture Reference Model (GARM) and provides representation constructs for expressing the constructive elements of GARM, referred to as *architectural elements*, as well as their hierarchical refinements.

It supports a normalized vocabulary for specifying the structural view of software architectures. It also establishes a framework for expressing architectural constraints as architectural rules. In addition, ASPECT supports mechanisms for extending the structural view with auxiliary non-structural information, such as behavior or quality specifications, expressed in other languages. In this respect, ASPECT provides facilities to establish relations between different specifications and to pull them together into an overall architectural description.

This language design allows flexibility and easy adaptation of the methodology to the specifics of a domain or a family of systems: the common structural properties are described in ASPECT, while the set of auxiliary specifications and the corresponding representation languages can be selected to fit the purpose of the architecture being specified, the class of systems being addressed, or domain traditions. All specifications together contribute to a comprehensive architectural description. The auxiliary, non-structural specifications can be processed by external tools, integrated with an ASPECT editor.

Our main objective in developing ASPECT was to provide means for representing domain architectures in a form that allows its communication to the main stakeholder of an architecture: architects, domain engineers, system engineers, application developers and tool (infrastructure) designers, [71, 29].

In the remainder of this section we describe the constructs and concepts of ASPECT in detail. Appendix B provides the complete abstract grammar of the ASPECT language.

5.2.1 Architectural Elements

ASPECT is built upon a set of constructive concepts, *architectural elements,* which form its core ontology and include: components, interfaces, ports, contracts, roles, scenarios and architectures. These elements constitute a "library" of generic design element-types. They are at the root of the hierarchy of architectural descriptions and have to be refined and instantiated with architecture-specific data during the process of an architecture description[6]. The architectural elements in ASPECT are first class objects, i.e., they have a name and are refinable (they may be aggregated and decomposed as well as specialized and generalized). ASPECT provides templates for representing the architectural elements, [72]. (The element-types and their relationships are illustrated in Figures 5.3 and 5.4.)

5.2.1.1 Component: As introduced in GARM, *architectural components* represent the computational entities and data of a system. Components are characterized by functions, data, observable behavior, and quality attributes, all defined in the component's *interfaces.* Multiple interfaces can be associated with a component in order to reflect different possible functional aspects, where each provides a partial external view of a component. Interfaces comprise *ports,* at which interactions between a component and its environment take place. From a structural point of view, a distinction is made between two basic types of components: (1) a *building block* and (2) a *cluster.* Building blocks are atomic units in the context of an architecture, while clusters are compositions of building blocks and/or other clusters and contracts.

[6]These elements are supported as object types in our prototype architecture specification tools – *d*-ASPECT and ArchE (see Appendix C).

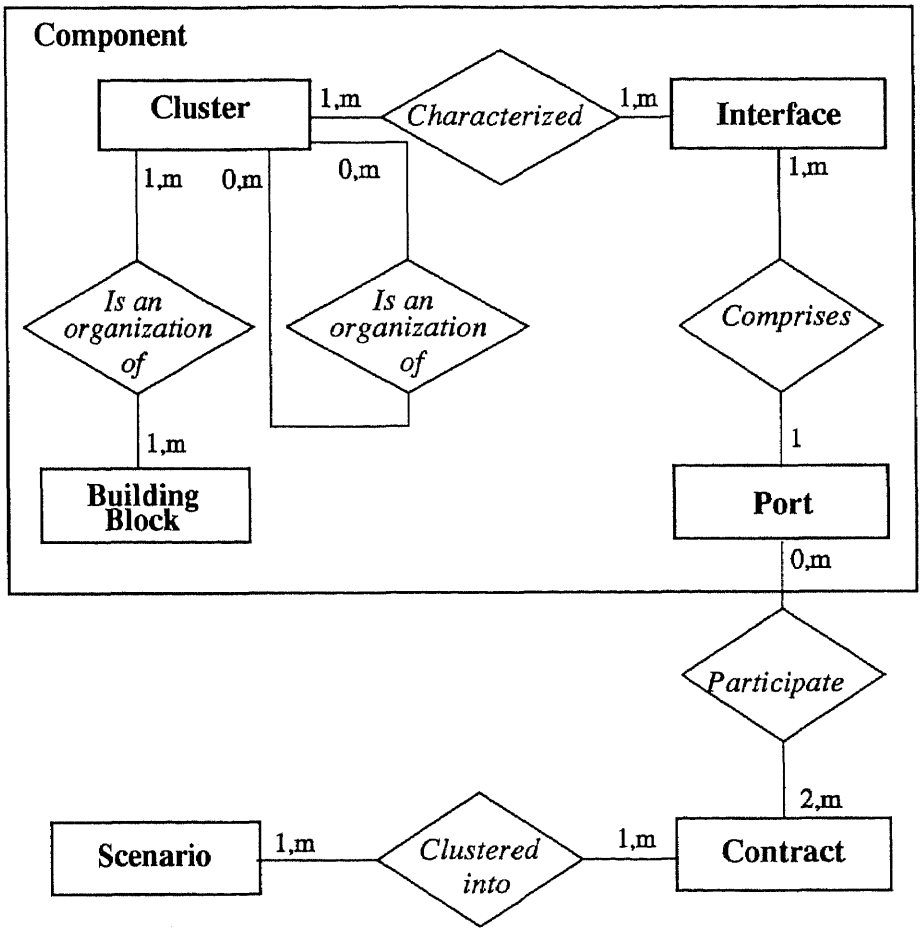


Figure 5.3 A Basic View of Architectural Elements and Their Relationships

To support the description of component types and instances, ASPECT provides a *COMPONENT TEMPLATE.* A component[7] has a name and is specified in terms of a triple which consists of:

- *HEADER,*
- *INTERFACE(S),* and
- *BODY(S).*

1. HEADER

   A header comprises definitions of component's type, part-of relations, instance identifier, and organization type – building block or cluster, as defined in GARM[8].

2. INTERFACE

   An interface is an abstraction that defines an external image of a component - functional and data properties, observable behavior and expected qualities. In ASPECT, *INTERFACE TEMPLATES* are used to represent the features of interface types. They are named and typed objects which comprise two major subelements, DATA DEFINITIONS and a list of PORT descriptions. Architectural constraints, if any, defined for an interface type are tagged by the interface name and stored as part of the rule-portion of the architecture description containing the interface (see the architecture template latter in this section).

[7]Please, notice that where no ambiguity can arise, we use the term component to refer to a component template, or a component type, as well as to its instances.

[8]In our first prototype implementation the "organization type" (of the grammar) was resolved up-front to two generic templates – for building blocks and for clusters, respectively.

- DATA DEFINITIONS

  The data definitions part captures the component's data aspect. It describes the data structures and types defined in the context of the particular interface.

- PORT

  Ports are locations at which interactions between components and their environment occur and component properties are observed. They localize and reflect the component's properties in relation to the set of addressed aspects, and as such they are characterized by functional attributes (FA); data attributes (DA); behavioral attributes (BA) – locally-applied behavior constraint rules, e.g., for ordering, dependability, synchronization, termination, etc.; and quality attributes (QA), e.g., timing, security and fault-tolerance.

  In order to support the description of ports and their attributes ASPECT provides a *PORT TEMPLATE* as a named and typed part of an interface template. The attribute elements of the port template can serve as place holders, where references to external (possibly) formal specifications can be provided (for an example, see Fig. 5.6).

3. BODY

A body is a reference to "implementations" that support the functionality and behavior observed at component's interfaces. According to our architecture model, means for describing two types of architectural components has to be provided – atomic building blocks and composite clusters.

In case of a *building block,* the body is an external REPRESENTATION – outside the domain of ASPECT. (For instance, any text or graphics representing the algorithms and data structures behind the building block's

interfaces can be referenced from the REPRESENTATION but will not be considered a part of the architecture description.) The internals of a building block are prerogatives of lower levels of design and, if associated with the architecture description, are treated as annotations. They are, however, "handy additions" which can, for example, allow relating architectural specifications to some (optional) existing implementations. They can also be used to support linking to external analysis tools or just to capture design rationale, related to the particular type of building block, e.g., some guidelines. Thus, a representation is a "hook" to externally available tools, techniques and descriptions.

In the *cluster* case, the body is *a composition*, denoting an aggregation of two or more components and connectors yielding a composite component – a cluster – where the characteristics of the cluster are determined by the components being combined and by the way they are combined. To accommodate this aspect, ASPECT provides a *COMPOSITION TEMPLATE* comprised of an INTERNAL STRUCTURE – *de facto* an architecture; and a set of ASSOCIATION objects, mapping internal and external ports. The ports of the cluster, resulting from the composition, are referred to as *external ports*, while the ports of the components participating in the composition are considered *internal ports.*

5.2.1.2 Contracts: Components in a software system are interacting entities. In order to provide means for modeling types of interactions and for expressing their properties, the *connector* concept was introduced in GARM, where it denotes a set of requirements, obligations, and constraints on two or more ports, interfaces or components, participating in an interaction, prescribing in this way their collective behavior.

The ASPECT language provides a CONTRACT construct, which is a template for describing types of *connectors*[9]. In the "ExampleClientServer" architecture (shown in Figures 5.5, 5.6) for example, two contract types are specified "ExampleRemotePipe" and "ExampleRemoteProcedure", describing two interaction models defined in the context of this architecture – Remote Pipe and Remote Procedure, [57].

A CONTRACT has a NAME and is specified in terms of:

- *HEADER*,
- *DATA DEFINITIONS*,
- *ROLES*, and
- *LIAISON*.

1. HEADER

   A header comprises definitions of contract's type, part-of relations, instance identifier, and organization type, primitive or composite, as defined in GARM[10].

2. DATA DEFINITIONS

   The data definitions part of the template describes the data structures and types defined in the context of the contract.

[9]Please, notice that the two terms – contract and connector – are used interchangeably in the rest of the presentation.

[10]In our first prototype implementation the "organization type" (of the grammar) was resolved up-front to two generic templates – for primitive and composite contracts, respectively.

3. ROLE[11]

Roles represent the external view of a connector. They define the participants in the interaction being modeled by the connector. A role is a placeholder that will be substituted by a matching port when a system is constructed. Roles define the expected behavior of the interaction points (ports) in the context of a particular connector. They prescribe the ports' behavior only to an extent required for a port to be able to "play" a role in a way that is indistinguishable in the context of the connector. In the example in Fig. 5.6, the "PipeSource" role specifies the expected behavior of a port that in a software system will "play" this role when participating in an interaction according to the "RemotePipe" protocol. In our example the match is an instance of the the "PipeSource" port type.

Syntactically, the ROLE template matches the PORT template. In this way, comparison and analysis can be easily performed.

4. LIAISON

The interaction channel established between the roles/ports is represented by the contract's liaison. A liaison represents both the operational aspect of a connector, that is the protocol, and the connector's internal-structure – the body of composite connectors. The protocol constrains the cooperative actions of the roles and is a home for Stimulus Response Rules (SRR). The optional internal structure, on the other hand, allows for modeling complex structures supporting an interaction.

The LIAISON element of the CONTRACT template comprises a PROTOCOL and a BODY parts. The PROTOCOL part names an interaction protocol and

[11]The role concept here is similar to the role element of the channel concept in ESTELLE, [38].

can be a pointer to an external, possibly formal specification of the protocol[12]. The BODY part accommodates the internal structure aspect of a connector. It is empty or a REPRESENTATION in case of a primitive contract (no internal structure), while it is a composition CTCOMPOSITION in the case of a composite contract. In the "ExampleRemotePipe" contract (Fig. 5.6), for example, the body (CtBody) is empty as the contract is primitive. A contract COMPOSITION has two elements: first, an INTERNAL STRUCTURE which is a reference to an architecture that defines the components, contracts and their compositions encapsulated in the contract; second, a number of ASSOCIATIONS, which represent the mappings between internal and external roles. (The roles of the composite contract are defined as external, while the encapsulated roles are considered internal.)

A helpful analogy can be made between the contracts of ASPECT and the channels of the ESTELLE language, [38], with two major differences. Contracts in ASPECT are decomposable, and capture interactions with multiple aspects, defined at various levels of abstraction, while the channels in ESTELLE are atomic and transfer only messages.

5.2.1.3 Scenario: Scenarios name static or dynamic *configurations* (see GARM) of interconnected components and connectors, where components' ports are mapped into connectors' roles.

ASPECT provides a SCENARIO template which has two subelements: TYPE and DESCRIPTION. A scenario's type is either "STATIC" or "DYNAMIC". Any description is a set of port-role mappings. A static scenario is a set of port to role mappings that represent a system's topology. A dynamic scenario is a list of port instance to role instance mappings which form a "use case," [65, 66].

[12] A use of formal techniques for specifying protocols, roles and ports is recommended, [8, 25].

5.2.1.4 Architecture: A *reference architecture description* comprises *component types*, representing the computational entities, *connector types,* representing the allowed interaction models, and a set of *architectural rules*, capturing constraints on components, connectors and their relationships. Also, typically as part of an application architecture, configurations of components and connectors are defined, referred to as *scenarios.* The architecture may also be annotated with "design rationale" and "design experiences," in terms of guidelines and a library of patterns (as discussed in GARM), [119]. An architecture has a name and may be a refinement of another architecture, which defines its type. Also an architecture can be a part of a component, i.e., representing the infernal architecture of a cluster component.

In order to support the description of types of architectures, ASPECT provides an ARCHITECTURE template which pulls together all of the above elements. The template contains the following subelements: HEADER, COMPONENTS, CONTRACTS, SCENARIOS, RULES, and DESCRIPTIONS.

The header defines the position of the architecture in the hierarchies of architectural descriptions. It comprises definitions of the architecture type, part-of relations, and instance identifier. The components sub-element contains a set of components. The contracts sub-element defines a set of contracts. The scenarios contains a set of scenario names. The rules sub-element contains all rules whose scope is defined to be this architecture or its constituent elements. Finally, the descriptions sub-element is the home of all annotations.

As a simple example consider the "ExampleClientServer" architecture portrayed in Fig. 5.5. It comprises two component types ("ExampleClient" and "ExampleServer"), two types of contracts ("ExamplePipe" and "ExampleProcedure"), and a set of "Rules". Fig. 5.6 presents a partial textual description of this architecture and its constituent elements.

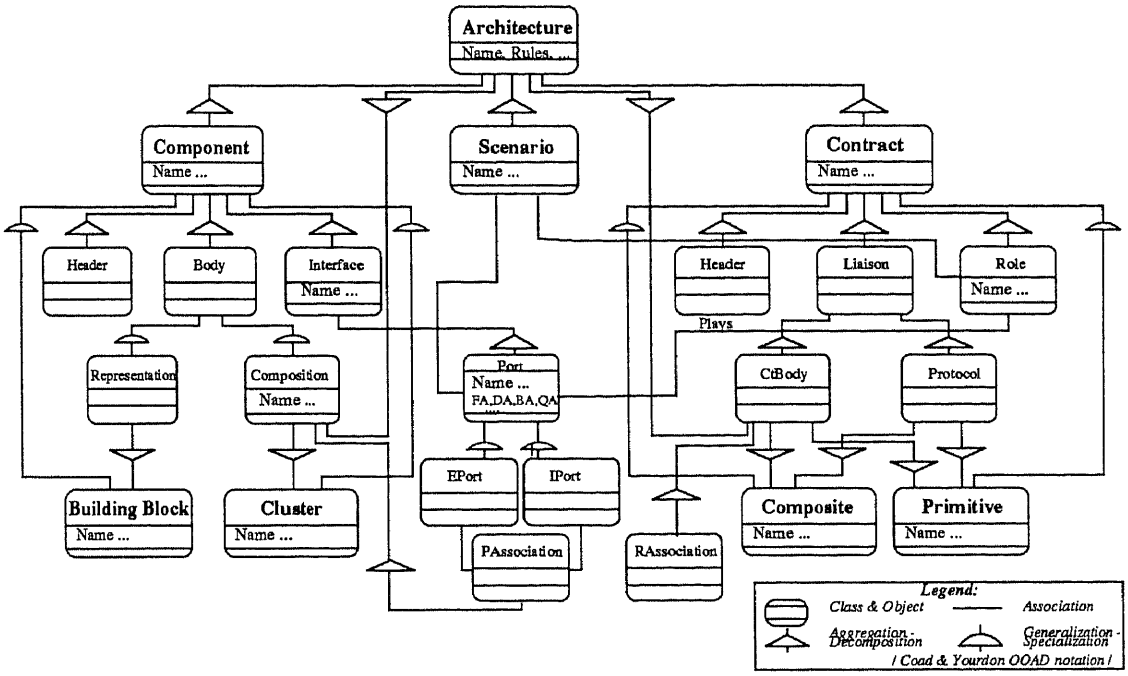


Figure 5.4 Architectural Elements – A Graphical Overview

In summary, ASPECT provides facilities for specifying the *architectural elements* discussed above and illustrated in Fig. 5.4. It also supports refinement processes based on the two forms of abstraction discussed earlier – aggregation-decomposition and generalization-specialization – which result in hierarchies of architectural descriptions. The aggregation-decomposition is supported by the availability of atomic and composite elements, components and connectors, and the mechanisms for matching internal and external ports and roles, respectively. The subtyping discipline of ASPECT needs further discussion and is addressed in Chapter 6.

5.2.2 ASPECT in Template Form

We have derived a simple template-oriented notation, following the abstract grammar, included in Appendix B. For ease of use a few optimizations have been made to the general approach captured in the grammar. Namely, separate templates are provided for the atomic and composite entities: building block – cluster, and

primitive – composite contracts. For the rest, the translation between the abstract and the concrete forms is immediate.

5.2.2.1 Visibility and Naming Rules: Architectural elements are identified and referred to by means of associated names. Elements are grouped into entity types to allow flexible naming rules. The following element types exist:

1. Architecture
2. Component (Building Block, Cluster)
3. Contract (Primitive Contract, Composite Contract)
4. Interface
5. Port
6. Role
7. Scenario
8. Composition
9. Contract Composition

The following are the scope units:

1. Architectural Repository
2. Architecture
3. Cluster
4. Composite Contract

No two elements within a scope unit and belonging to the same element type can have the same name. The name of an element is visible within the enclosing scope unit (defined by the element's part-of attribute). Name resolution follows standard rules for lexical nesting and inheritance; conflicts will be reported. Currently it is the designer's responsibility to resolve any remaining name conflicts, although the semantics could certainly be extended to use type or parameter information, or other resolution rules to eliminate some conflicts. For a further discussion on scope, see Chapter 6.

5.2.2.2 The Generic Templates – Textual Forms: Notice that: (1) text in italics is only explanatory and is not part of the generic templates; (2) keywords are in boldface; (3) the closures (*) and (+) have been suppressed and (...) used in both cases; (4) the braces {}, parentheses ( ), and commas (,) are literal.

THE GENERIC TEMPLATES:

Generic_Architecture: **Architecture**{
Header{
Type: { *"Generic"* }
POF:{ *Component_name, ...* }
Implements:{ *Architecture_name, ...* }
}
Components: { *Component_name, ...* }
Contracts: { *Contract_name, ...* }
Scenarios: { *Scenario_name, ...* }
Rules: { *Generic_Architecture_rules* }
Description: { *Text* }
}

Generic_Building_Block: **BuildingBlock**{
Header{
Type: { *"Generic"* }
POF:{ *Architecture_name, ...* }
Implements:{ *BuildingBlock_name, ...* }
}
Interface: { *Interface_name, ...* }
Body: { *Representation_name, ...* }
}

Generic_Cluster: **Cluster**{

Header{

Type: { *"Generic"* }

POF:{ *Architecture_name, ...*}

Implements:{*Cluster_name, ...*}

}

Interface: {*Interface_name, ...*}

Body: { *Composition_name, ...*}

}

Generic_Composition: **Composition**{

Int_structure: { *Architecture_name* }

Associations: { *(Int_port_name, ...;*

Ext_port_name), ...}

}

Generic_Interface: **Interface**{

Type: { *"Generic"* }

Data_Model: {*External_text* }

Data_Description: { *Data_definition, ...*}

Compatible_With: { *Interface_name, ...*}

Port: {*Port_name, ...*}

}

Generic_Port: **Port** {

Type: { *"Generic"* }

Port_attr{

FA: { *Function_signature, ...*}

DA: { *Data_description*}

BA: { *External_spec_reference* }

QA: { *External_spec _reference* }

}

}

Generic_PContract: **PContract**{

Header{

Type:{ *"Generic"* }

POF: { *Architecture_name, ...*}

Implements: { *PContract_name, ...*}

}

Data_Model:{*External_text* }

Data_Description: {*Data_definition, ...*}

Role: {*Role_name, ...*}

Liaison{

Protocol:{*Extenal_spec_reference* }

CtBody:{*Representation_name, ...*}

}

}

Generic_CContract: **CContract**{

Header{

Type:{ *"Generic"* }

POF: { *Architecture_name, ...*}

Implements: { *CContract_name, ...*}

}

Data_Model:{*External_text* }

Data_Description: { *Data_definition, ...*}

Role: { *Role_name, ...*}

Liaison{

Protocol:{*Extenal_spec_reference* }

CtBody:{*CtComposition_name* }

}

}

Generic_CtComposition: **CtComposition**{

Int_structure: { *Architecture_name* }

Associations: { *(Int_role_name, ...;*

Ext_port_name), ...}

}

Generic_Role: **Role** {

Type: { *"Generic"* }

Role_attr{

FA: { *Function_signature, ...*}

DA: {*Data_description* }

BA: { *External_spec_reference* }

QA: {*External_spec_reference* }

}

}

Generic_Scenario: **Scenario**{

Type: { *"Static" or "dynamic"* }

SDescription: { *(Port_name; Role_name), ...*}

}

The semi-formally defined textual forms presented above characterize our representation approach and have been used in examples and case studies. They, however, are cumbersome and haven't been optimized for use. They are rather a "model" of the graphical templates supported by the ASPECT prototype editors – *d*-ASPECT and ArchE, [71, 74]. A brief overview of these tools is provided in Appendix C.

5.2.3 ASPECT: A Toy Example of Application

The following is a simple example of applying GARM-ASPECT method. Its only purpose is to illustrate the basic construct of ASPECT as well as their relationships. In the example, we describe a simple client-server architecture defined after the Guilford and Glasser 's *channel model*, combining remote pipes and procedures, [57]. (In the text, the names of architectural elements and constraints are in boldface.)

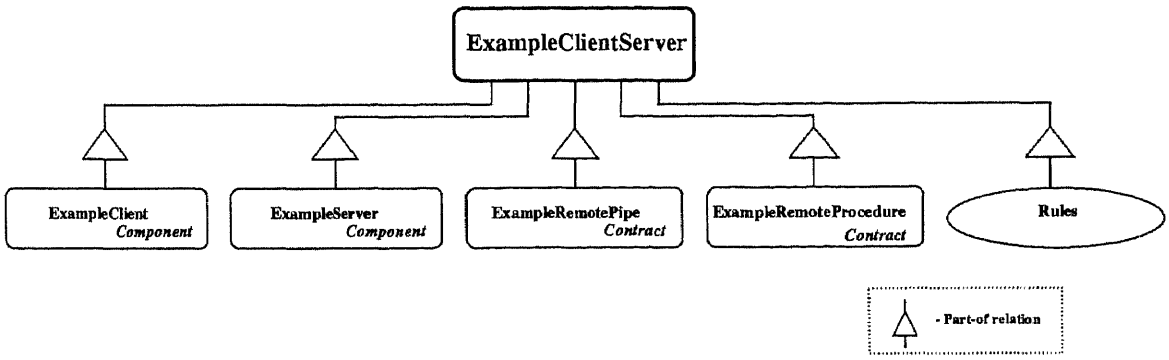


Figure 5.5 Reference Architecture: An Example

The reference architecture **ExampleClientServer**, portrayed in Figure 5.5, comprises two component types, **ExampleClient** and **ExampleServer**, two types of connectors, **ExampleRemotePipe** and **ExampleRemoteProcedure** and a set or **Rules**.

In Figure 5.6, a textual description[13] of the architecture and some of its constituents are presented. The generic templates of ASPECT are refined, adding architecture-specific data. The **ExampleClientServer** architecture, for example, is a refinement of the generic architecture template. Therefore, it is of type *Generic (architecture)*. The **ExampleClient** building block is a part-of the **ExampleClientServer** architecture. It is of type *Generic (building block)*. Its body is empty as it is a building block. It has one interface type defined – the **Client** interface. The interface comprises two ports, **PipeSource** and **ProcedureSource**, from which

[13]Notice, that the syntax used is only for illustration purposes. Some of the sub-elements that were not defined are omitted from the description for simplicity.

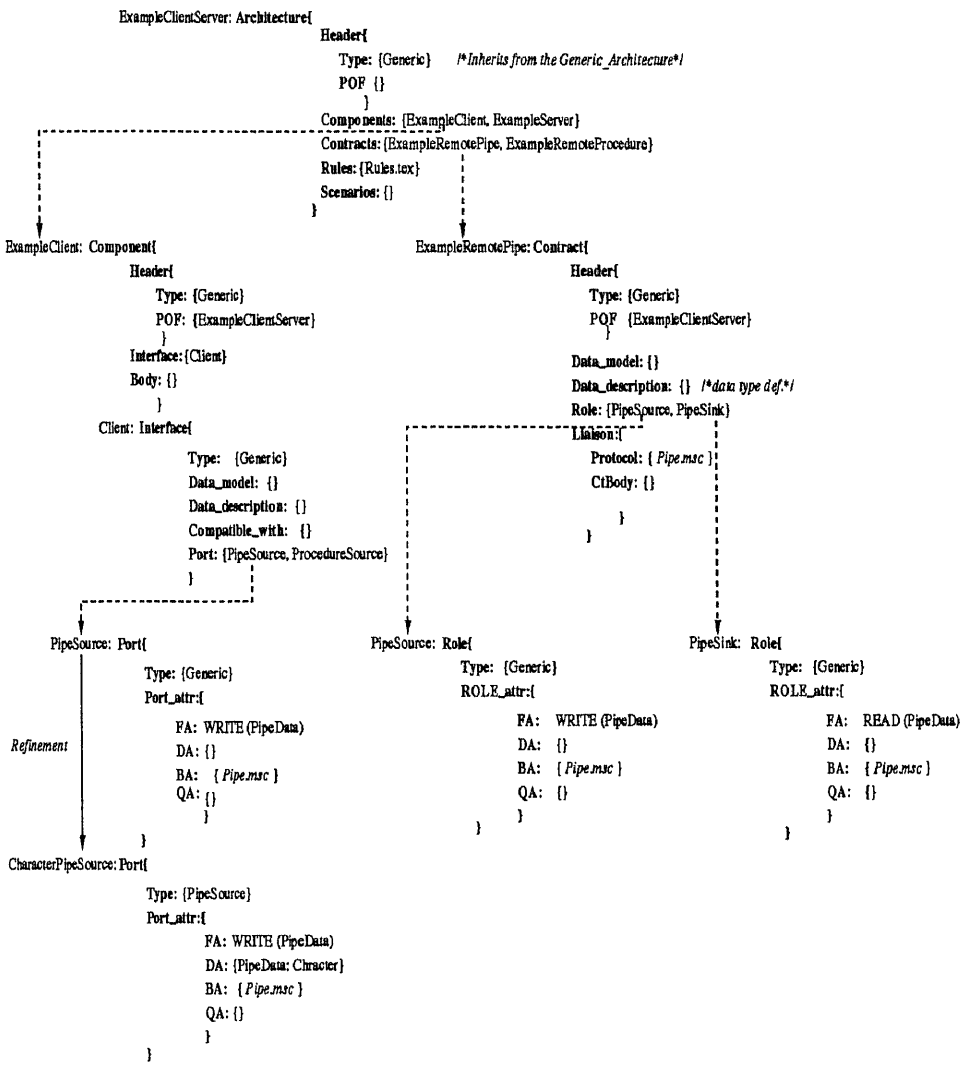


Figure 5.6 ExampleClientServer Architecture: A Partial Textual Description

instances will be created when creating instances of the **Client** interface type. The description of the **ExampleRemotePipe** contract is also included in Fig. 5.6. It comprises two roles: **PipeSource** and **PipeSink**, whose descriptions are also included.

Behavioral attributes of ports, roles and protocols can be defined in terms of external formal specifications, for instance using WRIGHT, a language specifically designed for this purpose. In our experimentation we have also used Message Sequence Charts (MSC), a trace language for specification and description of the

communication behavior of system components and their environment, [25]. For instance, the behavior attributes of port and role descriptions in Fig. 5.6 contain references to MSC descriptions (*Pipe.msc*). These descriptions are external to the domain of ASPECT.

The **ExampleClientServer** architecture can be further refined into more detailed or problem-specific reference architectures. It can also be used as a reference in engineering of specific system architectures.

CHAPTER 6

ASPECT SEMANTICS

In this chapter we address two aspects of ASPECT semantics. We present the refinement process supported by the language by defining the relationships between the architecture elements types, subtypes and instances. We discuss the issue of open and closed specifications. We also clarify the open semantic framework underlying the language design.

6.1 ASPECT: Types, Subtypes and Instances

ASPECT supports the following groups of architectural element types: architecture, component, contract, interface, port and role (see Chapter 5). The language type definitions are predicates that elements can satisfy. A type definition therefore determines a set of architectural elements (instances) – those that satisfy the type predicate. Specifically, an element that satisfies type τ's predicate is said to satisfy type τ. In this section we define the ASPECT semantic rules in regard to defining element types, subtypes and instances.

6.1.1 Defining Element Types and Instances

6.1.1.1 Element Types: An architecture element type specifies the structure and attributes that an element must have in order to satisfy the type. The basic syntax for declaring an architecture element type is:

< TypeName >: *< ElementType >*{

Header{

Type: { }

POF:{ }

Implements:{ }

}

< *RequiredStructureAndConstraints* >

}

In the above informal syntax the <*TypeName*> specifies a valid identifier. The <*ElementType*> can be any of the literals **Architecture, Component, Contract, Interface, Port** or **Role**. The <*Required Structure And Constraints*> section refers to the structure and rules defined for the corresponding element type. It defines the minimal requirements that all instances of a particular type must satisfy.

The following example is a specification of a simple Client component type :

```
Client : Component{
        Header{
            Type: { }
            POF:{ClientServerArchitecture}
            Implements:{ }
            }
        < Client : RequiredStructureAndConstraints >
        }
```

The three fields of the header define the place of the element type in the specification hierarchies. The **Type** field is empty. This implies that the Client type inherits from the Generic Component definition which defines only the basic structural elements of a component. The **POF** element defines the container architecture for this component, i.e., the ClientServerArchitecture, which de-facto defines the scope of this type. If the POF is not defined, the scope of the type will be the global scope, i.e., the entire architectural repository. The **Implements** field must be empty as we are defining a new type and not an instance.

Scope of Element Type Declarations:

The scope of type declarations in ASPECT may be global (the repository of architectural specifications) or architecture specific, including reference architecture-specific (i.e., architecture type specific) or system architecture-specific (i.e., architecture instance specific). Type names are lexically scoped. Element types declared outside of any architecture have global scope. Types with global scope are visible to all instance elements in that global space. They are, however, not visible within the scope of architecture types which de-facto define classes of systems. Types declared within an architecture instance are visible in this instance. Types defined within an architecture type are visible within the class of systems and within the systems that belong to that class.

6.1.1.2 Instances: The following (informal) syntax is used to define instances of the basic architecture element types.

$< InstanceName >$: $< ElementType >$\{

Header\{

Type: \{ \}

POF:\{ \}

Implements:\{$< TypeName >$ $|default\}$

\}

$< ExtendsWithStructureAndConstraints >$

\}

The $<TypeName>$ in the Implements part of the Header defines the type from which the instance is created. This attribute must be specified for all instances. If an explicit type name is not supplied then the special word **default** is expected which implies the respective generic element type. (Notice that if the Implements part is empty the definition will be understood as a type and not as an instance.)

In the example below we create an instance from the Client component type defined earlier.

```
DirClient : Component{
                    Header{
                            Type: { }
                            POF:{ }
                            Implements:{Client}
                            }
                    <>
                    }
```

The Client type specification imposes the following constraints on DirClient component: (1) the instance will comprise the interfaces, ports and port attributes (see Chapter 5) defined by the Client type. (2) The DirClient does not introduce any extensions. If an extension is defined, then an union operation is performed recursively on structural elements declared in either the type constructor or the extension.

The above declaration is semantically equivalent (up to the type constraints) to the following definition directly specifying a component instance:

```
DirClient : Component{
                    Header{
                            Type: { }
                            POF:{ }
                            Implements:{default}
                            }
                    < Client : RequiredStructureAndConstraints >
                    }
```

Here the DirClient is an instance form the generic component type. In this case, the architect should define explicitly the entire structure and the constraints for the component.

6.1.1.3 Element Type Semantics: An architectural element type specification defines the minimal set of structural elements and attributes that the elements of that type must have. For every architectural element type τ, we assume a boolean function f_τ on architectural elements which defines membership in τ. If the function f_τ evaluates to true for element e, then element e satisfies type τ.

Declarations of substructure and attributes in a type τ have the following informal semantics:

- substructure: All elements e that satisfy type τ have the structural elements (e.g., interface and ports for a component type) defined by τ, as elements of their structural view. The body element of a building block component type definition is a collection of attributes (see attributes). The elements of a body of a cluster component type definition are structures which are instantiated recursively as defined in this rule. If multiple body elements are defined for a cluster type the instances of this type implement one specific body element.

- attributes: All elements e that satisfy type τ have the attributes (e.g., QA of a port type, or Data_model of an interface) defined by τ, as elements of their attribute-definitions. (Attributes are identified by their names. Multiple attributes cannot have the same name.)

- constraints: Constraints defined in the scope of an element type are imperative for elements of that type. They can be redefined, introducing stronger constraints. (Currently, ASPECT implementations do not provide automated

checks for this process. It is the architect's responsibility to verify the relationships between the constraints. External tools could also be integrated to provide automated support.) Constraints are identified by their names. Multiple constraints cannot have the same name. Constraints are defined in the scope of architectural element types and their names are qualified by the name of the element type.

6.1.2 Subtypes

ASPECT supports a strict form of subtyping that ensures substitutability of subtypes for supertypes. That is if a type S is a subtype of type $S < T$ than an element that satisfies S may be used whenever an element of type T is expected. The following informal syntax is used in ASPECT for subtyping:

< SubTypeName >: *< ElementType >*{

Header{

Type: {*< SuperTypeName >*}

POF:{}

Implements:{ }

}

< ExtendsWithStructureAndConstraints >

}

The *<SubTypeName>* specifies a valid identifier. The *<ElementType>* can be any of the literals **Architecture, Component, Contract, Interface, Port** or **Role** referring to the corresponding category of architectural elements, e.g., components. The *<SuperTypeName>* is a valid identifier of an element type from the same category as the addressed element type. There may be multiple

supertypes, <*SuperTypeName*>, defined, as ASPECT supports multiple inheritance. The semantics of <*ExtendsWithStructureAndConstraints*> is straightforward. The subtype consists of the union of the <*Required Structure And Constraints*> of the supertype(s) and the structure and constraints declared by the <*ExtendsWithStructureAndConstraints*> section. Any instances of a subtype can be used in any place where an instance of one of its supertypes is required.

Let us look at the following example in which a PipeClient component type is defined as subtype of the Client supertype:

```
PipeClient : Component{
                    Header{
                        Type: {Client}
                        POF:{ClientServerArchitecture}
                        Implements:{ }
                        }
                    < ExtendsWithPStructureAndPConstraints >
                    }
```

This definition is equivalent to the definition shown below, where the PipeClient is a subtype of the Generic Component element of ASPECT (there is no other supertype defined). The structure and the constraints for this component type are explicitly defined in its declaration.

```
PipeClient : Component{
                    Header{
                        Type: {}
                        POF:{ClientServerArchitecture}
                        Implements:{ }
                        }
```

< *Client* : *RequiredStructureAndConstraints* >

< *ExtendsWithPStructureAndPConstraints* >

}

6.1.2.1 Subtyping Semantics: GARM-ASPECTS supports structural subtyping[1] as follows:

- **Architecture:** Architecture A is a subtype of architecture B if: (1) the rules of architecture A are equivalent to or stronger than the rules of architecture B, that is, the rules of A imply the rules of B (The implication cannot be checked in the current version of ASPECT. Rules are identified by their names.); (2) the components and contracts of A are subtypes or equivalent of the components and contracts of B, and for any element of B there is a corresponding element of A, although there may be additional elements in A.

  Multiple inheritance is supported. Assuming no name conflicts, semantics for multiple supertypes are straightforward. The subtype consists of the union of the types defined in all supertypes and those defined in the subtype. When creating an architecture subtype with multiple supertypes there is a possibility of name conflicts, as the names of architectural element types are scoped by the architecture in which they are defined; thus elements from the same category but from different supertype architectures may have the same type name. Such a name conflict will generate an error.

- **Component**

  Component A is a subtype of component B if any interface of A is a subtype, equivalent or additional in regard to the interfaces of B. Moreover, for any

[1] Structural subtyping as used in GARM-ASPECT is analogous to subtyping for records.

interface of B there is a corresponding interface of A. (The body elements are encapsulated by the interfaces and do not need to be specifically addressed in the context of subtyping, assuming that the bodies of a subtype properly support the interfaces of that subtype.)

- **Interfaces**

  Interface A is a subtype of interface B only if the attributes of A are equivalent to the attributes of B; the constraints on A are equivalent to the constraints of B (or stronger, i.e., constraints on A imply the constraints on B; this, however, currently cannot be checked), the "protocol", [157], of B is a subset of the protocol of A, i.e., the sequences of operations accepted by A include the sequences of operations accepted by B (the protocol specification is a constraint and has to be checked externally); the ports of A are subtypes, equivalent, or additional in respect to the ports of B, and each port of B corresponds to some port of A.

- **Ports**

  Port P is a subtype of port T only if the attributes of P are equivalent to or subtypes of the attributes of T. (To provide more flexible sybtyping mechanisms, the semantics of the ports have to be further defined.)

- **Contracts**

  Contract A is a subtype of contract B if any role of A is a subtype, equivalent or additional in regard of the roles of B. Moreover, for any role of B there is a corresponding role of A. (The liaison elements are encapsulated by the roles and do not need to be specifically addressed in the context of subtyping.)

- **Roles**

  Role R is a subtype of role G only if the attributes of R are equivalent to or subtypes of the attributes of G. (The same as for the ports: to provide more flexible sybtyping mechanisms, the semantics of the roles have to be further defined.)

- **Attributes**

  Two attributes of the same type (e.g., the FA attributes of ports) are equivalent if their definitions are the same. (The definition of an attribute is a name, possibly of an external specification.) An attribute is a subtype of another attribute (of the same type) if their definitions relate in the following way. For every element of the supertype definition there is a corresponding element (with the same name and from the same group) in the subtype definition. (The elements of an attribute definition can be grouped, e.g., a DA of a port may have two groups of elements, i.e., *in* and *out*, each containing multiple elements, in which case this division must be respected in the subtype.) The elements of an attribute definition can be further constrained (typically, during an application architecture definition) with qualifiers, such as operation names or parameter names. For example, the FA of an *RPC* port defined as *REQUEST* in a caller component of a conceptual architecture can be refined in a subtype port of a caller component of an application architecture as a *REQUEST=RequestQuota*, where the *RequestQuota* portion *de-facto* names an operation to be called in a callee component.

6.2 Semantic Framework

Above all, ASPECT is concerned with the architectural structure of software systems and thus does not define any specific behavioral semantics. Rather it defines a

framework which provides a basic structural semantics and allows external specifications to be associated with the ASPECT specifications in order to define the behavioral aspects of architectural descriptions.

The structural view is naturally expressed in elements, relationships and structural constraints (SCR or scenarios). An architectural specification in ASPECT is a prescription that system designs and implementations have to be validated against. (The semantic view presented below may evolve with our experience in using ASPECT growing.)

For a specific system architecture: The semantics is based on the closed world assumption, which implies that all components, interfaces, ports, contracts and roles have been defined for the addressed architecture; the static scenario defines the total system configuration; and there are no additional elements. (If some elements are not yet defined the specification is marked as incomplete, using a special character (*).) The ports that are not mapped with roles are the external system ports, which if the system is to be used as an element of a system of systems will be used during the integration process. Dynamic scenarios are only use cases. Also, all elements are uniquely identified (which follows from the ASPECT naming conventions). There are no two elements that have the same name but different meaning. Two elements with the same name are the same.

For reference architectures: The structural semantics of a reference architecture is constrained by the SCR with the default being that element types can be extended with additional sub-elements (parts) and new constraints can be added if consistent with the existing ones. An element may be defined as closed (see specific system architecture above) by adding a rule that states that no new sub-elements can be added to this element. Another, related aspect, is the ability to define

optional and multiple sub-elements. For the implementation of ArchE, for example, mainly triggered by the need to represent hardware components in an architectural repository, we have added to the component and interface constructs the ability to define explicitly multiple and optional sub-elements. Such architectural decisions can also be captured as constraints. (The default number of sub-elements of a specific type is one, the optional-multiple is represented as (0-m) and multiple-and-required is (1-m) or (n-m).) The static scenarios defined for a reference architecture represent allowed patterns of configuration. (If no constraint-rules are specified, the static scenario(s) specify allowed configurations but there may be additional configurations.) The dynamic scenarios represent use-cases. They should not violate the structural constraints; beyond that, they are a test-facility for validation of the systems developed according to the architecture (typically, application architecture).

6.3 Discussion

Returning to the ASPECT's design goals presented in Chapter 5, we can now see how the language addresses them. ASPECT addresses its primary goals – support for describing common structural aspects of software systems in a domain and for creating a skeleton for extending the structural view with auxiliary formal and informal external specifications in other possibly domain specific languages – by providing a simple basic vocabulary of architecture element types including *architecture, component, contract, interface, port, and roles.* It also provides the *scenario* construct to explicitly represent architectural configurations. These elements create a basis sufficient for representing structural view of an architecture. Beyond this structural view, the ASPECT intent is only to define a framework for addressing architectural issues in which additional tools can be integrated with an ASPECT editor, and architects and designers can work in a systematic and guided way toward a comprehensive overall architectural specification. Of course, the more tool support is

provided, for example, for specification and analysis of behavior, the more expressive and verifiable architectural specifications are created. This language design also allows the use of formal and informal elements together, building an overall architectural specification that appeals to multiple stakeholders and which is of specific value when describing domain architectures.

By binding very few decisions about the operational semantics in an ASPECT specification, the architects in a domain are free to select additional domain-specific languages with appropriate semantic models to write auxiliary specifications and to attach them to the structural core provided by an ASPECT specification.

If we had chosen to define a specific semantics of the ASPECT ports, for instance, this would allow more strict subtyping and verification mechanisms to be supported, but would constrain the language to specific classes of systems, which would contradict the goal of supporting the description of domain architectures, which by definition are heterogeneous collections of systems. Let us, for example, assume that we have defined the ports to be operations. Then we can define more precise subtyping rules for the ports functional attributes as follows: operation f is a subtype of operation g if all input parameters of f are supertypes of the corresponding input parameters of g, and all output parameters of g are supertypes of the corresponding parameters of f. (Corresponding means that the parameters have the same name, for instance.) Further, a more constrained behavioral subtyping relation could be added as follows: the preconditions of g must imply the preconditions of f, and the postconditions of f together with the preconditions of g must imply the postconditions of g. All this will provide for analysis and quality of architectural designs; however, it will also constrain the language to specifying naturally only systems from one specific class.

Despite its simplicity, GARM-ASPECT provides a non-trivial basis for architecture representation and analysis. First, it separates the computations and the

interactions and provides two separate constructs to represent them. This permits an independent treatment of interaction models and allows the definition of abstract types of interactions, above the typically used remote procedure calls and message passing. Second, a component may have multiple interfaces to represent multiple external views. Third, components and connectors/contracts have bodies representing their internal organization and properties. This allows information hiding, supports aggregation-decomposition abstraction, and permits the specification of multiple views. Fourth, the use of types, subtypes, and instances of architectural elements allows the specification of reusable elements and patterns of system organization, and provides for the application of generalization-specialization mechanisms to help to deal with system complexity.

CHAPTER 7

CASE STUDY: DirSA

In this chapter we summarize the results from a case study in which the GARM-ASPECT method has been applied in an industrial environment. (The results of the case study are published in September 1998 issue of Bell Labs Technical Journal.)

7.1 Introduction

This case study exemplifies GARM-ASPECT as an approach in the architecture engineering phase of a project, and shows how the method can be applied to a significant problem in the domain of enterprise communication systems.

Enterprise communication systems are integrated solutions which offer extended voice, data and unified communication services to a business. The growth in business needs and system complexity has revealed many shortcomings of existing engineering practices and has been a principal source for increased interest in software architecture and architecture-focused methodologies and tools. Architects practicing in the domain have been searching for new methods to provide: (1) guidance to their engineering processes and (2) modeling concepts and notations with sufficient expressive power to handle the architectural issues of the domain. Consistent with general software industry trends, their attention has recently shifted from specific system architectures to reference architectures, family architectures and more general architectural frameworks, all seen as means to support planning, development, reuse, integration, management, maintenance, and evolution of computer-based systems in the domain.

The Directory System Architecture project, in which GARM-ASPECT has been used, had to define an architecture, called DirSA, to meet the needs of unified

directory services for an overall enterprise solution of integrated business communication systems.

The case study had three main objectives:

- to evaluate the architectural solution space;
- to apply the GARM-ASPECT method for engineering and representation of the architectural solution: DirSA;
- to investigate how the availability of a common, well-defined architecture model and representation language influences the process of architecture engineering and trade-off analysis.

In addition, we have chosen to establish an initial architectural framework which defines the context and the boundaries of the directory architecture. (Using such framework, on-going architectural efforts can be interrelated and commonly understood. Even more importantly, future efforts can be derived and systematically guided.) The results from this effort are not discussed in detail in this chapter as they are external to the main focus of the case study. They have, however, have influenced the work and shaped the overall outcome to a large degree.

In the following sections we discuss the important elements and aspects of the case study and present the results from it. Figure 7.1 illustrates the relationships among the basic elements of the case study: METHOD, PROBLEM DOMAIN, REFERENCES, and ARCHITECTURE. Section 7.2 provides a pointer to the GARM-ASPECT METHOD. An overview of the Directory PROBLEM DOMAIN is presented in section 7.3. Some drawbacks in current implementations are discussed and a set of customer requirements are highlighted. Section 7.4 points to Directory-related STANDARDS. The architectural SOLUTION is discussed in section 7.5. We briefly define the context of the target architecture, identify and compare

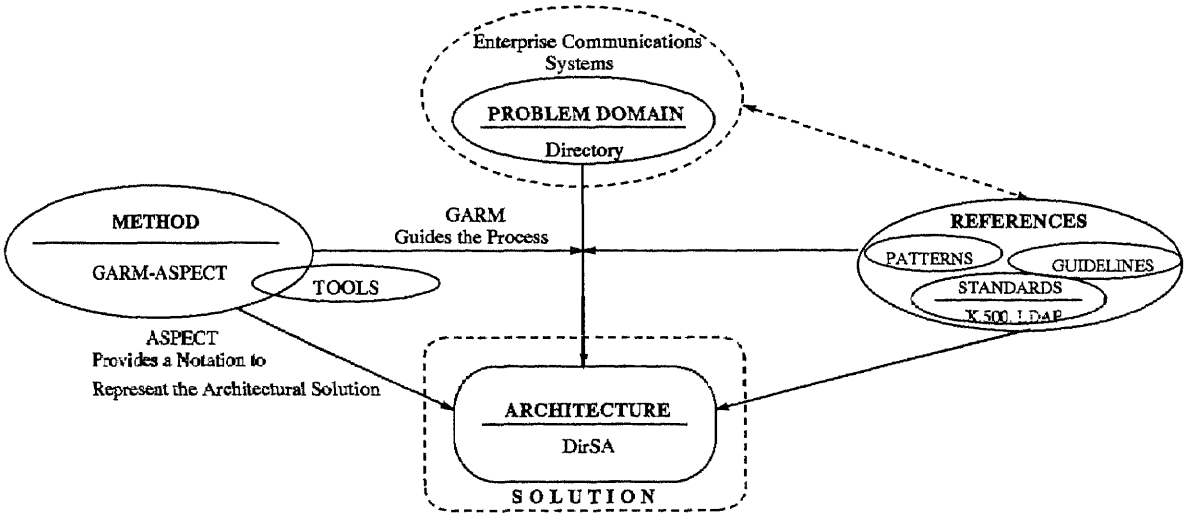


Figure 7.1 PROBLEM–METHOD–SOLUTION

several alternative solutions and demonstrate how ASPECT can be used to specify architectural structures. Section 7.6 is a summary of our results.

7.2 The Method: GARM-ASPECT

As defined in this thesis, GARM-ASPECT is a general method for architecture engineering and representation (see Chapter 5). Its core concepts and notation are domain independent. It also supports placeholders for auxiliary, external specifications, or pointers to such specifications, which can be written in domain specific languages.

In the case study, the Generic Architecture Reference Model (GARM) provided the fundamental understanding of what an architecture is and what are the basic modeling concepts and abstractions that can be used for describing the architectural solution. As the model implies a way of thinking and an engineering strategy, it has guided the process of architecture engineering. The ASPECT ADL provided the notation for describing the target Directory System Architecture (DirSA).

7.3 The Problem Domain: An Introduction to the Directory Project

The Directory project focused on development of a coherent Enterprise Directory System architecture that supports common, across multiple communication products, directory services, and meanwhile remedies problems with the legacy directory implementations in the domain. The architecture had to introduce a standard-based directory and to prescribe ways for its integration with communication servers and applications from the domain.

Directories essentially act as repositories of network user and resource information. Consider, for instance, the Domain Naming System (DNS), the well-known hierarchical, replicated naming service on which the Internet is built. Although DNS is the backbone directory system of the largest data network in the world, it doesn't provide enough flexibility to act as an enterprise directory. DNS is largely a service for mapping machine names to IP addresses. A full directory service provided by an enterprise-capable directory system must be able to map names of arbitrary objects (e.g., users, machines, applications, systems and services) to any kind of information about those objects, [27]. It must allow for locating diverse types of entities from users to systems and services.

Today's enterprise communication systems comprise different products such as switches, messaging systems, multi-media communication support systems, and Internet applications (e.g., voice and fax over Internet). Most of them contain some form of independent directories supporting different subsets of directory-related functionality. The problems with these existing solutions are manifold. First, these directories cover different populations, have different attributes and constraints as well as user and administration interfaces. They don't meet the customer requests for unified and possibly single-entry administration across multiple products operating on their premises. This situation is further aggravated by the need to integrate

"closed" products from different vendors as well as by the increasingly quick introduction of new products, each adding yet another directory of their own.

A different aspect of the problem is that directories are currently perceived as just a means for recovering phone numbers, addresses and possibly account information. This is a serious understatement of the potential utility and power of directories. When properly designed and implemented, a directory becomes an information storage and retrieval system, outfitted with a powerful query-based interface. As such, a directory can process "all" sorts of information pertinent to an organization and its business operations. As more sophisticated multi-media and collaborative communications solutions come into place, extended functionality, such as address conversions or replication and synchronization of directory information across networks, needs to be supported.

A third set of problems, constraining the current use of directories, includes implementation-related problems: rigid database systems, lack of integration with the communication tasks, and complexity of the query languages.

The Directory System Architecture (DirSA) had to address the above problems and to meet the following primary requirements:

- to provide consistent directory information across multiple communication products;
- to support consistent mechanisms for interoperation;
- to assist applications in completing "all" types of communication tasks ;
- to promote an unified and uniform user administration; and
- to allow an easy integration of the directory with the rest of the customers' computer-based systems.

In addition to these, the architectural solution had also to meet a number of extra-functional requirements: increased overall performance, reliability and scalability, lower development costs, simplified serviceability, manageability, and maintainability.

7.4 The References: Directory Standards

More and more of the power of software development is expected to come from reuse of design knowledge codified in the form of patterns, guidelines and finally architectural standards. In respect to the directory, a number of international standards, emerging standards, and implemented services are available. They have created a foundation for developing an open interoperable directory system that can meet the objectives and requirements discussed in the previous section. Two standards have played an important role in respect to the solution addressed in our case study: ISO Directory Standard/ X.500 Recommendation, [26] and the Lightweight Directory Access Protocol (LDAP), [153].

The X.500 Directory Standard defines a number of models, operations and protocols. The models include: an information model, a namespace, a functional model, a security model and a distribution model. The defined operations are in three main areas: search and read, modify, and authenticate. The protocol that prescribes the allowed functionality is called Directory Access Protocol (DAP). The interactions between distributed directory servers is prescribed by three additional protocols: the Directory System Protocol (DSP), the Directory Operational Binding Management Protocol (DOP) and the Directory Information Shadowing Protocol (DISP), [26].

LDAP, the Lightweight Directory Access Protocol, [153], is an emerging directory standard, widely supported by the Internet community. Instead of requiring the "heavy" ISO OSI protocol stack it uses TCP/IP. Otherwise, it adopts the infor-

mation model and namespace from X.500. The LDAP functional model is a subset of the one defined by ISO X.500.

Initially, the focus of the Directory project was on an X.500 compliant solution[1]. Therefore, the specifications in this chapter are X.500/DAP-based and reflect the first version of the description. Later, the work shifted to an LDAP-based solution.

7.5 The Solution: An Architectural View

7.5.1 ECS Architectural Framework: The Context of the Directory System Architecture

The first step toward the solution was to define the context of the directory architecture and its influences on current and future product development and integration efforts. To do this, an initial architectural framework for the domain of Enterprise Communication Systems (ECS) has been identified with three primary roles: to foster common understanding and, hence, treatment of architectural issues across the domain of enterprise communication systems; to facilitate reuse and component-based development; and to create a basis for interoperation and sharing of services.

This framework is an evolving collection of architectural standards, specifications, rules, and agreements under the guidance and terms of which the domain collection of systems is planned, developed, integrated into flexible "customer offers," and maintained.

In the framework two major types of architectures are identified: (1) *product line architectures,* which are parameterized architectural solutions for families of products, for instance a messaging system architecture or a switching system architecture; and (2) *cross product architectures,* which introduce integration platforms and define shared or reusable components providing common services, such as security, transaction management, and directory.

[1]The project began before the wide acceptance of LDAP.

The Directory, as introduced in the previous sections, is as a repository of consistent, across multiple communication servers, clients and end-users, (directory) information. In the context of the ECS architectural framework, it is a server component that provides common (directory) services to multiple software components, systems and users.

7.5.2 Architectural Alternatives

Based on the requirements discussed in Section 3 and considering the issue of legacy systems, cost, and performance, three alternative architectural solutions were identified and examined.

1. *Alternative 1: Local Legacy Directory – Global United Directory*

   The first architectural alternative provides a common directory while at the same time protecting customers' investment and allowing a smooth transition from an environment of loosely related (legacy) products to integrated solutions. This alternative was considered as a near-term solution (NTS) which can be implemented quickly in order to improve the current situation and then expanded and evolved over time.

   According to the NTS architecture, the legacy communication servers remain unchanged and their administration can be handled independently. Together with the Directory server, the architecture introduces a new "Synchronizer" component which plays an important role in the integration strategy. It provides access to the directory and as its name suggests, the synchronizer is responsible for keeping the overall directory-related information in a consistent state across multiple communication servers. It acts as a gateway between the legacy products and the directory server, communicating with the communication servers in their native protocols and with the directory by using the standard DAP protocol (and in the latter versions LDAP).

2. *Alternative 2: No Local Directory – Global Standard Directory*

   The Directory server is the only entity to manage directory information and the only point of administration. The communication servers and applications interact with the directory server in real time in order to finish their tasks. In the solution described in this chapter, the X.500 DAP protocol is the vehicle to relay clients requests to the directory server.

3. *Alternative 3: New Local Directory (a Replica) – Global Standard Directory*

   The Directory server is the only point of directory information management and administration. Communication servers contain local copies (partial customized replicas – views) of the directory content. They interact with the directory server in order: (1) to refresh the local copy and (2) to provide extended services such as communications with users from different domains.

We have compared the three alternatives according to the following three groups of goals:

1. Customer Requirement Goals:

   - Single Point of Administration (SPA);
   - Consistent Directory Information (CDI);
   - Extended Services (ES)

2. Design Goals:

   - Avoidance of Single Point of Failure (ASPF);
   - Performance/Real-time communication (PF);
   - Data Integrity (DI);
   - Algorithmic Simplicity (AS)

3. Business Goals:

   - Low Cost of Legacy Product Management (LCLPM);
   - Low Cost of Additional Development Efforts (LCAD);
   - Cost Ratio

A brief overview of the comparison, the pros and cons for the three alternatives, and our conclusions are summarized in Figure 7.2. We see alternative 1 as favorable to the legacy products, near-term solution, applicable to small scale integration efforts. Alternative 3 was recommended as a long-term solution for large scale integrated product-offers built of new products. Alternative 2 is similar to 3 as it assumes that the communication servers and applications will interact with the directory server by using standard DAP (LDAP) protocol. This interaction, however, is required for every directory-related operation, which makes the architecture more suitable for low volume, low risk offers. Notice, that the global directory server does not imply a monolithic component. The directory can be a cluster, built of multiple Directory Service Agents (DSA), [26], distributed over multiple machines.

7.5.3 DirSA: The Selected Solution

Considering customer needs and business goals, the near-term solution, Alternative 1, has been selected initially as basis for the architectural work. In addition, however, the evolution of legacy products as well as new development efforts have also been taken into consideration. As a result an overall generalized architecture for integrating multiple legacy, modified and new communication servers and clients with an open real-time directory server has been defined. In this solution, the directory server is an autonomous component, compliant with ISO X.500 directory standard. It provides common directory services in the overall architecture, characterized as component-based, modular design.

Alternative \ Goals	Customer req. Goals	Design Goals	Business Goals	Conclusions
I. Alternative 1: Local Legacy Directory <--> Global United Directory Legacy Com. Products Directory Server Synchronizer *Constraints:* Data Synchronization	SPA: - CDI: + ES: -	ASPF: + PF: + DI: - AS: - -	LCLSM: + LCADE: - CR: +	Near term Legacy products Small scale
II. Alternative 2: No Local Directory <--> Global Standard Directory Com. Prod./Dir. Service Directory Server *Constraints:* Real-Time Commun. Prod.=Com. Serv.+Dir.Sever	SPA: ++ CDI: + ES: ++	ASPF: - PF: -- DI: + AS: ++	LCLSM: - LCADE: + CR: -	Like III but for Low volume Low risk
III. Alternative 3: New Local Dir.(a Replica) <--> Global Standard Directory New Com. Product Directory Server *Constraints:* Directory Replicas Synchronization	SPA: + CDI: + ES: +	ASPF: + PF: + DI: + AS: -	LCLSM: -- LCADE: + CR: -	Long term New products Large scale

SPA: Single Point of Admin.
CDI: Consistent Dir. Info
ES: Extended Services

ASPF: Avoidance of Single Point of Failure
PF: Performance
DI: Data Integrity
AS: Algorithmic Simplicity

LCLSM: Low Cost of Legacy System Management
LCADE: Low Cost of Additional Development Efforts
CR: Cost Ratio

(+) - a goal met by the arhitecture
(-) - a goal not (fully) met by the architecture

Figure 7.2 Architectural Alternatives: Comparison

The box-and-line diagram in Figure 7.3 provides an overview of the structure of the Directory System. It is an informal graphical illustration of the suggested architectural solution and serves as a basis for our discussion.

The directory system is decomposed into a collection of components, each of which is allocated a particular responsibility in the system. The components, drawn as boxes in the architectural diagram, are combined into a configuration via contracts, drawn as double-headed-arrows.

The core in our system is the Directory Data Server which provides support to a number of Clients – administrators, browser-based applications, new or modified

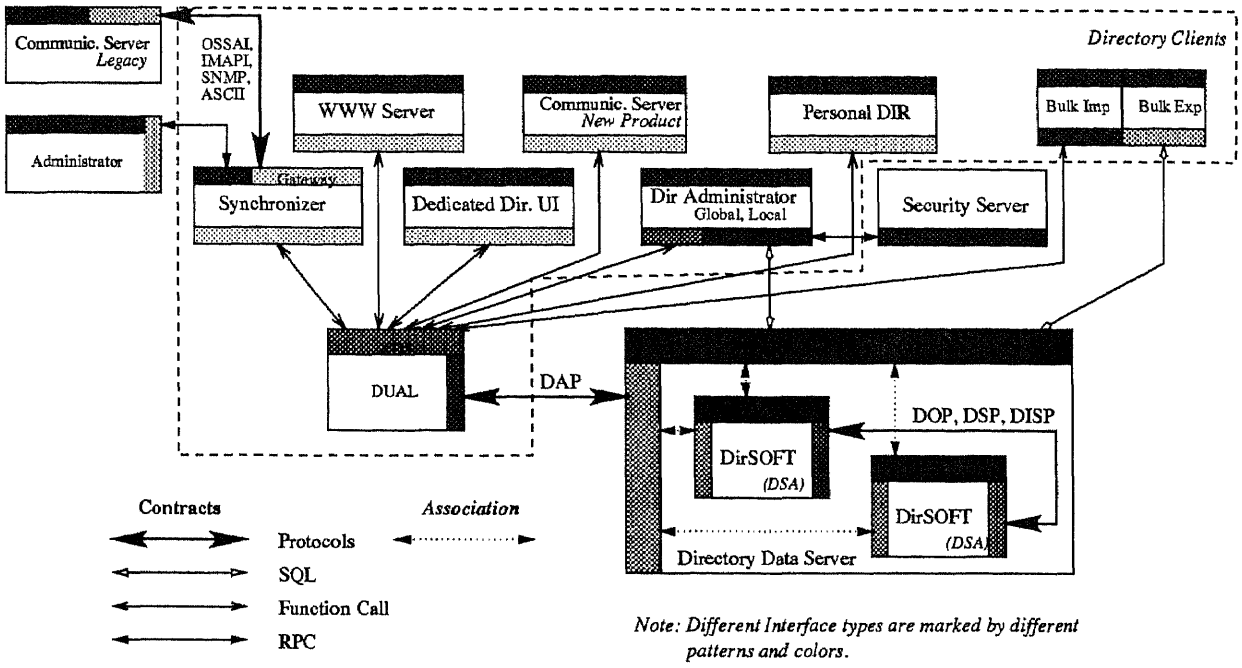


Figure 7.3 The Directory System Architecture: An Implementation View

communication systems (call control engines, messaging servers and clients) and auxiliary products (e.g., the synchronizer).

The interactions among the components represent a separate view of the solution. As shown in Figures 7.3 and 7.4, there are different interaction types involved in the system, i.e. "SQL," "RPC," and "DAP," which imply design and implementation constraints.

Specific architectural challenges in the solution were system scalability and extendibility. Ideally, the architecture should stay stable despite a significant increase in scale. For example, it was important to consider up-front at what point such a system have to evolve from a single directory data server (DirSOFT) to a cluster of such servers, see Figure 7.3. It was equally important to evaluate early-on what would be the cost to build in the cluster flexibility, and what the long-term benefits of such architectural solution are. The architecture described in the next section, defines the directory as a data capsule which could be implemented as a single or distributed directory. In regard to the implementation, based on analysis and evaluation of

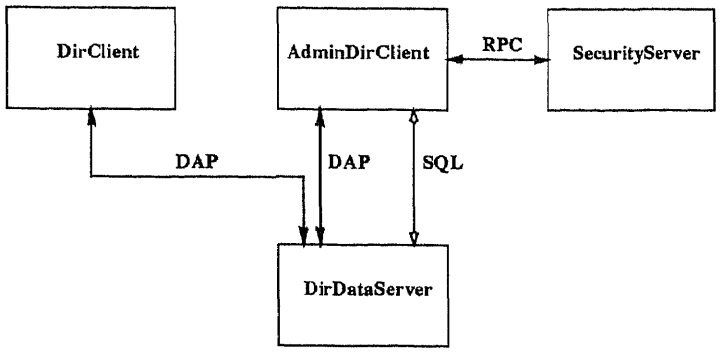


Figure 7.4 The Directory System Architecture: A Generic View

available off-the-shelf directory products, a decision was made initially to focus on a single directory server.

7.5.4 Applying GARM-ASPECT

GARM has provided guidelines to the process of abstracting architectural properties and the identification of architectural components and contracts. ASPECT has been used to write the DirSA specification (see Figure 7.5).

The architecture engineering process, as implied by the model, is a refinement process where more general architectural definitions are specialized and instantiated into specific system architectures. A general view of DirSA architecture is presented in Figure 7.4. The elements of architectural description, Figures 7.5, 7.6, are at the first two levels of refinement – the first level being a specialization form the generic element-types of the language, with the parent type attribute in their headers defined as "Generic" (see Figure 7.5).

The Directory Data Server, *DirDataServer,* is specified as a cluster component which could be implemented as a configuration of possibly distributed data servers, DirSOFT, compliant with the Directory Service Agent (DSA) specification of the ISO X.500 standard. The body of the directory server is a composition, *DirCluster,* whose internal structure is an architecture - *ScalDirArch* - composed of DirSOFT building blocks interoperating according to the rules of three protocols: DOP, DISP

```
DirSA : Architecture{
        Header{
                Type: {Generic}
                POF:{ } /* OCRA - BCS Cross Product RA */
                Implements:{ }
                }
        Components: {DirDataServer, DirClient, AdminDirClient, SecurityServer}
        Contracts: {DAP, SQL, RPC}
        Scenarios: {DirSAConfig}
        Rules: {DirSARules.tex}
        Description:{http://www.arch.lucent.com/Directory/DirSA-guidelines.html}
        }
DirDataServer : Cluster{
        Header{
                Type: {Generic}
                POF:{DirSA}
                Implements:{}
                }
        Interface: {DUI, DSQLI}
        Body: {DirCluster}
        }
DUI : Interface{
        Type: {Generic}
        Data_Model: {DInfoModel.txt}
        Data_Description: {ASN.1}
        Port: {DS_AP}
        }
DS_AP : Port {
        Type: {Generic}
        Port_attr{
        FA-&-DA: {DAPServiceProvide}
        BA: { ServerDAP.wright }
        QA: {SQualityControls.txt}
                }
        }
DSQLI : Interface{
        Type: {Generic}
        Data_Model: {DirSchema.txt}
        Data_Description: {}
        Port: {SQL_AP}
        }
SQL_AP : Port {
        Type: {Generic}
        Port_attr{
        FA-&-DA: {SQLServProvide}
        BA: {SQLServer.wright}
        QA: {}
                }
        }
SecurityServer : BuildingBlock{
        Header{
                Type: {Generic}
                POF:{DirSA}
                Implements:{}
                }
        Interface: {AdminRPCI}
        Body: {}
        }
AdminRPCI : Interface{
        Type: {Generic}
        Data_Model: {}
        Data_Description: {}
        Port: {RPC}
        }
RPC : Port {
        Type: {Generic}
        Port_attr{
        FA-&-DA: {RPCServiceProvide}
        BA: {RPC.wright}
        QA: {RPC.txt }
                }
        }
DirCluster : Composition{
        IntStructure: {ScalDirArch}
        Associations: {*}
        }
ScalDirArch : Architecture{*}

DirClient : Cluster{
        Header{
                Type: {Generic}
                POF:{DirSA}
                Implements:{}
                }
        Interface: {DUA_SRI}
        Body: {DirClientCluster}
        }
DUA_SRI : Interface{
        Type: {Generic}
        Data_Model: { }
        Data_Description: {ASN.1}
        Port: {DUA_RP}
        }
DUA_PR : Port {
        Type: {Generic}
        Port_attr{
        FA-&-DA:{DAPServiceRequest}
        BA: { ClientDAP.wright}
        QA: {ClQualityControls.txt}
                }
        }
ClientRPCI : Interface{
        Type: {Generic}
        Data_Model: { }
        Data_Description: {}
        Port: {RPCRequest}
        }
RPCRequest : Port {
        Type: {Generic}
        Port_attr{
        FA-&-DA: {RPCServReq}
        BA: {ClientRPCReq.wright}
        QA: {}
                }
        }
AdminDirClient : Cluster{
        Header{
                Type: {DirClient}
                POF:{DirSA}
                Implements:{}
                }
        Interface: {CSQLI, ClientRPCI}
        Body: {AdminDirClientCluster }
        }
CSQLI : Interface{
        Type: {Generic}
        Data_Model: { }
        Data_Description: {}
        Port: {CSQLP}
        }
CSQLP : Port {
        Type: {Generic}
        Port_attr{
        FA-&-DA:{SQLQuery}
        BA: {}
        QA: {}
                }
        }
DirClientCluster : Composition{
        IntStructure:{ClientImplArch}
        Associations: {*}
        }
ClientImplArch : Architecture{*}
```

Figure 7.5 DirSA Top Level Description: Architectures and Components

```
DAP : PContract{                                    SQL : PContract{
        Header{                                             Header{
                Type:{Generic}                                      Type:{Generic}
                POF: {DirSA}                                        POF: {DirSA}
                }                                                   }
        Data_Model:{ }                                      Data_Model:{ }
        Data_Description: {ASN.1}                           Data_Description: {}
        Role: {DAPRequest, DAPProvide}                      Role: {SQLQuery, SQLProvide}
        Liaison{                                            Liaison{
                Protocol:{DAP.wright}                               Protocol:{SQL.wright}
                CtBody:{ }                                          CtBody:{ }
                }                                                   }
        }                                                   }
RPC : PContract{
        Header{
                Type:{Generic}
                POF: {DirSA}
                }
        Data_Model:{ }
        Data_Description: {}
        Role: {RPCReqTransf, RPCDeliver}
        Liaison{
                Protocol:{RPC.wright}
                CtBody:{ }
                }
        }
DirSAConfig : Scenario{
Type: {Static}
Description: { ((DirDataServer.DUI.DS_AP, DAP.DAPProvide) (DirClient.DUA_SRI.DUA_PR, DAP.DAPRequest))
((DirDataServer.DUI.DS_AP, DAP.DAPProvide) (AdminDirClient.DUA_SRI.DUA_PR, DAP.DAPRequest))
((DirDataServer.DSQLI.SQL_AP, SQL.SQLProvide) (AdminDirClient.CSQLI.CSQLP, SQL.SQLQuery))
((SecurityServer.AdminRPCI.RPCProvide, RPC.RPCDeliver) (AdminDirClient.ClientRPCI.RPCRequest, RPC.RPCReqTransf))}
        }
```

Figure 7.6 DirSA Top Level Description (continued): Contracts and Scenarios

and DSP, [26]. The allocation of the information tree to the encapsulated data servers as well as the replication-shadowing mechanisms are according to the rules defined by the X.500 standard. The directory services are provided at the directory server's ports comprised by its interfaces: a DAP-based directory user interface *(DUI)* and an SQL interface *(DSQLI)*.

As illustrated in Figure 7.3, the Directory Data Server provides services to a set of Directory Clients. These Clients interoperate with the Directory Data Server following the DAP protocol. A specialized class of clients also have an SQL-based interface to the directory data. This SQL-based (non-DAP) interface to the directory has been provided for performance reasons. The *DirClient* component is a description of a generic client. The *AdminDirClient* component is a specialization from it. Any specific directory client can be derived from one of these component types. In theory the *DirClient* might be represented as a refinement of an abstract Directory User Agent (DUA), [26]. Practicality, however, prevailed and the actual

representation essentially captures an implementation relationship, that is, a cluster of a domain specific component (Personal DIR, WWW server, etc.) and a DUA library component (DUAL). The practical matter that most influenced this decision was the availability and popularity of off-the-shelf DUA libraries and common APIs (e.g., XDS, [12]).

The contracts, see Figures 7.4, 7.5 and 7.6, refer to the three types of interaction defined in the architecture: the *DAP* contract which represents the DAP protocol, the *SQL* which defines an SQL-based interaction with the directory data, and the *RPC* contract which defines an RPC-based interaction.

As shown in Figures 7.5 and 7.6, ASPECT templates have been used to represent the types of components, interactions and common structural patterns of the target architecture. The constituents of the description define element types that can be refined and extended in order to produce new definitions. They can be also further instantiated into specific elements and specific system architectures. For instance, the *DirClient* component discussed above is a component type. From this component type, multiple clients (see Fig. 7.3) can be derived and extended with additional interfaces, strictly observing the subtyping discipline of ASPECT (see Chapter 6). In this way, for example, the administrative directory client, *AdminDirClient,* was derived from *DirClient* and extended with the SQL-based interface, SQLI.

7.5.5 Usability of GARM-ASPECT Elements: A Summary

Throughout this chapter we have shown how to apply ASPECT constructs and have demonstrated the majority of their specific features. As summarized in Figures 7.7, 7.8, and 7.9 we have used architecture, component, interface, port, contract, and role element types to represent a generalized architectural solution, a reference architecture, which further served as an input in creating architectural instances.

Patterns, as introduced in GARM, were not used during the case study as we have worked top-down and have defined a general reference architecture. (The details of instance specification, the implementation phase of the development process, and the evolution of the architecture have been of no concern to the case study.) Guidelines, as defined in GARM, have been reflected as "rationale documents" linked to the architecture specification.

Construct	Feature Demonstrated	Examples
rchitecture		
Types and Instances	The architecture construct has been applied to define architecture types used to relate the elements of the directory architecture and to define the internal organization of the cluster components.	*DirSA, ScalDir Arch, ClientImplArch (Fig.7.5)*
Header	The header element has been used to capture the position of the architectures in specification hierarchies.	
Components	See Fig. 7.8.	
Contracts	See Fig. 7.9	
Scenarios	A static scenario has been used to capture the configuration of the systems to be developed according to the architecture.	*DirSAConfig (Fig.7.6)*
Rules	Rules have been used to informaly capture constraints on the architecture and their elements.	*DirSARules.txt (Fig.7.5)*
Description	Used to store "document descriptors," links - pointers to external documents annotating the architecture specifications.	*http://www.arch.lucent.com/Directory/DirSA-guidelines.html (Fig.7.5)*
atterns (GARM)	Patterns were not captured as they are, typically, collected bottom-up with the (re)-use of an architecture.	Not used at the time of the case study.
uidelines (GARM)	Guidelines are used to store design experience. See ***Description*** above.	*http://www.arch.lucent.com/Directory/DirSA-guidelines.html (Fig.7.5)*

Figure 7.7 Usability of GARM-ASPECT Elements/Constructs: Architecture, Pattern, Guideline

Construct	Feature Demonstrated	Examples
Component		
Cluster and Building Block	Cluster and Building Block types of components have been used to represent the composite and atomic components of the architecture.	Cluster: *DirDataServer, DirClient AdminDirClient (Fig.7.5)* BuildingBlock: *SecurityServer (Fig.7.5)*
Header	The header element has been used to capture the position of componets in specification hierarchies.	
Component Body	The body elements provided pointers to external properties of atomic components and to Composition specifications of cluster components.	
Types and Instances	Component types have been specified with the potential to define instances for the specific imlementations.	
Interface		
Types and Instances	Interface types have been defined.	*DUI, DSQLI, AdminRPCI, DUA_SRI ClientRPCI, CSQLI (Fig.7.5)*
Multiple Instances	Multiple interfaces per component have been defined allowing separation of component's obligations.	
Data_Models	The Data_Model element has been used to refer to the directory information models.	
Port		
Types and Instances	Port types have been defined.	*DS_AP, SQL_AP, RPC, DUA_PR RPCRequest, CSQLP (Fig. 7.5)*
Attributes	Port attributes have been used to name the functinality accessible at the port and to provide pointers to external specifications of behavior and quality constraints.	
Composition	Composition constructs capture alternative "implementations" of the body of a cluster component - the internal architecture and the associations between the internal and external ports.	*DirCluster, DirClientCluster (Fig.7.5)*

Figure 7.8 Usability of ASPECT Constructs: Component, Interface, Port, Composition

Construct	Feature Demonstrated	Examples
Contract		
Primitive and Composite	Primitive type of contracts have been used to define the allowed interaction types in the systems to be build according to the architecture.	PContract: *DAP, SQL, RPC* CContract: *Not Used*
Header	The header element has been used to capture the position of contracts in specification hierarchies.	
Liaison - Protocol	The protocol elements define pointers to external (formal) definitions of the protocols.	*RPC.wright, SQL.wright, DAP.wright*
Liaison - CtBody	The contracts are primitive, no body elements are defined (no external information is provided).	*Not Used*
Data_Model	No data-type connectors defined.	*Not Used*
Role		
Types and Instances	Role types have been defined.	*DAPRequest, DAPProvide; RPCReqTransfer, RPCDeliver; SQLQuery, SQLProvide (Fig. 7.6)*
Attributes	Role attributes have been used to prescribe the behavior of the ports that will play the role.	
CComposition	The contract composition (CComposition) construct has not been used: all specified contracts are primitive.	

Figure 7.9 Usability of ASPECT Construct: Contract, Role, CComposition

7.5.6 Alternative Approaches to Describing DirSA: A Review of Features

As we discussed in Chapter 4, recently there have been proposed a number of Architecture Description Languages. These languages constitute alternative choices for describing DirSA. In Fig. 7.10 we show a comparative summary of features of *Wright, UniCon, ACME*[2], and *ASPECT* ADLs, evaluated primarily from the case study requirements perspective.

By providing a direct realization of architectural abstractions as constructs in descriptive notations, all the ADLs shown in Fig. 7.10 permit architects to expose and define the organization of their systems. This provides a means of communicating more effectively for the purpose of evaluating the architecture, comparing alternatives, considering how the architecture might be adapted to another use, and for guiding the detailed design, implementation and integration of software systems.

All four languages provide notations for describing the structure of software systems in terms of hierarchical configurations of interconnected components; in this way creating an explicit and common basis for representing the structural view of a specific system architecture. All but UniCon support style representation and can serve as tools for representing the basic structure of DirSA. (The UniCon focus on specific system architectures was not in line with the project goals. Its strongest feature – providing tools for construction of executable systems – was not in the scope of the DirSA project objectives.) In addition to the component-port and connector-role structural representation scheme supported by the other languages, however, ASPECT supports an intermediate component interface template where multiple interfaces are allowed for a single component. This has allowed us to explicitly represent multiple functional views of DirSA components. In addition, ASPECT interfaces allowed us to capture the components' data view, observed at the specific

[2]ACME was not available at the time of the case study. The ADLs are new and evolving – the characteristics summarized in Fig. 7.10 represent their major features at the time of this write-up.

interface, which was of significant benefit when describing the *DirDataServer* (see Fig. 7.5).

Being focused on domain architectures, GARM-ASPECT also provides guidance for the transition between a conceptual and an application architecture; ASPECT supports allocation of domain-specific functions to architectural components (port's functional attributes). The ACME port property list could non-explicitly serve this purpose as well.

For representing system behavior and performing analysis, Wright is the only language from the group providing native support. ASPECT structural specifications can be extended with behavioral specifications in Wright (see Fig. 7.5).

ASPECT and ACME (in its latest version) provide support for refinement of architectural elements through subtyping mechanisms. This has allowed us to start with a general view of the architecture (see Fig. 7.4) and to develop the specification in a stepwise manner as more details were unveiled.

All four ADLs support hierarchical decomposition (decomposition-aggregation abstraction). Through the use of large scale constructs and the encapsulation of different parts of the system they permit the architects to select an appropriate level of granularity at which the system will be described and to hide the unnecessary details, for example of commercial of the shelf components (COTS).

7.6 Summary and Evaluation

In this chapter we have shown through the example of the DirSA prototype architecture, that GARM-ASPECT can be used to guide the architecture engineering process and to provide specifications of a practical architectural solution. The evaluation process and the specification have illuminated a number of aspects and issues that were not addressed by the project before the application of the method.

In the case study, we defined the scope of the architecture and identified the levels of abstraction at which architectural issues are to be addressed. We also identified and compared several alternative solutions to the "directory problem". Finally, we generalized the solution and described an architecture which can be refined into architectures that support alternative implementations (see the alternatives discussed in section 7.5.2). In this way an architectural specification becomes cost effective through its amortization across multiple systems.

Applying GARM-ASPECT we have underlined a number of important methodological aspects of abstract description of software architectures:

- separating interaction from computation as component and contract types;
- identifying and specifying all component interfaces; and
- separating different interaction types as different contracts.

Although we have concentrated on describing DirSA, an architecture for a well defined functional domain, the formalization of component, interface, and contract types has permitted us to reason about more general conceptual solutions expressed in this vocabulary. The constraints and alternatives identified and described have built a basis for developing a more successful solution.

The primary benefits of applying the method have been: 1) the concise specification of architectural solution which, being considerably more abstract than the actual functional view of the components in the system, provides a basis for reasoning about the solution and, at the same time, can be used as input for further refinement; and 2) the systematic approach and the availability of a common language used by project members, which helped to improve the engineering process by creating a basis for more easy integration of different portions of the work and comparison of identified alternative solutions.

ASPECT is a language for describing architectural structure - architecture elements and configurations (including dynamic configurations). It provides placeholders for external specification of behavior. In order to build a more complete specification of the three alternatives, discussed in the case study, the specifications in ASPECT have to be extended with auxiliary descriptions. Selecting an appropriate language for describing the behavioral aspects of architectural elements and using it consistently throughout the specification is an obvious recommendation/requirement for a good style.

The method has been evaluated as practical and applicable to capture results of architectural work at several levels: from building high-level architectural prescriptions, to representing structural properties of product line architectures, to capturing information on reusable assets – components and protocols. In its current implementation, however, ASPECT is not oriented toward low-level design or code generation.

The ability to store references to information models as part of interface descriptions has shown to be very useful for the specific project. The mechanism for extending the views by attaching external formal and informal specifications has been highly appreciated. The responsibility of the core structural view to put specifications and documents together and into the right perspective was important in documenting the overall architecture. Flexibility and finding the right balance between formal and informal specifications, and producing an overall specification that corresponds to the scope and the role of the architecture being specified, are major practical issues in representing architectural designs which GARM-ASPECT has also successfully addressed.

The case study once again confirmed that specifying architectures in textual forms is a cumbersome and ineffective process and that tool support is highly desirable.

	Language			
Characteristics	***Wright***	***UniCon***	***ACME***	***ASPECT***
Representing architectural structure	Architectural instances and style definition. Component and connector types, instances and attachments.	Component and connector types, instances and connections. No means of describing families of systems.	Architectural instances and style definition. Component and connector types, instances and attachments.	Reference architectures/ family architectures (see *DirSA*). Component, contract, interface and port types, subtypes and instances. Static and dynamic scenarios.
Representing behavior	CSP-based processes specifying patterns of behavior for ports, roles, connector glues and component computations - native.	Through element implementation but limited to a set of build-in atomic types or through external specifications associated through property lists.	Through un-interpreted external specifications associated through property lists.	Pointers to external specifications (e.g. in Wright) - external.
Representing multiple functional views				Multiple interfaces (see, for example, *AdminDirClient*).
Representing system's data aspect		A possibility to attach data models to a component as a whole through property lists.	A possibility to attach data models to a component as a whole through property lists.	An explicit data view of interfaces - data models, data definitions (see the *DUI* interface of *DirDataServer* component).
Analysis	Structural consistency checking. Behavioral analysis: port-computa-tion consistency, connector and role deadlock-free, port-role compatibility, etc.	Structural consistency checking	Structural consistency checking	Structural consistency checking
Decomposition aggregation hierarchies	Supported	Supported	Supported	Supported
Generalization specialization hierarchies			Refinement of elements (added recently).	Refinement of architectural elements.
Allocation of domain-specific functionality		Property lists could be used.	Property lists could be used.	Explicit transition between conceptual and application architecture - defining functional attributes of ports.
Remarks, strong features	An ADL based on formal description of abstract behavior of components and connectors. Extensive analysis support.	Focus on specific system architecture. Provides a configuration tool for constructing executable configurations.	Introduced in 1997. Was not an available choice at the time of the case study. Developed as an interchange language.	Focus on reference architecture and architectural frameworks. Balance between formal and informal representations.

Figure 7.10 Alternative ADL Choices

CHAPTER 8

CONCLUSIONS

8.1 Summary

The primary focus of this dissertation is on developing methods and tools to support domain architecture engineering and on leveraging architectures to achieve improved system development and integration in presence of increased complexity. In particular, the thesis explores issues related to the following three aspects of software technology: (1) system complexity and software architectures as tools to alleviate complexity; (2) domain architectures as frameworks for construction of large scale flexible enterprise-wide software systems; (3) architectural models and representation techniques – a basis for "good" architectural design.

> *A "good" architecture is a product of both methodology and good architectural design. Methodology defines the units of architectural abstraction and the system of specification. A good methodology allows important architectural properties like modularity and consistency, but it does not necessitate them: good use of the methodology – i.e., good design and process – is required as well.*

In the thesis we, first, explored the issue of system complexity and contributed to the justification of an architecture-based approach to system development. We identified a number of factors contributing to the complexity of software systems and focused on those of major concern when developing and evolving domain-specific, integrated, systems-of-systems. We made the argument that the complexity of such systems requires new means of coordination and standardization in their development and integration, and that one natural solution to the conceptual and practical aspects of the problem is to superimpose a well-defined, rigorously described, commonly accepted, and strictly applied high-level design framework, called *domain architecture* (see Chapter 2).

Next, we focused on characterizing architectures, and on building a sound understanding of software architectures and domain architectures, in particular.

Throughout the characterization process we have compared and classified multiple architectures according to their content, scope, and role in the development process. Consequently, we have defined a two-dimensional taxonomy in which a distinction is made between reference architectures and particular system architectures, as well as between conceptual (property-specific) and application (problem-specific) architectures or models. In the context of this classification we defined domain architectures as domain-specific reference architectures which comprise both conceptual as well as application models. We further discussed the important steps of a domain architecture engineering and its role as a means of building composable systems and large scale reuse (see Chapter 3). Related work is surveyed in Chapter 4.

The most prominent contribution of this dissertation is the development of GARM-ASPECT – a method for (domain) architecture engineering and representation. (Chapter 5 provides an introduction to the method.) We, first, defined the underlying architecture model – GARM. The model is a system of architecture modeling abstractions. It includes a set of modeling concepts, constructive and property related, as well as a rule system for expressing architectural constraints. It also suggests extending the description of a reference architecture with patterns and guidelines. Derived from GARM is ASPECT, a software architecture description language. We defined ASPECT's syntax, informal semantics, and important features. ASPECT provides a template-based notation for representing structural view of software architectures. It has an open semantics and allows informal and formal external specifications to be combined with the core structural descriptions into an overall architectural specification.

Generalization-specialization abstraction is fundamental to our approach to architecture modeling and representation. The refinement process supported by GARM-ASPECT method and the details of ASPECT subtyping discipline are

presented in Chapter 6. We define the semantics of architecture element types, subtypes and instances.

The applicability and practical value of GARM-ASPECT method has been tested by employing it in an industrial project. The case study demonstrated the utility of precise definition of architectural configurations and showed how the abstract description of architectures exposes critical issues and contributes to the effective consideration of architectural alternatives. The study also demonstrated how the model itself influences the architecture engineering process by defining what shall be the constituents of an architectural description and by establishing relations and orders between them. The results from the case study are summarized in Chapter 7.

To support the application of GARM-ASPECT method, we have developed tool-prototypes for architecture engineering (see Appendix C). Recently, a GARM-ASPECT-based tool, ArchE, has been developed at Lucent Technologies Inc. This tool provides a web-based user interface and facilities for storing and consistency checking of hierarchies of architectural descriptions. It serves as a "core ASPECT editor" which also supports links with external, domain-specific tools. This integrated tool set has been the main GARM-ASPECT testbench. (Figure 8.1 provides an illustration of the most important elements of the research presented in this dissertation.)

8.2 Future Work

While GARM-ASPECT represents a significant benefit for software architects, it also suggests a number of areas for future work.

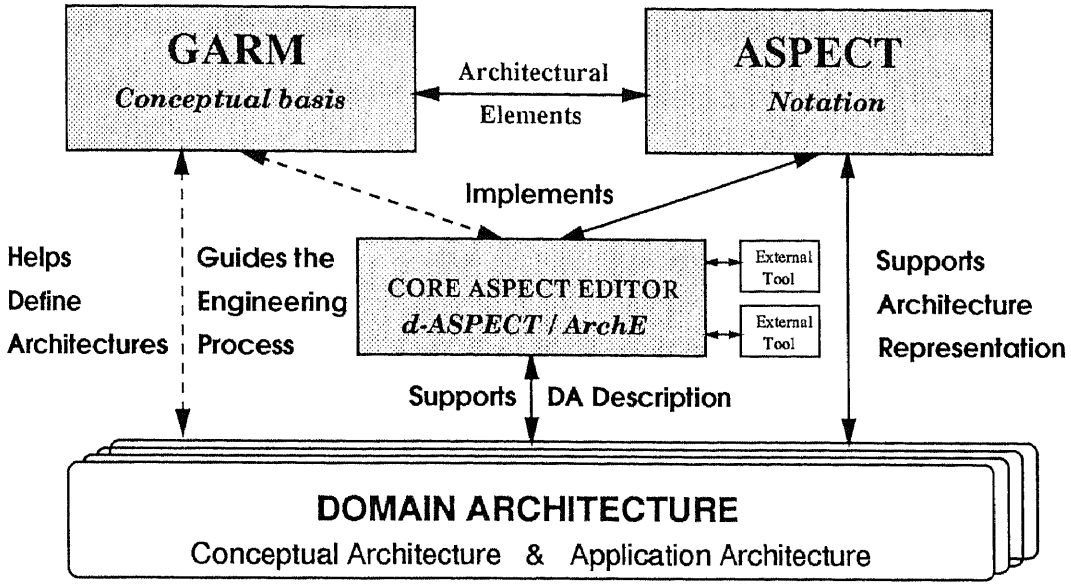


Figure 8.1 GARM – ASPECT – *d*-ASPECT

8.2.1 Near Term Objectives

In near term we intend to focus on applying the ASPECT language and extending the set of resources offered to ASPECT users.

- Experience with GARM-ASPECT

  First, we plan to gain more experience with the use of GARM-ASPECT by further applying it to realistic problems. We also intend to apply the method to different types of architectural efforts and to evaluate the scope of its practical applicability.

  In this respect, we are currently extending the work on the DirSA case study (see Chapter 7) and further applying GARM-ASPECT for representing the elements of an architectural asset repository, [74]. Also, we are using the method as a means of support in development of an intelligent design environment. We further intend to use the method for representing an architectural framework for construction of enterprise communication systems. This

kind of use will provide feedback on how the method can be evolved to best suit the needs of architects in specific domains.

- Improving the usability of ASPECT and its Editors

  An important extension to our architecture engineering tool set is to build a library of (domain-specific) patterns, protocols and components. More experience with the addressed domains, however, will be needed in order to build this extension.

  Another important extension is to develop graphical primitives for user-friendly representation of architectural configurations.

- More tools integration

  ASPECT supports the representation of architectural structures and the building of hierarchies of architectural descriptions. It also provides facilities to establish relations between different specifications and to pull them together into an overall description. We plan to further develop this aspect and to integrate additional tools with the ASPECT editors, in this way creating integrated (possibly domain-specific) architectural environments. Our current work is on integrating an UML/OCL parser in our environments in order to automate the processing of ASPECT rules.

8.2.2 Long Term Plans

Beyond using and improving ASPECT there are several avenues of long term research suggested by the results of this dissertation. They include the following: extending the architectural model and language to address additional system aspects and properties; exploring ways in which formal semantic models can be integrated with ASPECT; automating the construction of software based on architectural specifications; searching for the right balance between the formal and informal elements

of an architectural specification based on stakeholder profiles, and investigating the implications of architecture technology for software process and organizational culture.

8.3 Epilogue: The Promise of Architecture-Based System Development and Integration

Throughout this dissertation, we have taken as a premise that explicitly applying an architectural approach to software development, integration, reuse, and evolution is the right thing to do. As we have discussed, there is a considerable body of research and practical effort in developing and applying architectural technology.

Since its emergence in the late 1980's as an explicitly identified area of software engineering, software architecture has held out considerable promise for an improved software construction. By providing modeling techniques and notations for creating explicit, analyzable representations of the organization of complex software systems, by codifying architectural patterns and elements, by creating architectural standards, by promoting the independence of reusable system elements and increasing the opportunities to compose them into flexible systems, by focusing on system evolution and management of change, by providing support for cost-effective development of families of systems, the software architecture field seems to present a promising direction in software technology.

However, many of these benefits are yet to be realized. While there have been a number of successes such as improved planning, based on an established architectural vision; improved productivity as result of incorporating the architecture in the development process; or improved system quality, based on reuse of architectural patterns, the most expansive vision of an architecture as a unified framework for improved development, cost-effective reuse, instant integration of

separately developed or purchased off-the-shelf components, and planned evolution remains a future target.

By itself GARM-ASPECT is not intended to achieve the full promise of an architecture-based system construction. However, the method, together with the deeper understanding of different types of architectures and their purposes, provides a more solid basis for exploring the applicability of the abstractions used to define software architectures, the practicality of the representation approach, and the required changes to the development process. In this respect, this work stands as a step toward those larger goals for architecture-driven software construction.

APPENDIX A

GARM-ASPECT CONCEPTS AND CONSTRUCTS: A SUMMARY

This appendix provides a concise summary of the elements of GARM and ASPECT. The Figures 5.7, 5.8 and 5.9, included below, relate the basic concepts defined in GARM to the constructs provided in ASPECT.

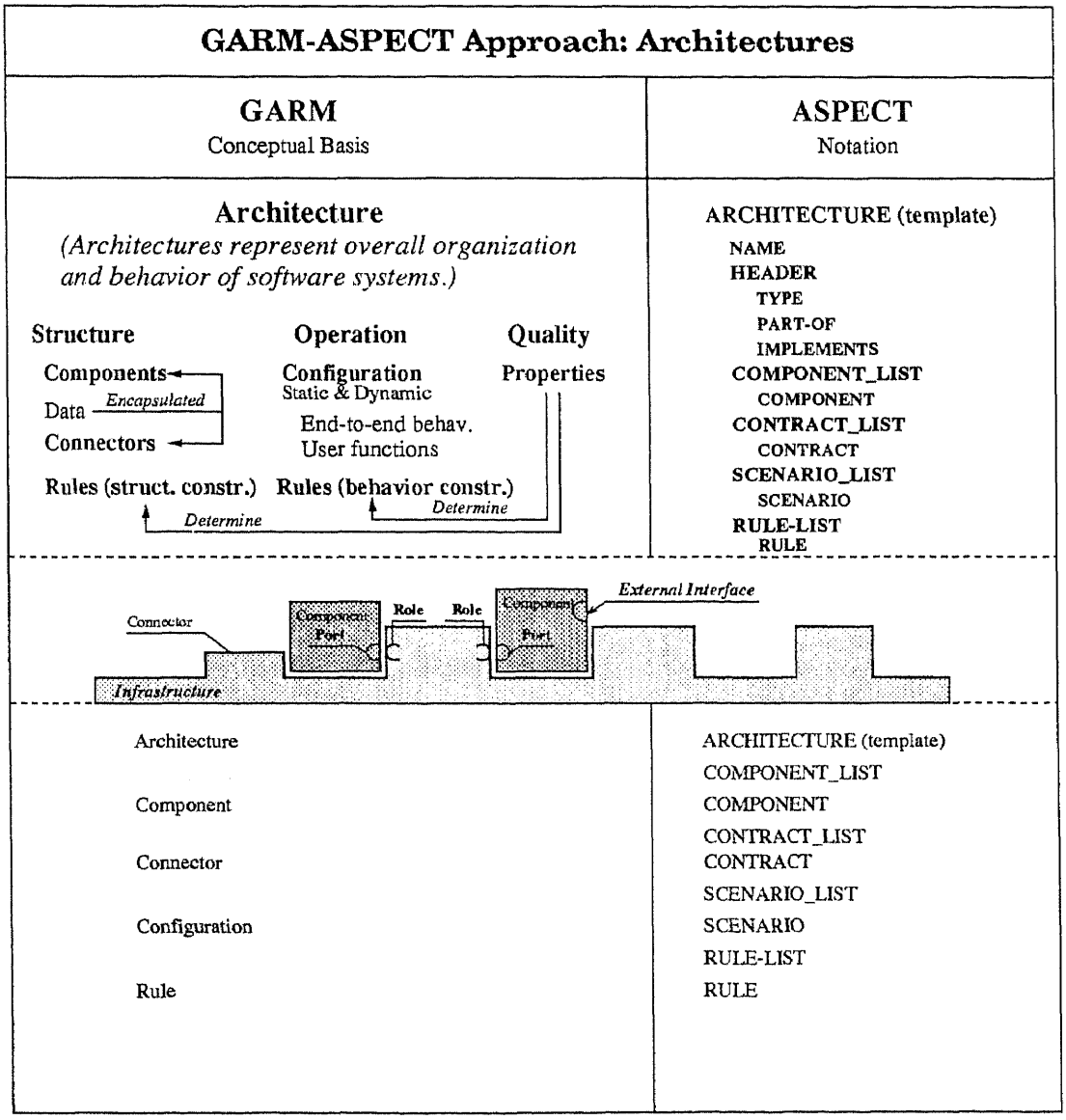


Figure A.1 Architectures in GARM and ASPECT

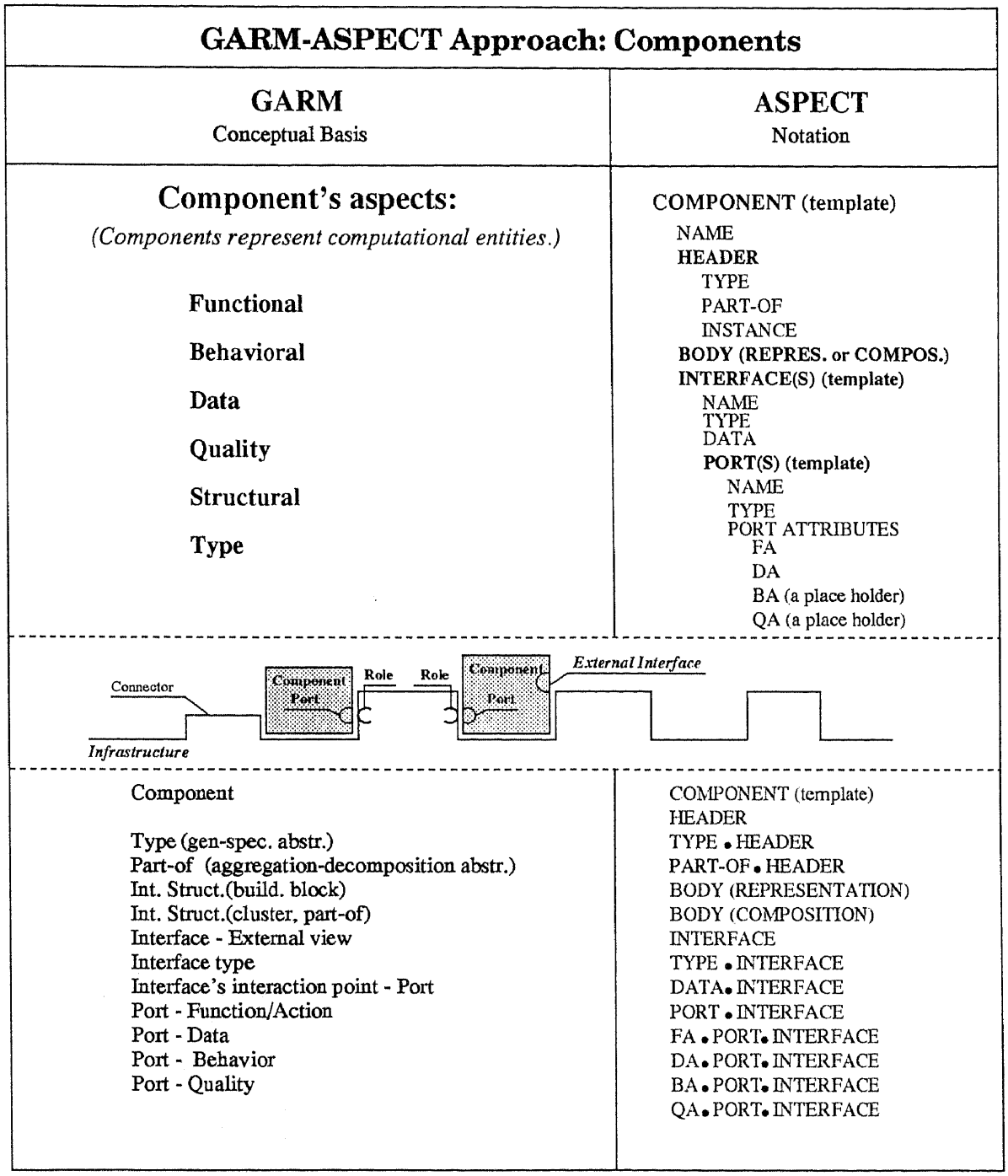
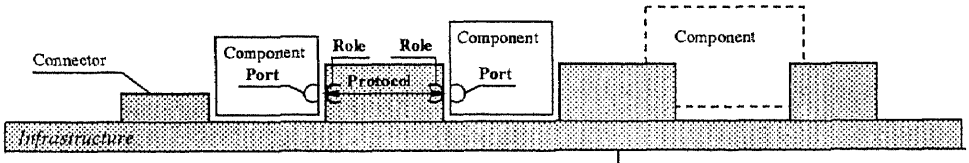


Figure A.2 Components in GARM and ASPECT

GARM-ASPECT Approach: Connectors

GARM Conceptual Basis	ASPECT Notation
Connector's aspects: *(Connectors represent interactions among components.)* **Functional** **Behavioral** **Data** **Quality** **Structural** **Type**	CONTRACT (template) NAME **HEADER** TYPE PART-OF INSTANCE **DATA** **LIAISON** PROTOCOL (a place holder) BODY (REPR. OR COMPOSIT.) **ROLE(S)** NAME TYPE ROLE ATTRIBUTES FA DA BA (a place holder) QA (a place holder)



Connector	CONTRACT (template) HEADER
Type (gen-spec. abstr.) Part-of	TYPE • HEADER PART-OF • HEADER
	LIAISON
Int. Struct.(atomic) Int. Struct.(composite, part-of)	BODY (REPRESENTATION) BODY (COMPOSITION)
Behavior aspect of an inter. - protocol	PROTOCOL
The Role played by a Port in the interaction context	ROLE
Role type	TYPE• ROLE
Functional aspect of a role-port Data types def. in the cont. of the role-port Behavior of a role-port Quality of a role-port	FA • ROLE DA • ROLE BA • ROLE QA • ROLE

Figure A.3 GARM Connectors and ASPECT Contracts

APPENDIX B

ASPECT ABSTRACT GRAMMAR

The abstract grammar, presented in this Appendix, formally specifies the relationships between the ASPECT architectural elements and the structural properties of the representation templates. The BNF specifies the core ASPECT language. It, however, does not define any concrete syntax.

Extended BNF Meta-Syntax ([96]) :

[...|...|...] indicates alternatives

{...}* denotes zero or more iterations

{...}+ indicates one or more repetitions

; indicates sequence

"..." indicates a literal string

Note also the following:

(1) a numeric superscript followed by a plus sign ($^{n+}$) indicates the minimum required repetitions;

(2) the braces can be omitted if a single element is being repeated;

(3) a text enclosed between the /* and */ is a comment; and

(4) terminals (other than punctuation and keywords) are in capital letters.

ASPECT Grammar:

The constructs and productions of the ASPECT grammar are as follows[1]:

Architecture ⇒ *name* : *Arch_name*; *header* : *Architecture_header*;
component : *Component_list*; *contract* : *Contract_list*;
scenario : *Scenario_list*; *rule* : *Rule_list*

[1]Note: anywhere a structured entity is expected, it is legal to use simply its name or a (meta) variable, with the rest of the structure being defined in a separate declaration.

$Arch\_name \Rightarrow ANAME$
$Architecture\_header \Rightarrow type : Arch\_type\_list;\ \ part{-}of : Arch\_POF\_list$
$\quad implements : Arch\_Instance\_id$
$Arch\_type\_list \Rightarrow Arch\_name^{+}$
$Arch\_POF\_list \Rightarrow Composition\_name^{*}$
$Arch\_Instance\_id \Rightarrow [Arch\_type | \text{“}Generic\text{”}]$
$Component\_list \Rightarrow Component^{+}$
$Contract\_list \Rightarrow Contract^{+}$
$Scenario\_list \Rightarrow Scenario^{*}$
$Rule\_list \Rightarrow Rule^{*}$
$Rule \Rightarrow RULE\_TEXT$

$Component \Rightarrow name : Component\_name;\ \ header : Comp\_header;$
$\quad interface : Interface\_list;\ \ body : Body\_list$
$Component\_name \Rightarrow CNAME$
$Comp\_Header \Rightarrow type : Comp\_type\_list;\ \ part - of : Comp\_POF\_list;$
$\quad org\_type : Comp\_org;\ \ implements : Comp\_Instance\_id$
$Comp\_type\_list \Rightarrow Comp\_name^{+}$
$Comp\_POF\_list \Rightarrow Architecture\_name^{+}$
$Comp\_org \Rightarrow [\text{“}BUILDING\_BLOCK\text{”} | \text{“}CLUSTER\text{”}]$
$Comp\_Instance\_id \Rightarrow [Comp\_type | \text{“}Generic\text{”}]$
$Interface\_list \Rightarrow Interface^{+}$
$Interface \Rightarrow name : Int\_name;\ \ type : Int\_type\_list;$
$\quad compatible\_with : Compat\_Int\_name\_list;\ \ data\_model : Data\_model;$
$\quad data\_definition : Data\_definition\_list;\ \ port : Port\_list$
$Int\_name \Rightarrow INAME$
$Compat\_Int\_name\_list \Rightarrow Int\_name^{*}$
$Int\_type\_list \Rightarrow Int\_name^{+}$
$Data\_definition\_list \Rightarrow Data\_definition^{*}$
$Data\_definition \Rightarrow DATA\_TYPE\_DEFINITION$
$Data\_model \Rightarrow DATA\_MODEL$
$Port\_list \Rightarrow Port^{+}$
$Port \Rightarrow name : Port\_name;\ \ type : Port\_type;\ \ attribute : Port\_attr$
$Port\_name \Rightarrow PNAME$
$Port\_type \Rightarrow Port\_name^{+}$
$Port\_attr \Rightarrow f\_attr : Function\_list;\ \ d\_attr : Data\_list;$
$\quad b\_attr : Behavior;\ \ q\_attr : Quality\_attr\_list$
$Function\_list \Rightarrow Function^{+}$
$Function \Rightarrow FUNCTION\_SIGNATURE$
$Data\_list \Rightarrow Data\_parameter^{*}$
$Data\_parameter \Rightarrow DATA\_DEFINITION$
$Behavior \Rightarrow BEHAVIORAL\_SPEC$
$\quad /{*}EXTERNAL\_FORMAL\_SPEC;\ \ (ACR){*}/$

$Quality\_attr\_list \Rightarrow Quality\_attribute^{*}$
$Quality\_attribute \Rightarrow QUALITY\_SPEC$
$/*EXTERNAL\_TEXT \mid EXTERNAL\_FORMAL\_SPEC*/$
$Body\_list \Rightarrow Body^{*}$
$Body \Rightarrow Representation|Composition$
$/*If\ Comp\_organ = \text{“}BUILDING\_BLOCK\text{”}$
$then\ the\ Body\ is - a\ REPRESENTATION;$
$else\ the\ Body\ is - a\ COMPOSITION.*/$
$Representation \Rightarrow REPRESENTATION$
$Composition \Rightarrow name : Composition\_name;\ \ int\_structure : Architecture;$
$association : PAssociation\_list$
$Composition\_name \Rightarrow COMPOSNAME$
$PAssociation\_list \Rightarrow PAssociation^{+}$
$PAssociation \Rightarrow internal\_port : IPort\_name\_list;$
$external\_port : Port\_name$
$IPort\_name\_list \Rightarrow Port\_name^{+}$

$Contract \Rightarrow name : Contract\_name;\ \ header : Contract\_header;$
$liaison : Liaison;\ \ role : Role\_list;\ \ data\_model : Data\_model;$
$data\_definition : Data\_definition\_list$
$Contract\_name \Rightarrow CTNAME$
$Contract\_header \Rightarrow type : Ct\_type;\ \ part - of : CtPOF\_list;$
$ctorg\_type : Ct\_org;\ \ implements : Ct_Instance\_id$
$Ct\_type \Rightarrow Contract\_name^{+}$
$CtPOF\_list \Rightarrow Contract\_name^{+}$
$Ct\_org \Rightarrow [\text{“}PRIMITIVE\text{”}|\text{“}COMPOSITE\text{”}]$
$Ct\_Instance\_id \Rightarrow [Ct\_type|\text{“}Generic\text{”}]$
$Role\_list \Rightarrow Role^{2+}$
$Role \Rightarrow name : Role\_name;\ \ type : Role\_type;\ \ attribute : Role\_attr$
$Role\_name \Rightarrow RNAME$
$Role\_type \Rightarrow Role\_name^{+}$
$Role\_attr \Rightarrow f\_attr : Function\_list;\ \ d\_attr : Data\_list;$
$b\_attr : Behavior;\ \ q\_attr : Quality\_attr\_list$
$Liaison \Rightarrow protocol : Protocol;\ \ ct\_body : CtBody\_list$
$Protocol \Rightarrow PROTOCOL\_SPEC$
$/*EXTERNAL\_TEXT \mid EXTERNAL\_FORMAL\_SPEC*/$
$CtBody\_list \Rightarrow CtBody^{*}$
$/*If\ Ct\_organ = \text{“}Primitive\text{”}\ then\ the\ CtBody\_list\ is\ empty$
$or\ an\ external\ REPRESENTATION;$
$else\ CtBody\ is\ a\ contract\ composition\ -\ CtComposition.*/$
$CtBody \Rightarrow Representation|CtComposition$
$CtComposition \Rightarrow name : CtComposition\_name;$
$ctint\_structure : Architecture;\ \ ct\_association : RAssociation\_list$

$CtComposition\_name \Rightarrow CTCOMPOSNAME$
$RAssociation\_list \Rightarrow RAssociation^{+}$
$RAssociation \Rightarrow internal\_role : IRole\_name\_list;$
$\quad external\_role : Role\_name$
$IRole\_name\_list \Rightarrow Role\_name^{+}$

$Scenario \Rightarrow type : Scenario\_type;\ \ description : Scenario\_description;$
$Scenario\_type \Rightarrow [\text{“}STATIC\text{”}|\text{“}DYNAMIC\text{”}]$
$Scenario\_description \Rightarrow Port\_Role\_list$
$Port\_Role\_list \Rightarrow Port\_Role^{+}$
$Port\_Role \Rightarrow port : Port\_name\_list; role : Role\_name$
$Port\_name\_list \Rightarrow Port\_name^{+}$

APPENDIX C

GARM-ASPECT: TOOL SUPPORT

In this thesis we have introduced a method for architecture engineering and representation and have demonstrated how to apply it to a real-world problems in a case study. In order to alleviate the complexity of an architecture specification process and to provide automated support for the manipulation of architectural specifications, we have developed prototype tools supporting the method. The implementations have provided us with an environment for experiments with the notation and its application in different contexts; they have further created a base for evaluating and extending the notation, for evaluating the model, and for validating the method. The tool support for GARM-ASPECT has evolved as follows.

C.1 *d*-ASPECT: An Overview

d-ASPECT is a common name for two prototypes developed at NJIT to serve as a testbed for our research on systems integration and domain architecture engineering and specification.

C.1.1 *d*-ASPECT (V1)

The first tool prototype, *d*-ASPECT (V1) was designed as a multi-aspect environment for architecture-based system development and integration. The tool's architectural design is based on client-server style and uses a central data repository for data exchange among the servers, see Fig. C.1. To implement the prototype we used the development and runtime environment provided by ANSAware 4.0 [11] – an infrastructure supporting the ISO ODP architectural standards, [1]. Therefore, the prototype was an ANSA-based open distributed system and the implemented components operated on an ANSA bus. The implementation included: an application architecture server (and a database), and a client providing Graphical User

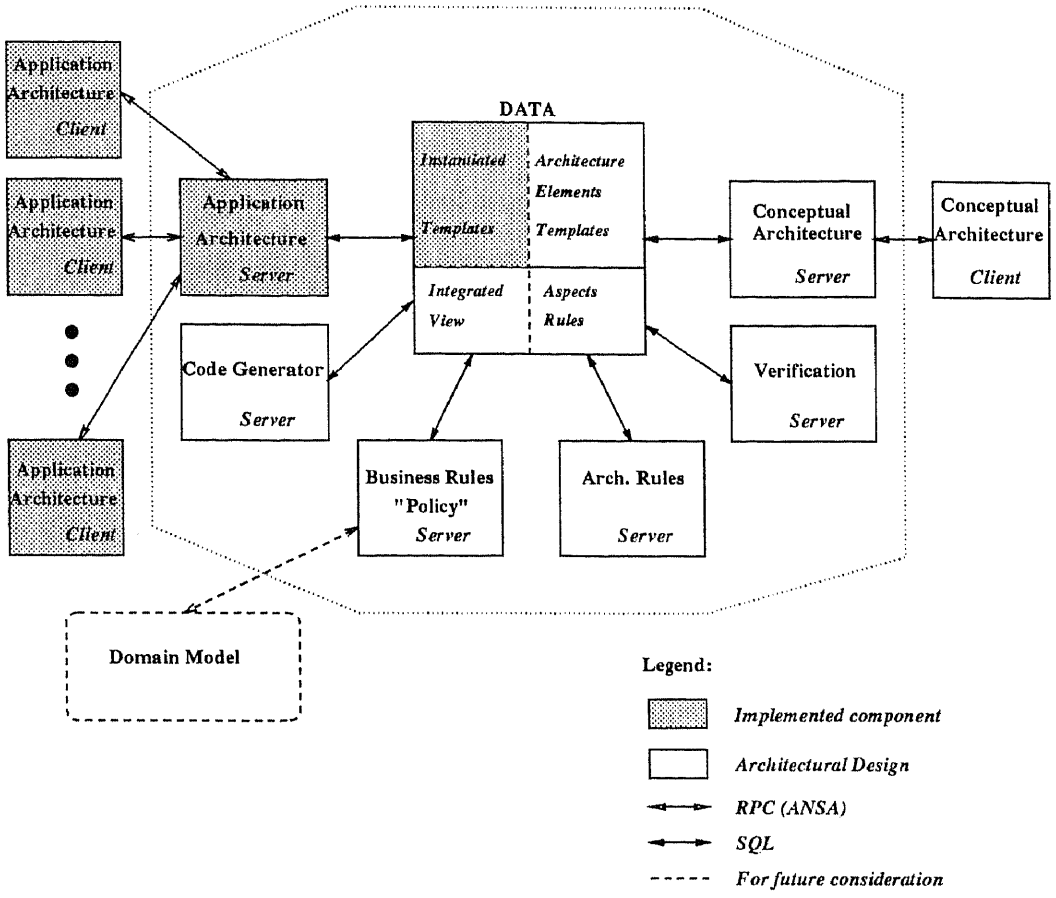


Figure C.1 *d*-ASPECT (V1): Architecture

Interface (GUI) for application architecture specification, see Fig. C.2. The architectural design also included additional elements such as components for conceptual architecture specification, domain analysis support, and rule-constraint processing. More information on this prototype can be found in [71].

C.1.2 *d*-ASPECT (V2)

In the second version of *d*-ASPECT (V2), currently under development at NJIT, we have narrowed the scope of the prototyping effort and have exclusively focused on architecture specification and on providing support for GARM-ASPECT. Thus, *d*-ASPECT (V2) is an ASPECT editor, designed to support specification of architecture element types and instances, rules and scenarios as defined in GARM-

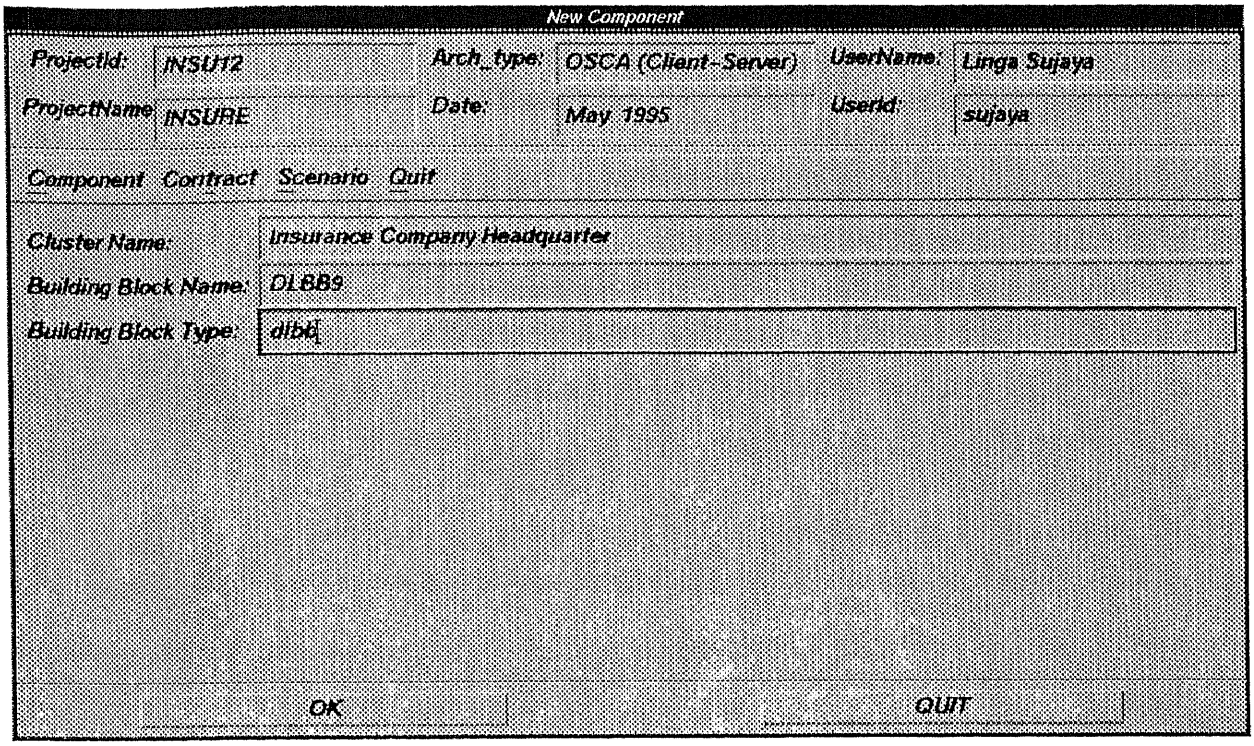


Figure C.2 *d*-ASPECT (V1) GUI: Application Architecture Component

ASPECT. Its main purpose is to serve as a test-environment for our work on the representation model and the language. It provides graphical templates for manipulating the architectural elements, and supports the refinement process presented in Chapter 6. The tool provides a GUI, implemented in *Motif* [129], and an ORACLE database at the back-end for storing architectural descriptions, [78]. Figures C.3, C.4, C.5, C.6 and C.7 illustrate certain aspects of this prototype. A web-based extension of the tool for visualizing the decomposition and refinement hierarchies is currently under development.

C.2 ArchE: An Overview

ArchE is an ASPECT editor, developed as a proprietary tool at Lucent Technologies. Its design and implementation followed the work on DirSA case study, discussed in Chapter 7.

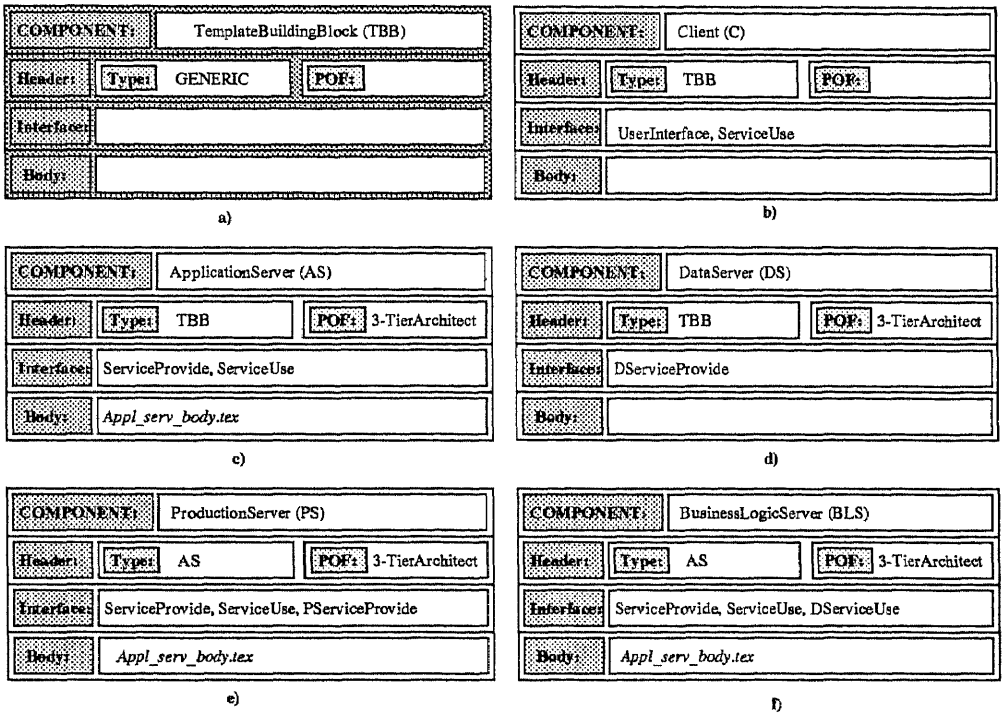


Figure C.3 *d*-ASPECT (V2) GUI: Example Components

ArchE provides a web-based user interface and a database component for definition, storing, search, visualization, manipulation, and annotation of architectural descriptions. The architectural elements of ASPECT are supported as graphical templates (forms) for storing and modifying architectural descriptions, see Fig. C.8, C.9. The tool also supports the decomposition-aggregation of components and connectors, hence, the specification of atomic and cluster components and primitive and composite connectors. Name consistency checking is supported in compliance to the ASPECT naming rules discussed in Chapter 6. Specification of scenarios is also implemented. Scenarios are represented as steps (or sequences of steps for dynamic scenarios) of port-to-role mappings.

In addition to the *architectural view,* organized by architecture, ArchE also supports an alternative *domain view,* structured by a set of domains. Domains are containers of components and contracts with specific functionality, for example, user interface domain and middleware services domain, see Fig. C.10.

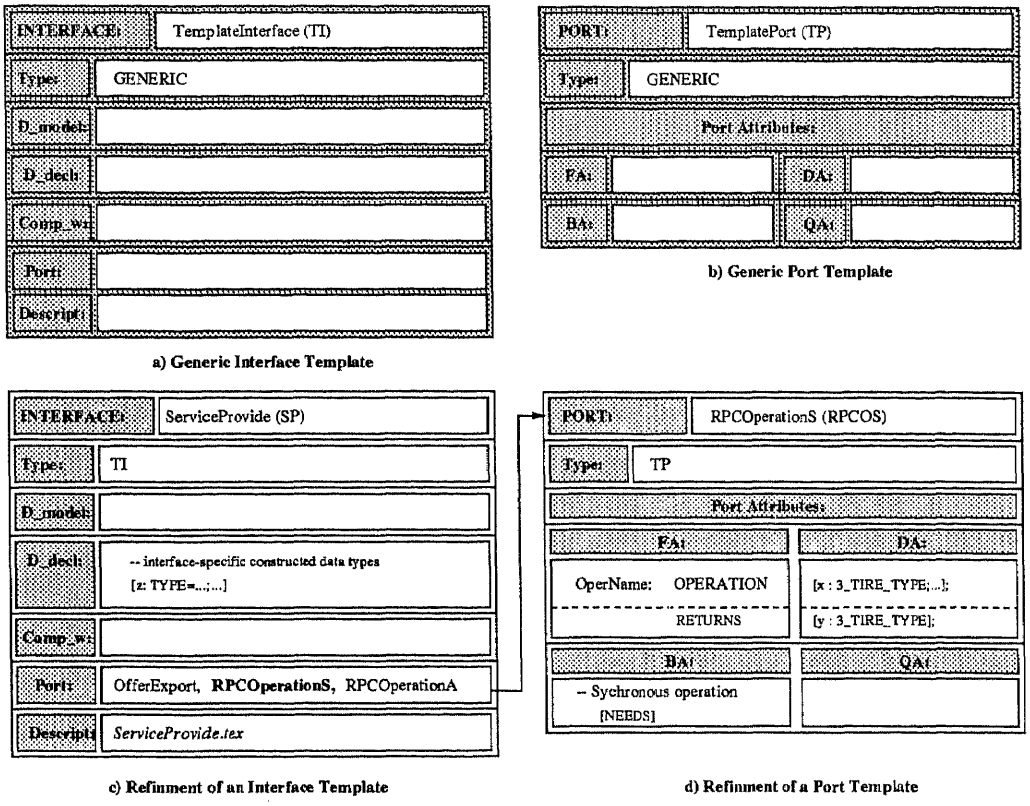


Figure C.4 *d*-ASPECT (V2) GUI: Interfaces & Ports

For visualization of dynamic scenarios, ArchE is integrated with a Message Sequence Charts tool set, uBET, based on work published in [65](see Fig. C.11).

A "right balance" between formal and informal elements of an architectural specification was a guiding principle in designing ArchE. ArchE serves as a core tool for storing the structural view of architectural specifications. It addition, it supports "document descriptors" for relating and linking architectural elements to informal or structured documents extending the meaning of structural specifications. The "document descriptors" establish links between ArchE and a Document Management System (DMS), see Fig. C.12. The ArchE and DMS pair can serve as a basis for creating an unified repository of software assets, Fig. C.13.

The next steps in our work on the prototypes discussed in this chapter include integration with external tools for automating rule specification, consistency checking

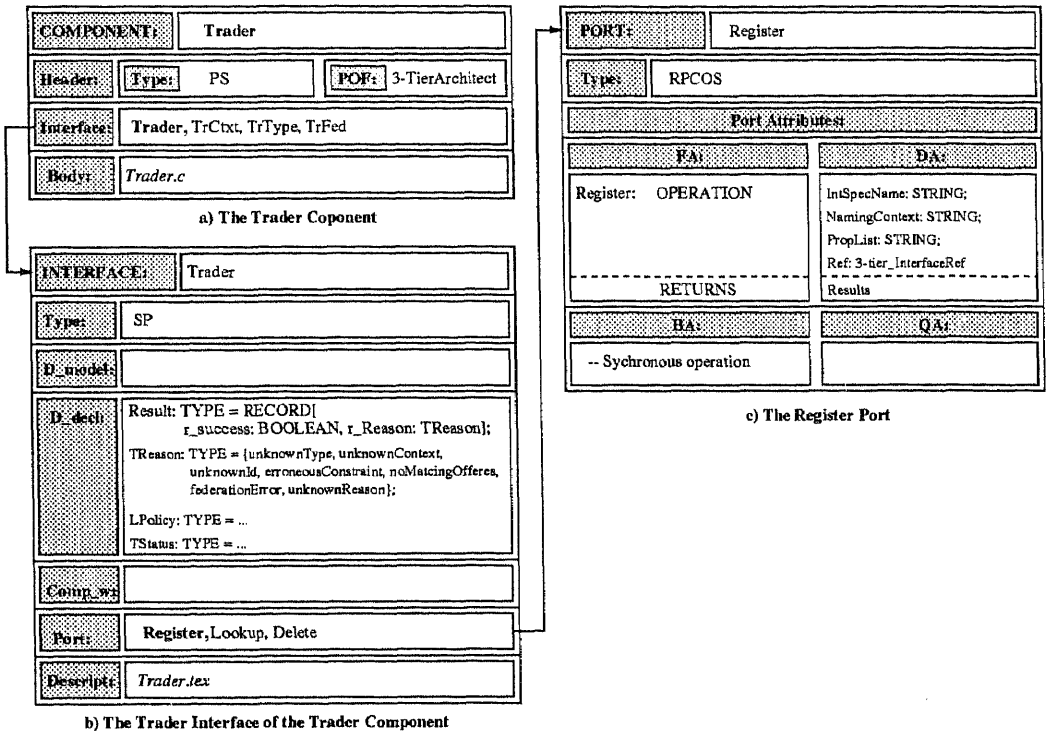


Figure C.5 *d*-ASPECT (V2) GUI: Example Component – Conceptual Architecture

and application; building library of reusable components, patterns and guidelines; and last but not least collecting feedback about the underling model and the notation.

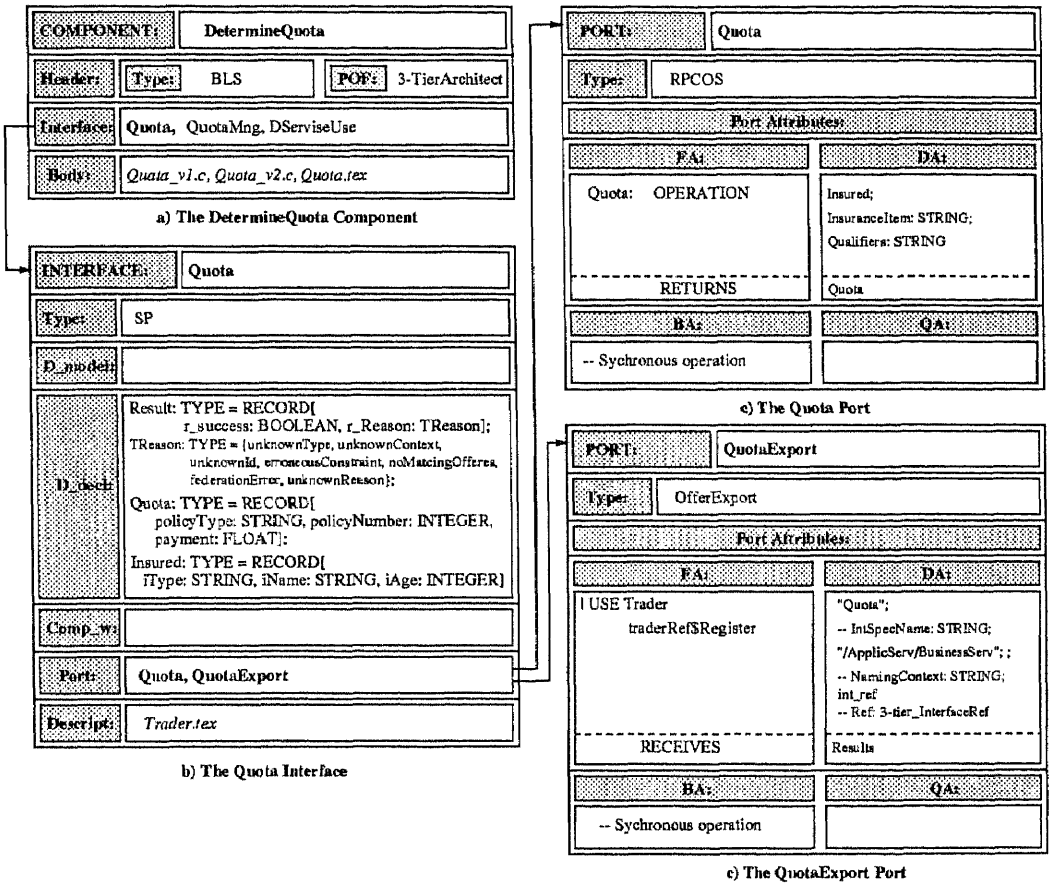


Figure C.6 *d*-ASPECT (V2) GUI: Example Component – Application Architecture

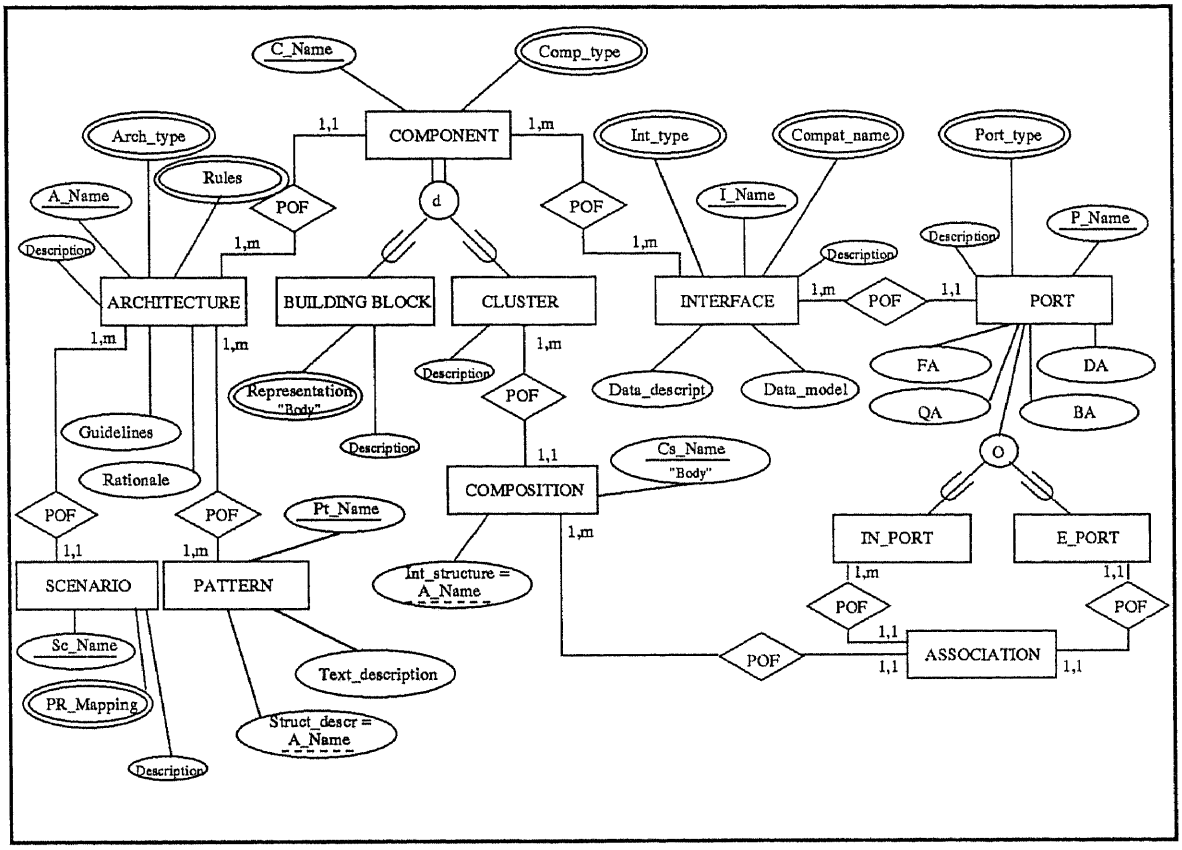


Figure C.7 *d*-ASPECT (V2) Back-end: ER model – Architecture and Component Part

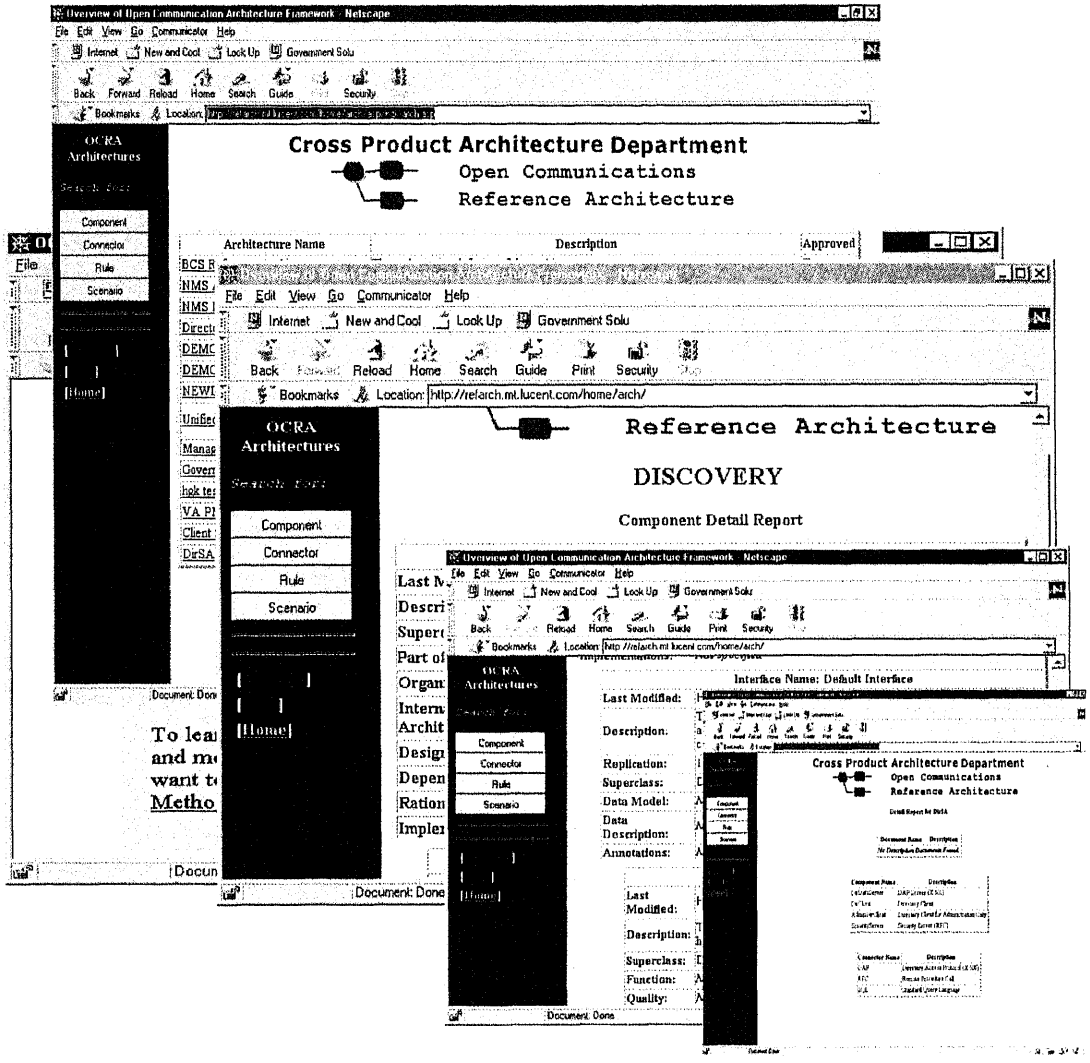


Figure C.8 ArchE: Architectural View

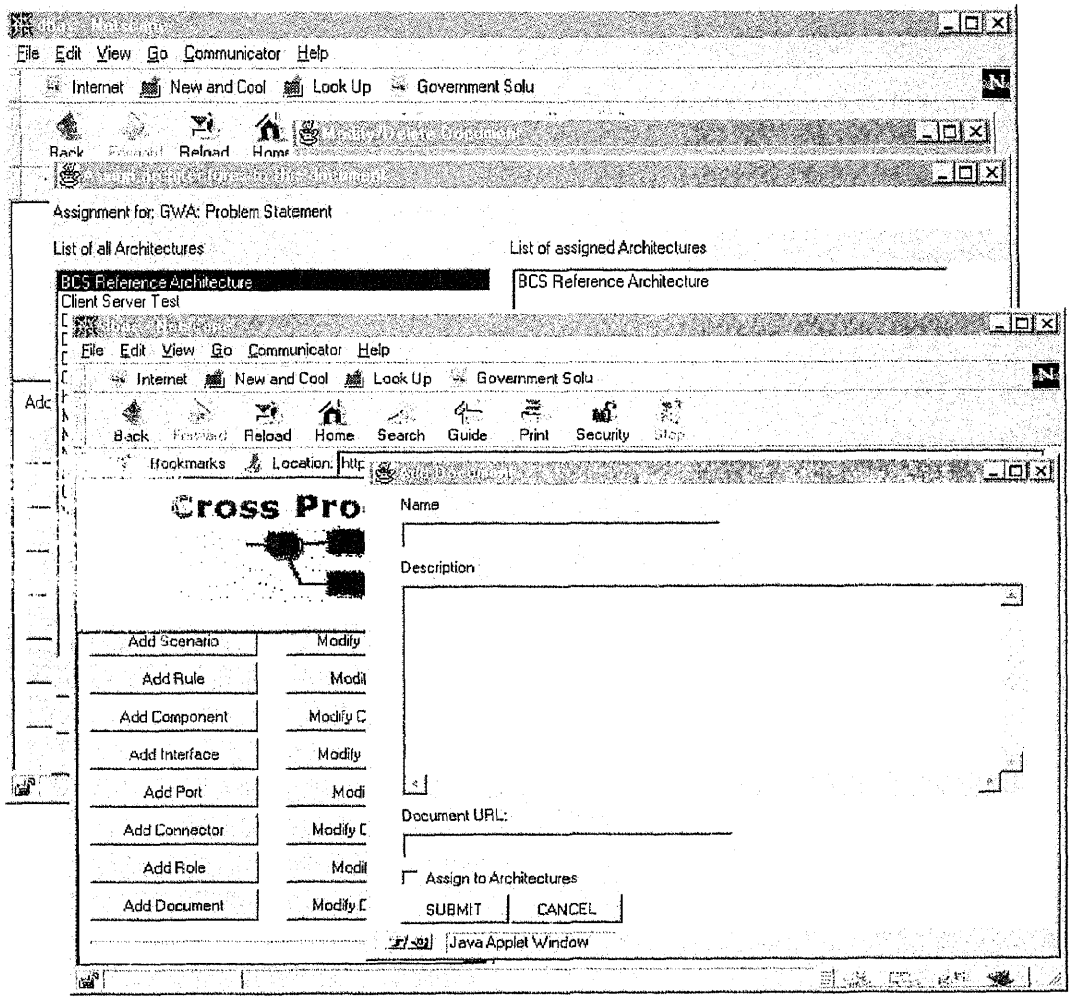


Figure C.9 ArchE: Adding Architectural Elements & Document Descriptors

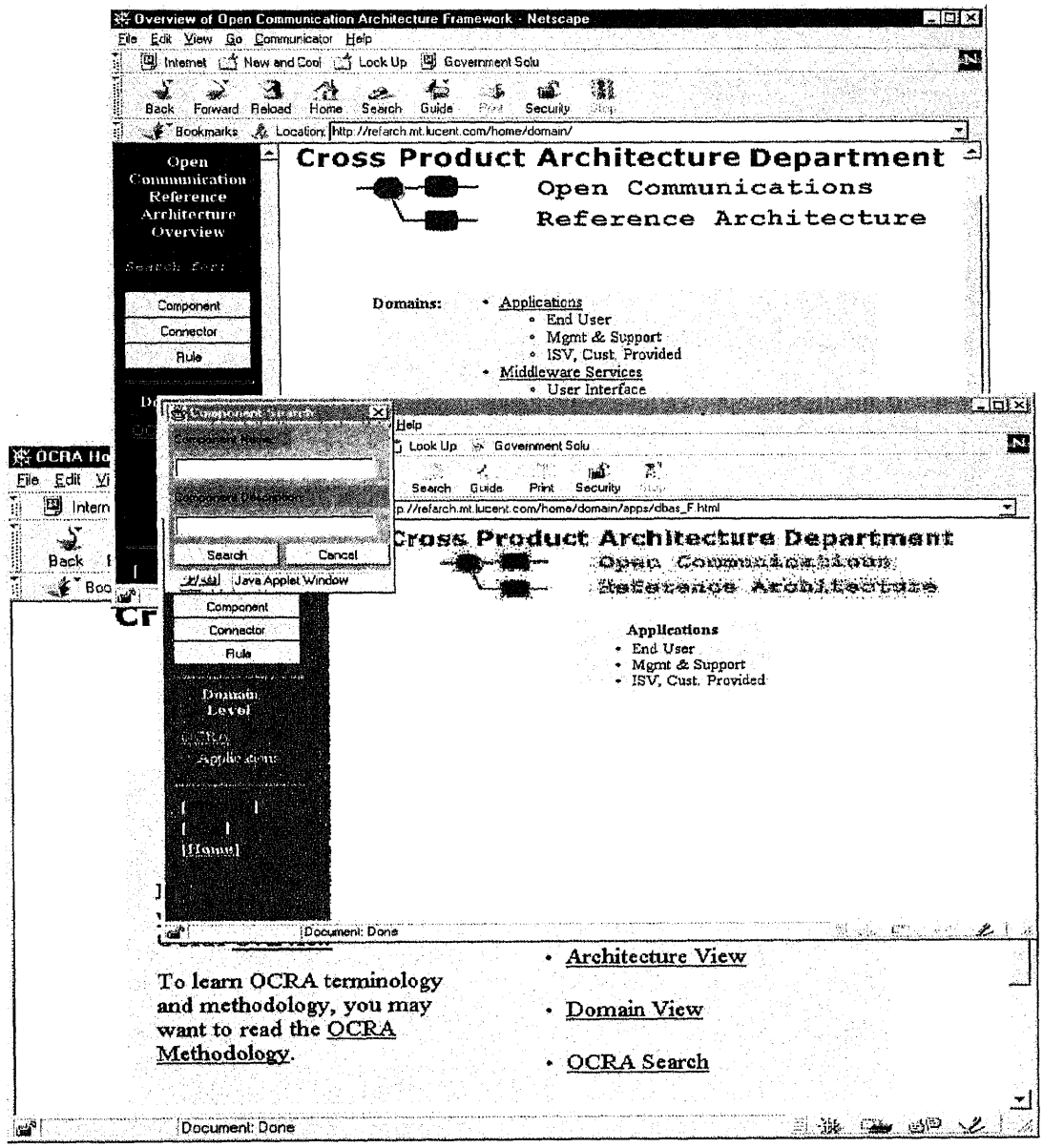


Figure C.10 ArchE: Domain View

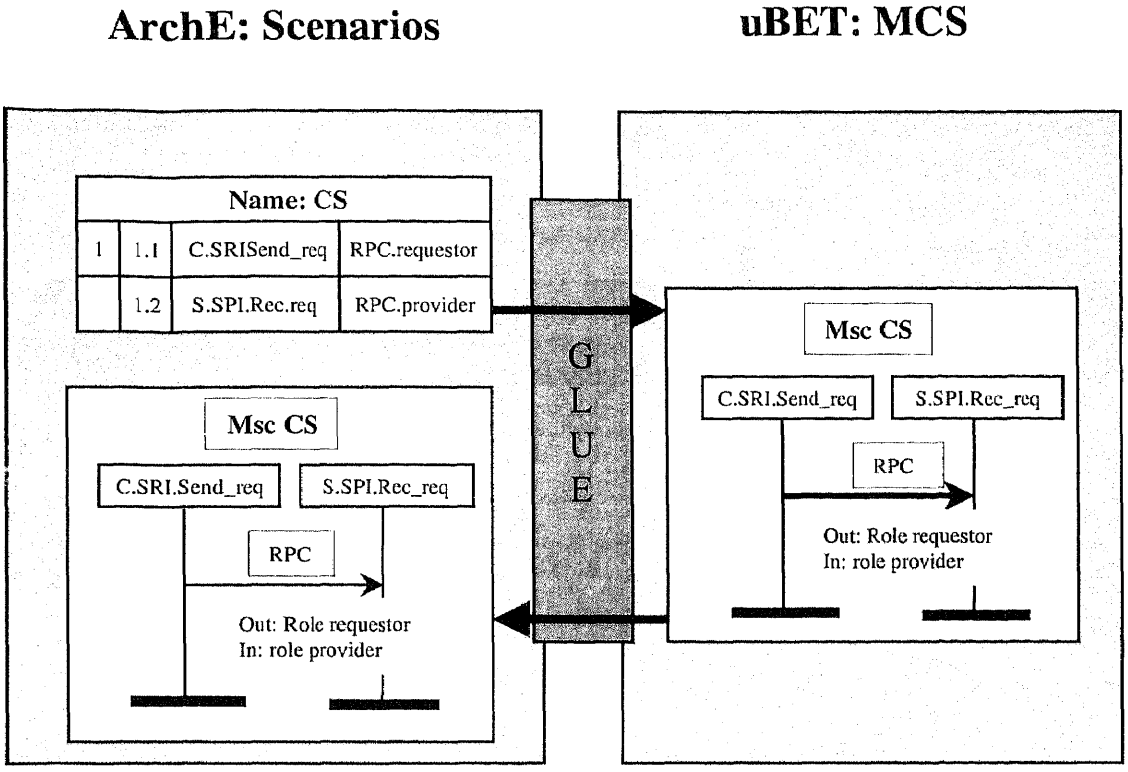


Figure C.11 ArchE & uBET

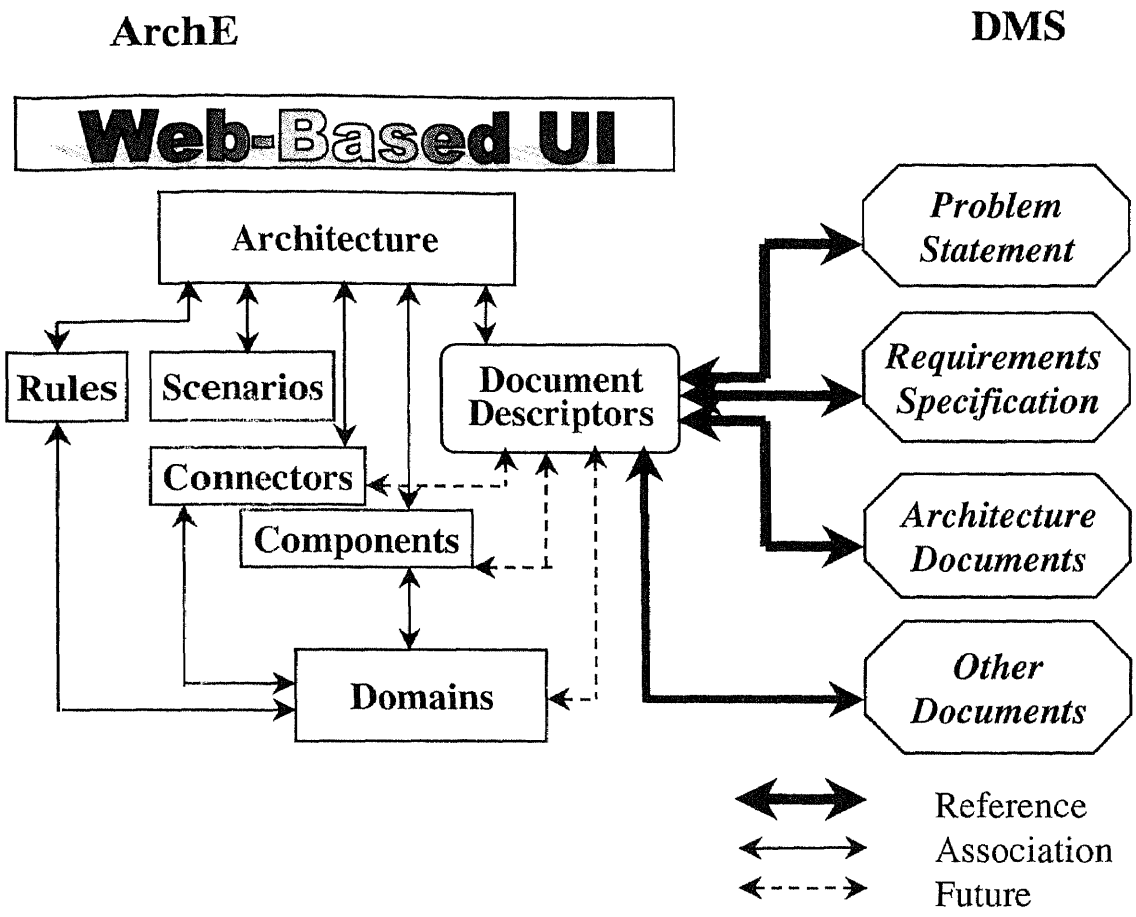


Figure C.12 ArchE & Architectural Documentation

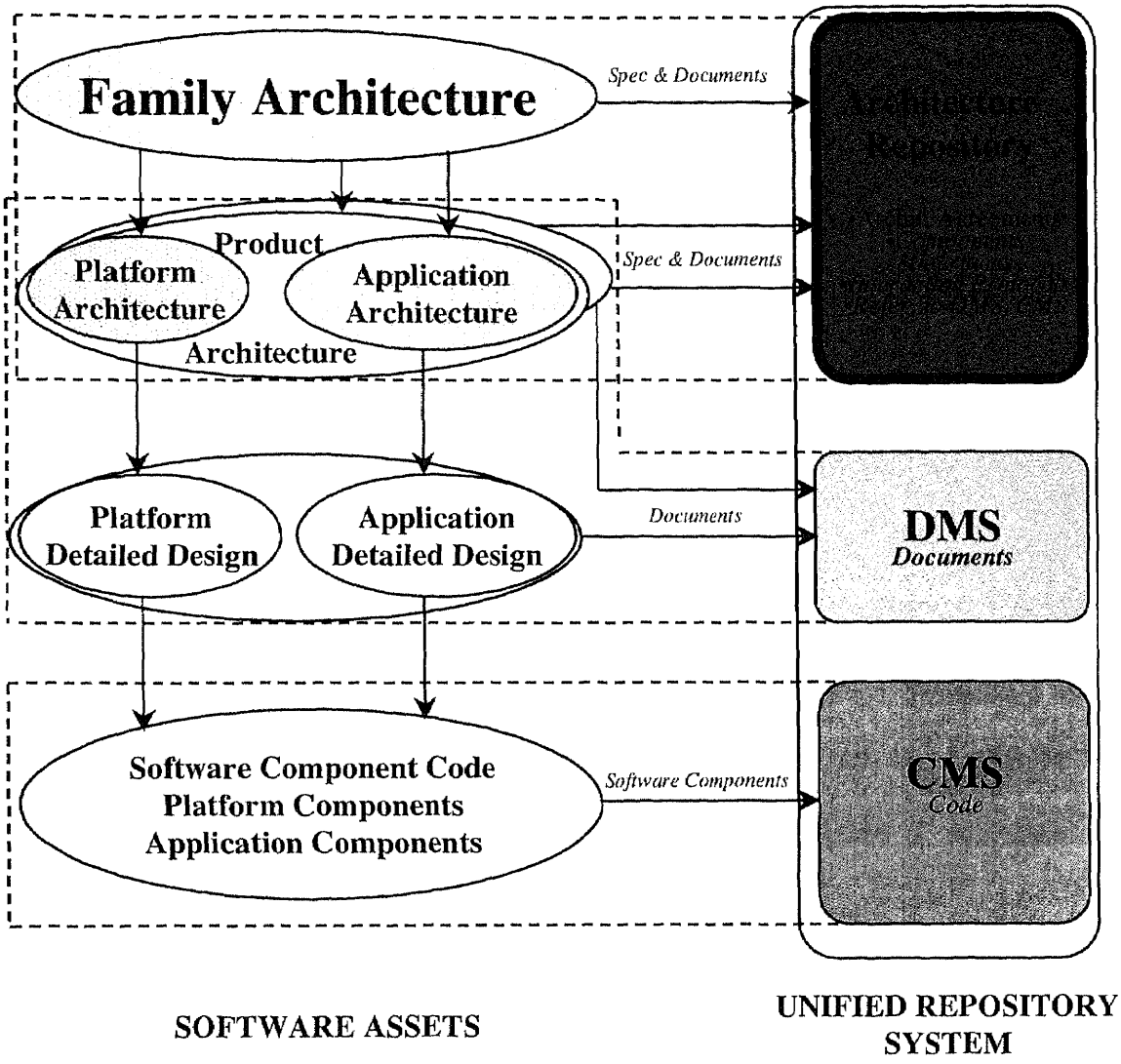


Figure C.13 A Repository for Software Assets

APPENDIX D
GLOSSARY

- **application architecture** – a domain-specific architecture in which problem-specific functions and data are explicitly defined and allocated to the design elements of the underlying conceptual architecture

- **application domain** – a denotation of the overall functionality provided by a class or family of systems as well as the related engineering principles. Examples include the domain of command and control systems and the domain of messaging applications.

- **architecture** – in IEEE Std. 610.12 (draft) the architecture concept is defined as follows: (1) a structure of components, their interrelationships, and the principles and guidelines governing their development, (operation,) and evolution support; (2) an organizational structure of a system or a component.

- **architectural framework** – a consistent, evolving collection of architectural standards, specifications, rules and agreements under the guidance and the terms of which a set of systems and components are developed, acquired, integrated, maintained and evolving

- **architectural rules** – basic principles prescribing/describing invariant system properties reflected in an architecture. They constrain architectural elements and their compositions.

- **architectural component** – a denotation that represents the primary computational entities and data of a software system. Examples include clients, servers, data capsules, filters and objects.

- **architectural connector** – a representation of the interactions among components of a software system. Connectors are "contracts" denoting sets of

requirements, obligations and constraints on the behavior of two or more ports participating in an interaction. They mediate the communication and coordination activities among components. Examples include RPC interactions, pipes, SQL links, and communication protocols.

architectural style – established pattern or idiom of system organization; a special case of a reference architecture. Examples include pipes-&-filters, layered systems, and client-server organization.

contract – an ASPECT ADL construct – a template for representing types of connectors

aspect – a perspective from which a software system properties are examined, for example, structure, behavior and security

business domain – an internally coherent, relatively self-contained space of human knowledge and action with well-defined functions, operations, services, and software engineering traditions; supported by families of domain-specific software systems. Examples include Banking, Avionics, University, Insurance, and Telecommunications.

component body – a representation of the "implementation properties" of a component required in order to support the functionality and behavior observed at the component's interfaces

connector body – a representation of the internal structure of a connector. (It reflects the structural view of a liaison/connector.)

connector protocol – a representation of the connector's interaction model. (It reflects the behavioral view of a liaison/connector.)

- **conceptual architecture** – a high-level design model of a class of software systems with characterizing specific organization and behavior, e.g., distributed systems or object-oriented systems. A conceptual architecture defines a collection of element types and their allowed configurations; typically, it prescribes a computational model, interaction model, type model and architectural solutions to properties like location transparency, fault-tolerance, etc. (The problem-specific functions are suppressed.)

- **configuration** – interconnection of components and connectors

- **domain** – a well-defined, relatively self-contained area of knowledge and activity. Typically, a domain is defined by consensus and its essence is the shared understanding of some community. Among others, domains can be characterized, by a number of (related) families of computer-based systems. From a software engineering perspective a domain is seen as an overall problem space to which computer-based systems, developed and maintained in the domain, provide solutions.

- **domain architecture** – a reference architecture constrained by the specifics of a business domain. A domain architecture constitutes a reference framework under the terms of which the domain-specific collection of software systems is developed, acquired, integrated, maintained and evolves.

- **domain model** – a description of the domain which reflects the understanding of the domain experts and the needs of system developers

- **domain infrastructure** – a unified set of domain-specific environments, common services and tools

- **family architecture** – an architecture of a class of software systems with common organization and functionality

- **guidelines** – design rationale and experiences documented as a "log" of an architecture engineering, application and evolution processes

- **interface** – an abstract external image of a component which represents its functional and data properties, observable behavior and expected qualities

- **liaison** – a representation of the interaction channel established between the roles of a contract. It has two views, structural – the contract's body, and behavioral – the contract's protocol

- **patterns** – established structural and behavioral fragments providing verified solutions to specific problems identified in the scope of a reference architecture.

- **port** – a component access point at which interactions between a component and its environment occur and component properties are being observed

- **product line architecture** – a parameterized software architecture. It defines the functional components, their interconnections and properties as well as the patterns of system organization that apply to any specific product being a member of the product line.

- **reference architecture** – an architecture of a class of software systems which serves as a prescription from which other architectures are derived

- **role** – an image of a participant in an interaction (modeled by a connector of which the role represent a partial external view). A role is a place-holder that will be substituted by a matching port when a system is constructed.

- **scenario** – a representation of a static or dynamic configuration of interconnected components and connectors where components' ports are mapped into the connectors' roles

- **service** – a denotation of functions, data and behavior provided by a component. It represents "results" being delivered to the end users or shared between the constituents of a computer-based system.

- **specific (particular) system architecture** – the architectural design of an individual software system providing a solution to a specific set of functional and extra-functional requirements

- **view** – a description emphasizing specific system properties relevant to a single well-defined perspective or aspect. Examples include structural view, data view, and control view.

REFERENCES

1. ISO/IEC JTC 1/SC 21. *Basic Reference Model of ODP, Interim revised CD text.* ISO, 1992.

2. NIST Special Publication 500-211. Reference model for frameworks of software engineering environments. Technical Report ECMA TR/55, 3rd ed., National Institute of Standards and Technology, Washington, DC 20402, August 1993.

3. G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *Software Engineering Notes*, 18, 5:9–20, December 1993.

4. Bellcore Technical Advisory. The Bellcore OSCA architecture. Technical Report TA-STS-000915, Bellcore - Bell Communications Research, March 1992.

5. A. Agrawala, M. Jackson, and S. Vestal. Domain specific software architectures for intelligent guidance, navigation & control. In *Proceedings of DARPA Software Technology Conference*, 1992.

6. R. Allen and D. Garlan. A formal approach to software architectures. In *Proceedings of IFIP'92*, 1992.

7. R. Allen and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, March 1994.

8. R. Allen and D. Garlan. A formal basis for architectural connections. Technical Report CMU-CS-97-117, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, January 1997.

9. APM. *ANSA Reference Manual, Vol.A.* Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, UK, 1989.

10. APM. ANSA Architecture Report - The ANSA Computational Model. Technical report, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, UK, 1991.

11. APM. *ANSAware4.0.* Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, UK, 1992.

12. X.400 API Association and X/Open Company Ltd. *Directory Service API Specification (XDS).* ISO, X/Open Company Ltd., 1992.

13. Anderson B. Building organizational competence in software architecture. *ACM SIGSOFT, Software Engineering Notes*, 20, 2:25–33, April 1995.

14. Bellcore. INA cycle 1 contract specification. Technical Report Issue 1, Bell Communications Research, June 1992.

15. Ph. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 32,2:86–98, January 1996.

16. L. Bertalanffy. *General System Theory.* George Braziller, Inc., New York, NY, 1993.

17. P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-specific software architectures for guidance, navigation and control. Technical report, Honeywel Technology Center, Minneapolis, MN, January 1994.

18. B Boar. *Implementing Client/Server Computing: a strategic perspective.* McGraw-Hill, New York, NY, 1993.

19. B. Boehm. Software process architectures. In *Proceedings of the First International Workshop on Architectures for Software Systems, in cooperation with ICSE-17*, 1995.

20. B. Boehm and W. Scherlis. Megaprogramming. In *Proceedings of DARPA Software Technology Conference*, 1992.

21. G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, 1993.

22. Ch. Braun and B. Balzer. Domain specific software architectures – command and control. In *Proceedings of DARPA Software Technology Conference*, 1992.

23. F. Bronsard, D. Bryan, W. Kozaczynski, E. Liongnsari, J. Ning, A. Olafsson, and J. Wetterstrand. Toward software plug and play. *ACM Software Engineering Notes*, 22:19–29, May 1997.

24. L. Callahan and J. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, 17, 6:626–636, June 1991.

25. CCITT. *Recommendation Z.120: Message Sequence Chart (MSC).* CCITT, March, 1993.

26. D. Chadwick. *Understanding X.500 The Directory.* Chapman & Hall, London, UK, 1994.

27. D. Chappell. NT 5.0 in the enterprise. *BYTE*, 22,5:66–72, May 1997.

28. P. Clements. Understanding architectural influences and decisions in large system projects. In *Proceedings of the First International Workshop on Architectures for Software Systems in Cooperation with ICSE-17*, 1995.

29. P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.

30. P. Coad and E Yourdon. *Object-Oriented Analysis.* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1991.

31. P. Coad and E Yourdon. *Object-Oriented Design.* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1991.

32. E. Cobb. Objects and transactions: together at last. *Object Magazine*, 4, 8:58–64, January 1995.

33. R. Crocker and J. Engelsma. Continuing investigation into an organizational-wide software architecture. In *Proceedings of the First International Workshop on Architectures for Software Systems, in Cooperation with ICSE-17*, 1995.

34. A. Davis. *Software Requirements: Objects, Functions and States.* PRT Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

35. F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2,2:80–86, June 1976.

36. S. Desai, D. Follett, H. Rao, G. Sinha, C. Sundaramurthy, and P. Yerneni. BaseWorkX platform for building object-oriented management applications. In *Proceedings of IEEE Telecommunications Conference*, 1993.

37. W. Deubler and M. Koestler. Introducing object orientation into large and complex systems. *IEEE Transactions on Software Engineering*, 20, 11:840–848, November 1994.

38. M. Diaz, J. Ansart, J. Courtiat, P. Azema, and V. (Editors) Chari. *The Formal Description Technique ESTELLE: results of the ESPRIT/SEDOS Project.* Elsevier Science Publishers Co., Amsterdam; New York: North-Holland; New York, NY, USA, 1989.

39. L. Druffel, N. Loy, R. Rosenberg, R. Sylvester, and R. Volz. Information architectures that enhance operational capability in peace time and war time. Technical report, US Air Force Scientific Advisory Board, Washington DC, February 1994.

40. B. Dudney. Case Study – Frameworks: They Work (Pan Canadian Petroleum uses frameworks to expedite delivery of applications). *Open Step Solutions - SIGS Publications*, pages 8–19, January 1995.

41. H. Eisner, J. Marciniak, and R. McMillan. Computer aided system of systems (S2) engineering. In *Proceedings of IEEE/SMC International Conference on Systems, Man, Cybernetics*, 1991.

42. H. Eriksson and M. Penker. *UML Toolkit.* John Wiley & Sons, Inc., New York, NY, 1998.

43. G. Fischer. Domain-oriented design environments. *Automated Software Engineering*, 1:69–81, No. 2 1994.

44. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design.* Addison-Wesley, Reading, MA, 1994.

45. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

46. D. Garlan. ICSE-17 software architecture workshop summary. *Software Engineering Notes*, 20, 3:84–89, July 1995.

47. D. Garlan. What is a style? In *Proceeding of the First International Workshop on Architectures for Software Systems, in cooperation with ICSE-17*, 1995.

48. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. *Software Engineering Notes*, 19, 5:175–187, December 1994.

49. D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, 1997.

50. .D. Garlan and D. Notkin. *VDM'91: Formal Software Development Methods. Formalizing Design Spaces: Implicit Invocation Mechanisms, pages 31-44.* Springer-Verlag, LNCS 551, Noordwijkerhout, The Netherlands, October, 1991.

51. D. Garlan and F. Paulisch. Summary of the Dagstuhl workshop on software architecture. *ACM Software Engineering Notes*, July 1995.

52. D. Garlan and D. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21, 4:269–274, April 1995.

53. D. Garlan and M. Shaw. An introduction to software architectures. In *Proceedings of IEEE, ICSE-15, Tutorial Notes: Architectures for Software Systems*, 1993.

Garlan, W. Tichy, and F. Paulisch. Dagstuhl software architecture workshop summary. *Software Engineering Notes*, 20, 3:63–83, July 1995.

Genesereth. Software agents. In *Proceedings of DARPA Software Technology Conference*, 1992.

Genesereth and S. Ketchpel. An agent-based framework for software interoperability. *Communications of the ACM*, 37, 7:48–147, July 1994.

Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communications. *ACM Transactions on Computer Systems*, 6, 3:258–283, August 1988.

Gotzhein. On conformance in the context of open systems. In *Proceedings of IEEE 12th International Conference on DCS*, 1992.

Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, 12, 6:648–692, Junet 1986.

Guillermo. Domain analysis – from art form to engineering discipline. In *Proceedings of IEEE 5th International Worksop on Software Specification and Design*, 1989.

Hammer. The development of large and complex software systems: A purely technical issue. In *Proceedings of Computers in Engineering Symposium ETCE'94*, 1994.

Harel. A visual formalism for complex systems. *Science of Computer Programming*, 8, 3:231–274, June 1987.

Herbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Management Systems*, 3, 3:253–278, 1985.

Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

Holzmann, D. Peled, and M. Redberg. Design tools for requirements engineering. *Bell Labs Technical Journal*, pages 86–95, Winter 1997.

Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* New York: ACM Press; Wokingham, England; Reading, MA: Addison-Wesley Pub., 1993.

Jeffay. Real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of ACM/SIGAPP Symposium on Applied Computing*, 1993.

68. P. Johannesson. *Schema Integration, Schema Translation and Interoperability in Federated Information Systems.* PhD thesis, Stockholm University, Stockholm, Sweden, 1993.

69. J. John, V. Kirova, and W. Rossak. Case Study: Insurance Domain Model into Domain Architecture Mapping. Technical report, Department of Computer and Information Science, NJIT, Newark, May 1994.

70. V. Kirova, H. Kradjel, W. Rossak, and T Marlowe. Engineering and Representation of Software Architectures, DirSA: Case Study. In *Proceedings of IEEE Conference on Software Process & Design Methodologies*, 1998.

71. V. Kirova and W. Rossak. Representing architectural designs: A central issue in the development of complex systems. In *Proceedings of First IEEE Conference on Engineering of Complex Computer Systems*, 1995.

72. V. Kirova and W. Rossak. ASPECT – An Architecture SPECification Technique: A Report on Work in Progress. In *Proceedings of IEEE Symposium and Workshop on Engineering of Computer-Based Systems*, 1996.

73. V. Kirova and W. Rossak. Elements of software architectures - the GenSIF/GARM model. *Informatica, The International Journal of Computing and Informatics, SSI*, 20:159–172, April 1996.

74. V. Kirova and W. Rossak. ASPECT: The Generic Architecture Description Language and its Customization Facilities. In *Proceedings of IEEE Conference on Engineering of Computer-Based Systems*, 1998.

75. V. Kirova, W. Rossak, and L. Jololian. Software architectures for mega-system development – basic concepts and possible specification. In *Proceedings of IEEE Third International Conference on Systems Integration*, 1994.

76. V. Kirova, W. Rossak, and H. Lawson. Software architecture: An analysis of characteristics, structure, and application. In *Proceedings of First International Workshop on Architectures for Software Systems, in cooperation with ICSE-17*, 1995.

77. V. Kirova and Rossak W. Some thoughts on architecture engineering. In *Proceedings of Computers in Engineering Symposium, ETCE'94*, 1994.

78. G. Koch and K. Loney. *Oracle: the Complete Reference.* Osborne McGraw-Hill, Berkeley, CA, 1997.

79. H. Kradjel, V. Kirova, and R. Choudhary. Leveraging Architecture and Process to Achieve Software Asset Reuse. In *Proceedings of IEEE Conference on Engineering of Computer-Based Systems*, 1998.

80. D. Lamb. IDL: sharing intermediate representations. *ACM Transactions on Programming Languages and Systems*, 9, 3:297–318, April 1987.

81. H. Lawson, V. Kirova, and W. Rossak. A refinement of the ECBS architecture constituent. In *Proceedings of Tutorial and Workshop on Systems Engineering of Computer-based Systems*, 1994.

82. B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31, 3:300–312, March 1988.

83. W. Litwin. Interoperability of multiple autonomous databases. *ACM Computing Serveys*, 22:267–293, September 1990.

84. H. Lockhart. *OSF DCE Guide to Developing Distributed Applications.* McGraw-Hill, New York, USA, 1994.

85. IONA Technologies Ltd. *ORBIX 2, Reference Guide.* IONA Technologies Ltd., IONA Technologies Inc., 201 Broadway, 3rd Floor, Cambridge, MA 02139, 1995.

86. D. Luckham, L. Augustin, J. Kenney, J. Vera, D. Bryan, and Maan W. Specification and analysis of system architecture using Rapid. *IEEE Transactions on Software Engineering*, 21, 4:336–355, April 1995.

87. D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21, 9:717–734, September 1995.

88. N. Lynch and M. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT, MIT Laboratory of Computer Science, 1988.

89. J. Magee, N. Dulay, S. Eisenbach, and Kramer J. Specifying distributed software architecture. In *Proceedings of the Fifth Europiean Software Engineering Conference, ESEC'95*, 1995.

90. J. Maranzano. System architecture validation: Review findings. In *Proceedings of the First International Workshop on Architectures for Software Systems, in cooperation with ICSE-17*, 1995.

91. S. Mellor. Reuse through automation: Model-based development. *Object Magazine*, 5, 5:50–55, September 1995.

92. D. Menasce, H. Gomaa, and L. Kerschberg. A performance oriented design methodology for large-scale distributed data intensive information systems. In *Proceedings of the First IEEE International Conference on Engineering Complex Computer Systems*, 1995.

93. Merriam-Webster. *Webster's Third New Internationsl Dictionary.* Ecyclopedia Britanica, Chicago, IL, 1986.

94. E. Mettala and M. Graham. The domain specific software architecture program. Technical Report CMU/SEI-92-SR-9, SEI, Carnegie Mellon University, June 1992.

95. E. Mettala and M. Graham. The domain specific software architecture program. In *Proceedings of DARPA Software Technology Conference*, 1992.

96. B. Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall, New York, USA, 1991.

97. N. Minsky and D. Rozenshtein. A software development environment for law-governed systems. *SIGPLAN NOTICES*, 24, 2:65–75, November 1988.

98. R. Mittermeir. Powder - a recursive methodology for prototyping of wicked development efforts with reuse. Technical report, Institute f. Informatik Universitaet Klagenfurt, Austria, Report for International Software Systems Inc., Austin TX, 1991.

99. R. Mittermeir. Conceptual modelling by layered specifications and intra-object schemas to gently refurbish very large systems. In *Proceedings of Computers in Engineering Symposium, ETCE'94*, 1994.

100. M. Moriconi and X. Qian. Correctness and composition of software architectures. *Software Engineering Notes*, 19, 5:164–174, December 1994.

101. M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21, 4:356–372, April 1995.

102. J. Nestor, J. Newcomer, P. Giannini, and D. Stone. *IDL: The Language and Its Implementation.* Prentice-Hall, Englewoods Cliffs, NJ, 1990.

103. D. Newman. Strategic planning for distributed object management. *Object Magazine*, pages 74–77, June 1994.

104. OMG. *The Common Object Request Broker: Architecture and Specification, revision 2.0.* Object Management Group, Inc., 1995.

105. D. Parnas and P. Clements. A modular structure of complex systems. In *Proceedings of IEEE 7th International Conference on Software Engineering*, 1984.

106. D. Perry. The inscape environment. In *Proceedings of IEEE 11 International Conference on Software Engineering*, 1989.

107. D. Perry. State of the art: Software engineering. In *Proceedings of IEEE International Conference on Software Engineering, ICSE'97*, 1997.

108. D. Perry and A. Wolf. Foundations for study of software architecture. *Software Engineering Notes*, 17, 4:40–52, October 1992.

109. J. Petterson. Petri nets. *ACM Computing Serveys*, 9, 3:223–252, September 1977.

110. R. Power. Post-facto integration technology: New discipline for an old practice. In *First International Conference on Systems Integration*, 1990.

111. R. Prieto-Diaz. Domain analysis: An introduction. *Software Engineering Notes*, 15, 2:47–54, April 1990.

112. R. Prieto-Diaz and G. Arango (eds.). *Domain Analysis and Software Systems Modeling.* IEEE Computer Society Press, New York, NY, 1991.

113. R. Prieto-Diaz and J. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6:307–334, November 1986.

114. Software Process Program. The capability maturity model for software. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA 15213-3890, 1990.

115. J. Purtilo. Polylith system. *IEEE Transactions on Programming Languages*, 17, 6:626–636, June 1993.

116. Rational. *Object Constraint Language (OCL).* Rational Software Corporation, http://www.rationale.com/uml/html/ocl, 1997.

117. Rational. *Unified Modeling Language (UML).* Rational Software Corporation, http://www.rationale.com/uml, 1997.

118. D. Riecken. M: An architecture of integrated agents. *Communications of the ACM*, 37, 7:107--116/146, July 1994.

119. W. Rossak and V. Kirova. A generic model for use of software architectures. Technical Report 119, Department of Computer and Information Science, New Jersey Institute of Technology, Newark , NJ 70102, May 1995.

120. W. Rossak, V. Kirova, L. Jololian, H. Lawson, and T. Zemel. A generic model for software architectures. *IEEE Software*, 14, 4:84–92, July/August 1997.

121. W. Rossak and P. Ng. Some thoughts on systems integration - a conceptual framework. *International Journal of Systems Integration*, 1, 1:97–114, 1991.

122. W. Rossak and T. Zemel. Integrative domain analysis via multiple perceptions. *Informatica,The International Journal of Computing and Informatics, SSI*, 17, 2:117–136, February 1993.

123. W. Rossak, T. Zemel, V. Kirova, and L. Jololian. A two-level process model for integrated system development. In *Proceedings of Symposium and Workshop on Systems Engineering of Computer Based Systems*, 1995.

124. W. Rossak, T. Zemel, and H. Lawson. A meta-process model for the planned development of integrated systems. *Journal of Systems Integration*, 1, 3:225–249, 1993.

125. M. Roy and A. Ewald. Two distributed object application models. *Object Magazine*, 1:79–81, September 1995.

126. K. Rubin and P. McClaughry. Modeling rules using object behaviour analysis and design. *Object Magazine*, 4, 3:63–67, June 1994.

127. J. Rumbaugh, M. Blaha, W. Premeriani, and F. Eddy. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

128. M. San Soucie and J. Almarode. Supporting enterprise-wide business objects. *Object Magazine*, 5, 5:44–59, September 1995.

129. M. Sebern. *Buiding OSF/Motif Applications.* Prentice Hall, Englewoods Cliffs, NJ, 1994.

130. SEI. *Proceedings of the Workshop on Domain-Specific Software Architectures.* Software Engineering Institute, Hidden Valley, PA, July 1990.

131. M. Shaw. Large scale systems require higher-level abstractions. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, 1989.

132. M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical Report CMU-CS-94-107, School of Computer Science, Cornegie Mellon University, Pittsburgh, PA 15213-3890, January 1994.

133. M. Shaw. Comparing architectural design styles. *IEEE Software*, 12, 6:27–41, November 1995.

134. M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21, 4:314–335, April 1995.

135. M. Shaw and D. Garlan. Characteristics of higher-level languages for software architecture. Technical Report CMU/SEI-94-TR-23, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, December 1994.

136. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, Upper Saddle River, NJ, 1996.

137. M. Shaw, D. Garlan, R. Allen, D. Klein, J. Ockerbloom, and C. Scott. Candidate model problems in software architecture. Technical Report V1.0, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, December 1993.

138. A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22:183–236, December 1990.

139. S. Shlaer and S. Mellor. *Object Lifecycles Modeling the World in States.* YOURDON PRESS, 1992.

140. K. Shumate and M. Keller. *Software Specification and Design: a disciplined approach for real-time systems.* John Wiley and Sons, New York, NY, 1992.

141. H. Simpson. Correctness analysis for a class of asynchronous communication mechanisms. *IEE Proceedings – E*, 139-1:35–49, January 1992.

142. H. Simpson. Architecture for computer based systems. In *Proceedings of Tutorial and Workshop on Systems Engineering of Computer-Based Systems*, 1994.

143. H. Simpson. Architectural issues embodied in the DORIS notations and concepts. Technical Report Draft, British Aerospace (Dynamics) Ltd., PO Box 19, Six Hills Way, Stevenage, Herts SG1 2DA, UK, June 1995.

144. J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, 1992.

145. Formal Systems. *Failures Divergence Refinement: User Manual and Tutorial.* Formal Systems (Europe) Ltd., Oxford, England, August, 1993.

146. A. Tang and S. Scoggins. Open networking with OSI. Prentice Hall, Englewood Cliffs, NJ, 1992.

147. D. Taylor. Objects for the enterprise. *Object Magazine*, 4, 8:65–76, Jenuary 1995.

148. W. Tracz. A conceptual model for megaprogramming. *ACM SIGSOFT Software Engineering Notes*, 16, 3:36–45, July 1991.

149. W. Tracz. Megaprogramming and domain engineering. In *Proceedings of IEEE 15th ICSE, Tutorial Notes*, 1993.

150. W. Tracz. DSSA frequently asked questions. *Software Engineering Notes*, 19, 2:52–57, April 1994.

151. W. Tracz and L. Coglianese. A case for domain-specific software architectures. Technical Report DRAFT, IBM Corporation, Federal Systems Company, Owego, NY 13827-1298, - 1993.

152. S. Vinoski. Distributed object computing with CORBA. *C++ Report Magazine*, pages 74–89, July-August 1993.

153. M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (LDAP) v3., REC 2251.* IETF, http://ds.internic.net/rfc/rfc2251.txt, 1997.

154. B. Whittle. Models and languages for component description and reuse. *ACM SIGSOFT, Software Engineering Notes*, 20, 2:76–89, April 1995.

155. G. Wiederhold and M. Genesereth. The basis for mediation. In *Proceedings of COOPIS'95 Conference*, 1995.

156. G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Communications of the ACM*, 35, 11:89–99, November 1993.

157. D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19, 2:292–333, March 1997.

158. A. Zaremski and J. Wing. Specification matching of software components. Technical Report CMU-CS-95-127, School of Computer Science, Carnegie Mellon University, March 1995.

159. T. Zemel. *MegSDF - Mega-System Development Framework.* PhD thesis, Department of Computer and Information Science, NJIT, Newark NJ, USA, 1993.