VETENSKAP OCH KONST KUNGL TEKNISKA HÖGSKOLAN

ENTERPRISE SOFTWARE SYSTEM INTEGRATION
AN ARCHITECTURAL PERSPECTIVE

Pontus Johnson

April 2002

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Industrial Information and Control Systems
KTH, Royal Institute of Technology
Stockholm, SWEDEN

Ex.R. 02-03
TRITA-ICS-0202
ISSN 1104-3504
ISRN KTH/ICS/R--02/2--SE

Cover photography © 2002 Pontus Johnson

Stockholm 2002, Universitetsservice US AB

Abstract

The present thesis is concerned with enterprise software systems of companies in the Swedish electricity industry, an industry that for the past few years has been exposed to a fairly tumultuous change process as a consequence of legislative reforms.

Previously, the business operations of electric utilities – as well as those of most companies in the computerized world – were supported by a number of isolated software systems performing specific tasks. In recent years, these systems have been extended, and more importantly, integrated into a company-wide system in its own right, in this thesis referred to as the enterprise software system. As enterprise software systems have evolved, so has a need for strategies, methods and techniques for their management. Enterprise software system management involves a number of concerns; of these concerns, software integration is one of the most prominent.

The discipline of software architecture is concerned with the modeling of large-scale structures of software systems. While the generated models are employed for a number of purposes, their perhaps most significant function is to serve as a base for reasoning about the represented system. Several methods for analysis of architectures have been proposed in recent years, and although software architecture analysis has displayed considerable success for a number of systems, enterprise software systems have to a large extent been ignored.

In the empirical context of the Scandinavian electricity industry, this thesis explores the applicability of software architecture analysis to enterprise software system integration. Two conceptually different architectural analysis methods – deduction- and induction-based approaches – are considered, as well as the engineering process in which architectural analysis is performed. As a result of the investigations, the thesis proposes a modified process for architectural analysis, presents an evaluation of deduction-based analysis methods, and proposes an adaptation of induction-based analysis methods to the enterprise software system context.

Key words:

Software architecture, Software integration, Enterprise software systems, Electric utilities, Software engineering

Acknowledgements

This doctoral thesis is a result of my close to five years of Ph.D. studies at the Department of Industrial Information and Control Systems, Royal Institute of Technology in Stockholm. Although the thesis bears my name, its contents have been influenced by many.

Firstly, I would like to express my sincere gratitude to Professors Torsten Cegrell and Johan Schubert. Without Torsten and Johan, the completion of this endeavor would not have been possible. The inspiring environment at the department is to a large extent the result of Torsten's enthusiastic and entrepreneurial commitment. Johan has provided me with invaluable support both practically and philosophically. Our discussions as well as Johan's insightful (and surprisingly rapid) comments on the considerable masses of text I have sent his way have helped me immensely.

I am indebted to the people at the department for creating an atmosphere that has made these years both rewarding and fun. In particular, I would like to thank Jonas Andersson for stimulating research cooperation, including many coffee-drenched late nights and early mornings of debating, reading, and writing. I have much enjoyed the cooperation and discussions with Mathias Ekstedt, ranging from music and photography to concepts and relations slightly bigger than our heads. I would furthermore like to express my appreciation to Magnus Haglind, who has been an important speaking partner during these years, as well as Judith Westerlund, without whom the department would surely come to a grinding halt.

Finally, I would like to thank my family and my friends for their encouragement and for all things that have not been work during this period.

Stockholm, April 2002

Pontus Johnson

Table of Contents

Chapter 1
Introduction

This thesis project is spawned from the general research conducted at the Department of Industrial Information and Control Systems at KTH. Originating in control systems procurement and development, the department is concerned with information system management practices in an industrial setting. A central stance of the department is the focus on the software system *user* organization as an active part in the engineering process, as opposed to the software-vending organization. Traditionally, the concerns of user organizations have been the domain of disciplines such as information systems research, with a focus on organizational issues rather than technological. In absence of an established engineering discipline of software management for user organizations, the department has linked itself to several related fields, including systems engineering [Ste98], information systems research [Avg00], requirements engineering [Dav93], software engineering [Pre00], and project management [Dun96]. In line with an increasing awareness at the department that software architectural methods and approaches [Sha96a] may be applicable also in the context of user organizations, a research project was conceived for exploring these issues. The present thesis is thus part of this first attempt at outlining the potential benefits of software architecture in the context of the department.

The empirical base of this research is to be found in Swedish electric utilities. During the past decade, the Scandinavian electricity industry has undergone radical changes due to market deregulation [Ene01]. The legislative reforms have resulted in lower margins, a wave of mergers and acquisitions, and new requirements for business operations. These consequences of the deregulation have had a significant impact on the software environment of electric utilities [Hag02] [And02]. Investments in new systems have resulted from attempts to streamline the operations, from initiatives to offer new services, as well as from regulatory requirements. Software system integration projects have resulted from mergers and acquisitions as well as from efficiency programs in the enterprises. The electricity industry has thus constituted an excellent empirical base for the present work.

The software systems of typical mid-sized to large electric utilities in the industrialized world belong to a category here referred to as *enterprise software systems*. An enterprise software system is the interconnected set of systems that is owned and managed by organizations whose primary interest is to use rather than develop the systems. Typical *components* in enterprise software systems are thus considered as proper *systems* in most other cases. They bear names such as process control systems, billing systems, customer information systems, and geographical information systems. In the early days, these components were separated from each other physically, logically, and managerially. During the last decades, however, an ever-increasing integration frenzy has gripped the enterprises of the computerized world. Today's enterprise software system is thus multi-vendor based and enterprise-wide, characterized by heterogeneous and large-grained components. The management of these interconnected systems has become a major issue for their owners [Ola01].

Software integration is in many respects similar to solving jigsaw puzzles. If the pieces were designed to fit, there is a fair chance of accomplishing the task with a reasonable effort. If – as unfortunately oftentimes is the case – there was little or no coordination between the developers of the individual pieces, then attempting to solve the puzzle becomes a difficult assignment, where either the pieces themselves have to be re-sawed or new interconnection pieces have to be fabricated and introduced into the puzzle. And when the pieces finally fit, the picture may very well have been altered. Previously of modest importance, enterprise software integration has recently and rapidly become a major concern for the software-dependent industry. With the increasing reliance on software systems and the proliferation of computer networks, the benefits of integration of related systems have become significant. According to the ARC Advisory Group, the revenues of the market for enterprise integration reached $4.8 billion in 2001, and will continue to grow by 20% per year in the next five years, despite the recent economic downturn [Arc01]. Software integration thus constitutes a considerable business.

Software architecture is a fairly new concept. In recognition of the ever-growing complexity of software systems, software architecture has been proposed as a tool for managing this complexity by means of abstraction [Sha89]. It is the belief of the author that the main benefit of software architecture is linked to the reasoning about the system that the architectural description allows. From a map, it is possible to reason about distances, estimate time of arrival, etc. From a useful architectural description, it should be possible to make similar estimations about properties of the system, be it performance, modifiability, etc. Because of its novelty, exactly what analyses software architecture descriptions can serve as an input for is unclear. Reliability [Ves98], performance [Spi98], security [Mor97a], and modifiability [And02] [Las02] are some of the properties that have been approached by the research community so far.

The bulk of the software architecture research is concerned with the processes and artifacts of software vendors. The implicit motivation is that software vendors are software developers while software users are nothing but users. This thesis is, as mentioned, based on the view that user organizations by necessity also are software-developing organizations. Even though the components in the enterprise software system are normally acquired from external sources, the management of the resulting interconnected system becomes the responsibility of the owner. The owner responsibility includes the long-term evolution of the system, the procurement and integration of new components, the modification and retirement of legacy systems. This text is specifically concerned with the integration aspects of these enterprise software systems.

Thus, the present work is located in the intersection of the concepts of software architecture analysis, enterprise software systems and software integration. The aim is to investigate the potential for extending software architecture analysis to enterprise software system integration. Although this ambition has remained in focus during the thesis project, the precise modus operandi has changed several times as new insights have been gained. However, since few readers would benefit from an account of the intellectual evolution of the author, the thesis is presented in an as coherent manner as possible.

1.1 RESEARCH QUESTION

The purpose of the thesis is here presented in the form of a main research question, which in the next section is refined into four sub-questions. The main question posed by the thesis is:

To what extent is software architecture analysis applicable to enterprise software system integration?

The key terms of the main research question are thus *software architecture analysis*, *enterprise software systems*, and *(software) integration*. They will all be extensively elaborated on in the bulk of the thesis.

1.2 RESEARCH RATIONALE

Keeping the delimitation of enterprise software system integration in mind, the investigations of the thesis can be refined into two areas of software architecture analysis.

The architectural analysis method. The architecture description of a software system is in many ways the core of software architecture. Central concepts, such as component, connector, and view are all represented in the architectural description. A consideration of software architecture without its description is as unimaginable as geography without a map. Tightly linked to the form of the architectural descriptions are the means used for reasoning about them, the architectural analysis methods.

The architecture analysis process. The architectural description and its analysis techniques do not exist in splendid isolation. Descriptions need to be constructed, data collected, scenarios elicited, actions taken based on the analysis results, and so on. The engineering process in which architectural analysis is performed is thus a second issue of this work.

Together, these two areas encompass the basis of architectural analysis. As mentioned, these issues are herein further guided by the aim of extending the use of software architecture to enterprise software system integration. Below, a number of sub-questions are considered that jointly attempt to address the main research question:

Q1: What is the difference between enterprise software systems and traditional software systems?

Q2: To what extent is the traditional analysis process applicable to enterprise software system integration?

Q3: To what extent are deduction-based architectural analysis methods applicable to enterprise software system integration?

Q4: To what extent are induction-based architectural analysis methods applicable to enterprise software system integration?

Q1 is the natural starting point of the research. Since the main approach is based on the differentiation of enterprise software systems from traditional software systems, it is necessary to detail the differences between the two system types.

Q2 concerns the process view of architectural analysis. The analysis process referred to as "traditional" is generic in its nature, in the sense that it is not dependent on the specific analysis methods [Kaz98]. It is however designed for the context of a traditional developer organization, motivating an exploration of its applicability to the enterprise software system context. Although integration is employed as a suitable delimitation, the main issue is the process.

Q3 and Q4 concern the artifacts of software architecture, the architectural descriptions, and their associated reasoning techniques. Within the discipline of software architecture, there is a distinction between informal and formal approaches. Consequently, also the analysis techniques are divided into formal and informal techniques. The approaches are in much as the rationalists and the empiricists were in the 17th century. Formal methods attempt to apply deduction-based reasoning on well-defined specifications, using inference rules to assess properties of the described system [Cla96]. An important kind of informal method of architectural analysis is based on architectural styles (patterns) [Bus96]. Architectural styles for analysis employs induction-based reasoning, categorizing system types and searching for generalizable properties of the categories. Because of their differences, the two approaches are considered separately in the thesis.

The area covered by the thesis is large. Firstly, when delimiting one subject by intersecting it with another, the contextual setting increases by the introduction of the second subject. In this thesis, three subject areas, all deserving their own disciplines, are considered. Secondly, even the intersection per se is extensive. A conclusive treatise on the present subject is beyond the reach of this single thesis. Therefore, the thesis may to some extent be viewed as a probe, searching for solid ground and soft spots within the area.

An important delimitation is related to the context from which the research has drawn its empirical substance. The thesis has mainly found its material in mid-sized companies in the Swedish electricity industry. Due to recent legislative reforms, the electricity industry has undergone considerable structural changes. These changes have mainly benefited the research, as they have both redirected and increased software-related activities of electric utilities. Although the empirical material is from a limited sector, the results of the research could in principle be applicable to similar companies in other industrial sectors. More important than the industrial sector, are presumably the types of software systems and the software management latitude of the considered organizations.

1.3 RELATED WORKS

Reviewed in later parts of the thesis, there is an abundance of literature on software integration. This is also the case with software architecture, architectural analysis and to some extent literature related to enterprise software systems.

This project is not alone in considering intersections of the subjects. Some literature has been written on the concept of architectural mismatch [Gar94a] [Abd96] [Gac98] [Com99], which is a type of integration problem. Related to this are classifications of architectural component properties required for successful integration [Kaz97] [Sha95a] [Yak99a] [Yak99b]. With a foundation in coordination theory [Mal94], Dellarocas has proposed an architecture-based method for integration solution design [Del96] [Del97a] [Del97b]. There is also some literature presenting specific architectures or architectural styles of integration solutions [Emm01] [Sou01] [Gam98] [Bus96] [Sch00]. Related is the employment of architecture specifications in the design of integration solutions [Gann00].

Furthermore, much, if not most, of the literature on software architecture is associated with software integration in the sense that architecture mainly is concerned with interacting components. Thus, although the main focus of most of the literature on software architecture is not on integration, it is to some extent inherent in the concept.

For the fundamental concepts of software architecture, the present work leans heavily on Garlan and Shaw's much-referred book on software architecture [Sha96a] as well as Robert Allen's thesis [All97]. Although the present text does not agree completely on all accounts, the applied approach to architectural analysis as presented by the people of the Software Engineering Institute [Bas98] [Bac00] [Bas01a], has constituted a major influence. This is also the case for [Bus96] and [Sch00] in the context of architectural styles. Linthicum's guide to enterprise application integration [Lin00] has served as a convenient overview of the practicalities of a trade otherwise best described in a multitude of product guides. A fruitful way of thinking of the people involved in the architectural process has been found in Herbert Simon's work on administrative behavior [Sim47]. Further influences are of course many and appear throughout the thesis.

Due to the legacy of the Department of Industrial Information and Control Systems, the general position of the thesis is applied rather than theoretical. Although several formal approaches are discussed in some detail in the thesis, contributions that are industrially viable in the short term are preferred to those that have a longer time horizon. In terms of literature, this position is represented by [Bas98], [Bus96], [Del96], [Hei01a], [Hof99], [Sch00] and [Wal01] rather than [Abd96], [All97], [Gac98], [Luc95a], [Med00], [Moo95], [Mor95] and [Sch86]. Nonetheless, much of the foundations of the thesis are attributable to the latter category.

1.4 CONTRIBUTION

The present work has the following contributions:

Q1: Significant differences between software systems traditionally considered in the software architecture discipline and enterprise software systems are considered in Paper A and Paper B as well as in Section 5.7.

Q2: An evaluation of the applicability of traditional architectural analysis processes to enterprise software system integration analysis is presented in Paper B.

An adaptation of the process to the enterprise software system context is proposed.

A process for selection of integration solution based on architectural integration styles is presented in Paper D.

Q3: A classification of integration issues and an evaluation of the applicability of deduction-based architectural analysis methods for the enterprise software system integration context is presented in Chapters 4 and 6 respectively.

Paper C explores the credibility of deduction-based analysis methods when assumptions of correct inter-specification transformations are relaxed.

Q4: Induction-based analysis in the form of architectural integration styles for enterprise software systems is proposed in Paper D.

1.5 OUTLINE

The thesis is composed of two parts. The first part consists of the included articles. The second part is an "extended introduction", consisting of nine chapters. In Chapter 7, the articles are summarized, in this section, the first part of the thesis is outlined.

Chapter 2 discusses the methodology employed in the research.

Chapter 3 considers the empirical setting of the research. The structure and development of the Swedish electricity industry is very briefly reviewed, and the current software milieu of electric utilities is presented.

Chapter 4 discusses software system integration. The chapter reviews three common approaches to software integration. In analogy to political science, the three sections describe the integration problems encountered and the solutions employed in a world dominated by 1) a monarchical software developer, 2) oligarchical software developers, and 3) anarchical software developers. This review subsequently leads to a framework of integration issues. The framework is used in Chapter 6 to evaluate deduction-based architectural analysis methods with respect to enterprise software system integration.

Chapter 5 considers software architecture. The first part of the chapter considers software architecture in general, describing the main concepts and presenting the view of the author. These concepts include the definition of software architecture, architectural views, components, connectors, architectural styles, architectural description languages, and finally some comments on architecture in the software process. These are all traditional issues and mainly used as a setting for the remaining chapters. The second part of the chapter, considers the concept of enterprise software architecture. It is presented in relation to traditional software architecture and subsequently briefly compared to the fields of component-based software engineering and enterprise application integration.

Chapter 6 considers architectural analysis methods. Although induction-based architectural analysis is discussed, the main concern of the chapter is deduction-based analysis. A number of analysis methods are reviewed. The extent to which the methods address software system integration is considered using the integration issue framework from the third chapter. The chapter concludes by a discussion on the applicability of the methods to the enterprise software system context.

Chapter 7 presents summaries of the included articles. Chapter 8 and 9 contain the conclusions and further works respectively.

Chapter 2
Methodology

2.1 RESEARCH IN SOFTWARE ENGINEERING

The study of software is a science of the artificial [Sim69], and as such perhaps the prime example. A computing machine is an approximation of a model that per se has no link to the natural world. It is the job of hardware manufacturers to create machines that approximate the model as well as possible, and it is the job of software developers to build new models upon the computing abstraction [Dij76]. The linkage between the software and natural world is thus weak.

The sciences that deal with the foundations of the natural world have surprised mankind with revelations of the reductionistic cleanliness of the world in which we live. A remarkable number of phenomena may be described with comparatively simple and elegant formulae. It has thus become a basic tenet of these research disciplines to search for the simple (cf. Occam's razor [Bri02]). The sciences that deal with the foundations of life have also found a fundamental principle to guide the research, the principle of evolutionary rationality [Cam86]. If a species has a good sense of smell, there is a reason; if it has oversized teeth, this too has an explanation.

The sciences of the artificial, however, are still searching for their guiding principle. A fundamental problem when studying that which is man-made, is bounded rationality [Sim47]. In the life sciences, evolution is a guarantee that biological machines are created rationally (at least for some environment). Although an underlying rationality is often perceivable in man-made artifacts, there is no guarantee that this rationality is not occasionally flawed. As Edsgar Dijkstra [Dij69] so elegantly puts it in the context of software development, "As a slow witted human being I have a very small head."

Nor can the artificial sciences find support for the idiom of simplicity permeating, for instance, physics. In much the same way as with the rationality, simplicity is often found in human constructions. This simplicity is, however, only a symptom of the designer's sense of aesthetics or willingness (or unwillingness) to think harder. It can therefore not be assumed that e.g. software is based on an inherently simple or beautiful design; on the contrary, irrational complexity is oftentimes found in software artifacts. Research on software is thus in a kind of limbo. Software can neither be assumed rational nor simple, although both of these traits seem to permeate the subject.

In software architecture, this dilemma becomes evident, since an architectural abstraction implicitly is based on assumptions of simplicity and rationality of the modeled system. Slightly more concretely expressed, an architectural description assumes some kind of homogeneity, for example that all components of a certain type have significant similarities. If the case were the opposite, the description would be meaningless. Nevertheless, these assumptions are not always correct. Because of the bounded rationality and bounded simplicity permeating software systems, such assumptions are almost always questionable. One might say that in the artificial sciences, the devil is in the details.

Considering software engineering [Boe81] [Bro75] [Pre00], this is a hybrid discipline mixing the study of organizational issues and happenings with the study of inanimate objects and their properties. And as mentioned above, the inanimate research objects are artifacts, and as such the products of human design activities. This twilight zone between the soft and hard sciences, between the artificial and the natural, contains a difficulty as to research paradigm. It is generally recognized that the traditional scientific methods of the natural sciences are insufficient, but the general heritage of the research community is one of positivism, often inspiring a sense of discomfort with too hermeneutic, relativistic or post-modern views on science [Sea99]. Further complicating the picture is the obvious link to engineering and engineering methodology.

When explicitly stated, research methods in software engineering are oftentimes case studies [Mur99] [Ben02], occasionally experiments [Bas96], and rarely surveys [Wan98]. Although seldom considered in research methodology literature, the by far most popular method, however, seems to be design and development of new artifacts, i.e. prototyping. This is traditionally considered an engineering method rather than a scientific one. Nevertheless, it is the preferred method of the software engineering research community.

2.2 RESEARCH METHODOLOGY

Research methodology may be categorized into analysis methodology and data collection methodology. The analysis methods employed in the present work are discussed in the next section as well as in the included articles. The empirical data on which this thesis is based is primarily collected in case studies. This section considers some issues on the topic of case studies. The next section presents the research designs of the individual contributions.

Case study research is by its proponents, e.g. [Yin96], considered especially appropriate when the studied phenomenon is inseparable from its environment and the environment is difficult to control. This is typically the case in organizational contexts such as software projects. Case studies are typically categorized according to the available theory base and theoretical intention into exploratory, descriptive, explanatory. Briefly, exploratory case studies are aimed at investigating poorly known phenomena, thereby generating theory. Descriptive case studies employed theory for classification of observations. Explanatory case studies are aimed at testing theories by proving or disproving relations between phenomena. Since the conducted case studies mainly were of an exploratory nature, the descriptive and explanatory types will not be elaborated on herein.

The main criticism of case studies as research methodology is related to validity and generalizability [Yin96]. The validity problems of case studies are primarily based on the uncontrolled environment; it is typically not possible to repeat a case study. This is a weakness, perhaps not of the research approach, but of the state of the world; it is more often than not impossible to step into the same river twice. The standard case study approach to mitigate this problem is to document the study with care. Although this procedure introduces a certain stringency and allows a review to a certain point, it does not remove the problem completely: even the most meticulously documented case study cannot be repeated.

Furthermore, the validity of case studies may be compromised by ethical considerations [Har94] [Wad94]. Most case studies in software engineering are performed in industry. Companies on competitive markets are, however, often hesitant to the public and uncontrolled distribution of information on their processes and products that may be the result of case studies. By restricted disclosure of sensitive information, this threat is reduced at the cost of validity.

Finally, participatory case studies, where the investigator actively participates in shaping the studied events, run the risk of bias [Kin94]. Participation has the great benefit of providing ample insight into the studied project, but the validity of the study may be compromised by an investigator consciously or unconsciously pursuing certain research results.

The argument of generalizability is based on the small number of similar entities that are normally investigated in case studies [Kin94]. Case study research as a term is almost synonymous with one-occurrence phenomena investigations. The main question is whether it is reasonable to assume that an observation in a singular case can be generalized to other cases, and if so to what other cases. This question is a difficult one and the answer is partially found in theory. As an example, the theory of gravity allows us to generalize over objects when observing their acceleration in a free fall. If we have determined the acceleration with certainty for one object, we may assume that this acceleration is also applicable to other objects (on the same planet). In complex cases where a multitude of facts, theories and credible hypotheses are intertwined, the issue of generalizability is often further complicated. A case study should not be compared with a singular response in a statistical survey, for instance. The case study may include a great amount of related information, and although the information will not be comparable in a statistical sense, general conclusions may be drawn by other than statistical means (as exemplified above).

Thus, case studies are associated with a number of inherent problems, but much due to the failure of the methods traditionally employed in the natural sciences, e.g. experiments and statistical studies, the number of reasonably positivistic alternatives are often few.

2.3 Research designs

The present work has been of a fairly explorative nature with the general aim of investigating the applicability of software architecture analysis to enterprise software system integration. Although the research designs for the individual contributions have been explicitly considered in each case, the general research design for the thesis as a whole has changed as the work has progressed. In this section, the sub-questions of the thesis are commented upon as to chosen research methods.

Q1 considers the differences between enterprise software systems and traditional software systems. Although these differences have been continually debated throughout the course of the thesis work, the main approach to answering the question has been a combination of literature reviews and industrial case studies. Literature on software architecture (as presented elsewhere in the thesis) has provided an image of the systems traditionally considered in the field as well as aspects relevant for the comparison. A number of case studies conducted by researchers at the department [Eri93] [And98] [Joh99] [And00a] [And00b] [Hel00] [And01a] [And01b] constitute the main base for the characteristics of enterprise software systems. This empirical information is strengthened by literature, as presented in Chapter 3 and Section 5.7.

Q2 considers to what extent the traditional analysis process (exemplified in e.g. [Kaz94a]) is applicable to enterprise software system integration analysis. The question was approached as a comparison of the results of Q1 and the traditional analysis process. This comparison generated a number of issues where the traditional process appeared inadequate. For each inadequacy, an adaptation to the process was proposed. In parallel, the evolving adapted process was applied in a participatory case study as reported in Paper B and [And01b]. The experiences of the study further refined the process proposition. The parallel development and application of the process allowed for a certain amount of iteration, but weakened the validating function of the case study.

Q3 is concerned with the extent to which formal architectural descriptions and analysis methods are applicable to enterprise software system integration. Chapter 4 presents a categorization of issues managed by general integration technologies. These issues are in Chapter 6 compared to formal, deduction-based architecture analysis methods available in literature. The comparison thus provides a view of the concepts that are addressed by these methods, as well as a view of those issues that are not. The methods and issues are further considered in relation to the results of Q1, thus setting the question in the context of enterprise software systems. In Paper C, the credibility of these deduction-based methods is considered with respect to inter-specification transformation distortions. The study was performed in the confines of a small controlled system development project conducted at the department. The specifics of the project are detailed in Paper C.

Q4 considers the extent to which induction-based architectural analysis methods are applicable to enterprise software system integration. In Paper D, considering architectural styles, the modus operandi was similar to that of Paper B. A comparison between the results of Q1 and traditional architectural styles lead to the proposition of a number of adaptations to the way in which architectural styles are described and employed. From literature (e.g. [Lin00]) and industrial studies previously performed by the department, a number of architectural integration styles were elicited. Finally, the approach was exemplified by applying it to experiences from a participatory case study performed by researchers from the department. The case study thus served with useful exemplifying empirical data, but did not fill a validating function in the research.

Papers A, B and D were written in close cooperation with Jonas Andersson; the specific individual contributions are therefore difficult to separate. However, Jonas Andersson's work [And02] is concerned with the concept of software system modification while the present author considers software system integration. Further differentiating Andersson's work from the present is his explicit focus on the aspects of time and evolution of enterprise software systems. Finally, the present work includes deduction-based (formal) approaches to architectural analysis. Paper C was also written in close cooperation with the co-author, Mathias Ekstedt. The authors have contributed equally to this paper. In all papers, the names of the authors are alphabetically ordered.

2.4 RESEARCH EVOLUTION

The present thesis is the result of a journey that has traversed more topics than those presented herein. Figure 1 illustrates these disciplines in relation to the focus of the thesis.

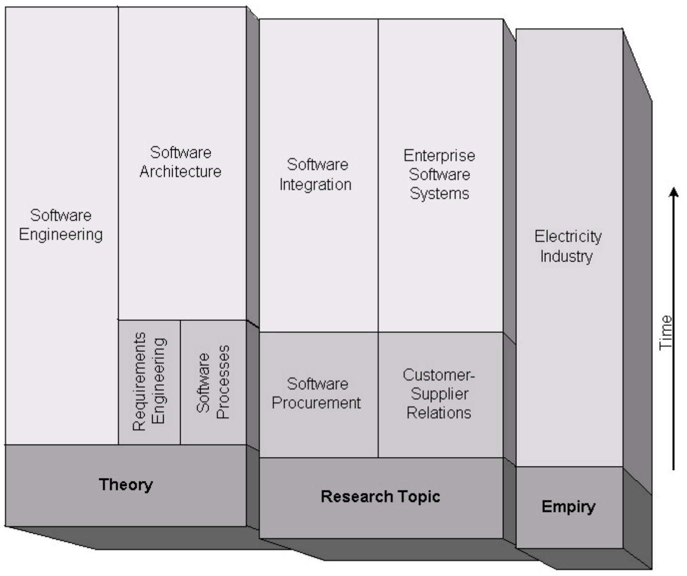

Figure 1. Evolution of thesis work.

The research topic has during the thesis project shifted from an initial focus on software procurement projects and customer-supplier relations to software integration and enterprise software systems. Several of the papers that are not included in the thesis are concerned with procurement projects and their characteristics [And98] [And99] [Joh99]. Although the areas at first glance may seem to have few connections, many issues are tightly linked. In particular, there is a tight coupling between component procurement and system integration and management. The procurement of enterprise software components determines not only the base components that are to be integrated and managed, but also the organizational latitude of subsequent enterprise software system management.

The literature considered during the project has had a clear base in software engineering. However, as the research topic has been adjusted, so has the relevant literature; primarily from requirements engineering and software processes to software architecture. In the investigations on procurement projects, also theory on inter-organizational relations has been considered.

The empirical context has remained constant during the project, namely the electricity industry. As a part of the Energy Systems Program, the work is founded in a fairly extensive study of the Swedish energy system, as provided by the program. This wide base has in subsequent studies been narrowed to the electricity industry.

Chapter 3

Software Milieu of Electric Utilities

3.1 INTRODUCTION

Due to the empirical context of the included articles, this chapter provides a description of the software situation within the Swedish power industry. The chapter considers the deregulation in general and the resulting structure of the electricity market. This is followed by a short review of three drivers of software system evolution that have resulted from the deregulation. The next section contains a review of software systems typical for electric utilities. The chapter is concluded by a short discussion on the current needs for software integration in electric utilities.

3.2 THE SWEDISH ELECTRICITY INDUSTRY

After an approximate century of monopolistic governance, the Swedish electricity market was deregulated[1]. This section considers the deregulation and the structure of the new market in terms of its actors.

3.2.1 A DEREGULATED INDUSTRY

The previously mainly monopolistic and integrated structure of the power industry was in 1996 divided into two segments based on two different regimes. Ownership and management of the electric grid maintained a monopolistic foundation, while the structure for electricity retailing and trading was transformed into a market-based form of governance. As a direct consequence of this legislative reform, the formerly integrated electric utilities have been split into several companies, abiding by fundamentally different rule sets. Under the supervision of the Swedish National Energy Administration are the still-monopolistic network operation companies. Under the new market regime are the generation companies,

[1] It is occasionally pointed out that the "deregulation" was in fact a "re-regulation", since the market still is regulated. In this text, "deregulation" is used, as it is the commonly employed term.

the electricity retailing companies and the electricity trading companies. Most of the European countries have undergone similar changes.

3.2.2 ACTORS

The electricity business is by the new regulations viewed as divided into a market and a network segment.

Distribution companies. Since the cost of networks is high, the network business is considered a natural monopoly. The monopoly is mainly of a local nature in the sense that distribution (or network) companies have geographical concession areas, covering, e.g., a municipality. Furthermore, the distribution companies are private and profit seeking. Approximately 1500 in the fifties, the number of distribution companies have been reduced to 200 during the last half-century. Previously a consequence of municipal mergers, the continuing reduction is now mainly motivated by the deregulation [Ene01]. Thus, the traditional distribution companies, defined by their municipal concession areas, are slowly being replaced by larger actors.

Regulatory agency. To avoid the unsound pricing that typically results from profit-seeking monopolies, it is the responsibility of *the Swedish National Energy Administration* to supervise operations of the distribution companies.

System operator. Transmission is mainly managed by the national system operator *Svenska Kraftnät* (SvK). This state agency is responsible not only for managing electricity transmission on the national grid, but also for upholding the balance between electricity production and consumption as well as import and export to and from the national network. To control the balance, Svenska Kraftnät provides a *regulation market*, where producers may bid for increases or decreases in power production.

Generation companies. The generation, trading and retail companies are active in the market-driven segment. Generation companies are typically parts of major energy companies. Together, Vattenfall and Sydkraft are responsible for approximately 70% of the total Swedish electricity production.

Trading companies. Electricity is bought and sold either according to bilateral contracts between companies, on *NordPool*, the Nordic market place for electricity trade, or on the system operator's regulation market. Electricity is a commodity with some particular characteristics, making the trade slightly different from other markets. The most important characteristic of electricity in the context of a market place is the requirement of simultaneous production and consumption, since electricity cannot (in principle) be stored. This makes the financial trade of futures and other derivates important as a risk-distributing mechanism. Another interesting characteristic is the perfectly undifferentiated nature of electrons, making it possible to use the network as a giant pool into which producers pump and consumers suck electricity.

Retail companies. The retail companies are the electricity industry's primary faces towards the customers. No production is necessary for retailing, since electricity can be bought both on NordPool and by bilateral contracts with producers. Most retail companies are the market-governed offspring of the formerly integrated municipal electric utilities, with the distribution companies as their monopolistic sisters. The retailers have been subjected to a more extensive wave of mergers and acquisitions, and are therefore now fewer in number than the distribution companies.

Consumers. *Buildings and services* is the largest sector of consumption in Sweden, with an electricity use of an estimated 73 TWh in 2001. In comparison, the industrial sector consumed 55 TWh the same year [Ene01]. The main concept behind the deregulation is the establishment of consumer choice in selecting electricity supplier (retailer). Consumers consequently abide by the market regime in electricity consumption. Network access is, however and as mentioned, provided by regulated distribution companies.

3.3 DEREGULATION AND SOFTWARE SYSTEMS

The deregulation has created three primary drivers of software system evolution. Firstly, the escalated organizational dynamism of the industry, with an increase in the numbers of mergers and acquisitions has resulted in an increased number of uncoordinated enterprise software systems. To manage these, many retirement, and in particular integration projects have been initiated. Secondly, new business operations resulting from the reform have required new support systems. In particular, new routines for metering and settlement have created a demand for new software systems specific for the Swedish electricity market. Thirdly, increasing competition has lead to expectations of lower margins on electricity trade. These expectations have, in turn, led to a search for new business opportunities.

3.3.1 MERGERS AND ACQUISITIONS

The deregulation of the electricity industry in Sweden and much of Europe has both facilitated and created an incentive for national and international mergers and acquisitions. This has coincided with a general privatization of Swedish municipality-owned companies. In combination, these two factors have turned a formerly stable industry into turbulent and dynamic grounds in search of a new structure.

Only in 2000/2001, the Finish company Fortum stepped up from 50% to 100% ownership of Birka Energi. Vattenfall invested in a third of the stock of the in the German company Hamburgische Elektricitäts-Werke (HEW), 49% of the Berlin energy company Bewag, as well as acquiring the Polish distribution company GZE, German Veag, Norweigan Oslo Energi, Swedish Uppsala Energi and the network part of Sigtuna Energi [Dag01a] [Dag01b] [Dag01c] [Dag00a] [Dag00b] [Dag00c] [Dag00d]. With German EON as a new majority owner, Sydkraft invested in Norrköping Miljö och Energi, WM Sverige and Nora Energi [Dag00f] [Dag00g] [Tid00a]. Birka Energi acquired Arvika Energi and Graninge attempted to acquire Norrtälje Energi [Tid00b] [Tid01]. The French company Electricité de France increased its share in Graninge to 35% [Dag00h].

As will be discussed below, the considerable number of mergers and acquisitions in the industry have generated a need for management of the resulting uncoordinated systems.

3.3.2 NEW BUSINESS OPERATIONS

Not only the organizational movements in the wake of the deregulation have influenced the software situation for the electricity companies, but also the new market operation.

Trading of electricity on the spot and regulation market as well as financial trade on the derivative market have required new systems for all of the major actors in the industry. An open market requires new customer relations management systems for gaining new and keeping old customers. A new regulative system for maintaining the balance of the electricity system demands new systems for the system operator. The division of the formerly integrated electric utilities into network and retail companies has necessitated a division of customers and thus of customer information systems. The list of new support system requirements goes on. Here, we highlight the electricity metering issue, and some consequences it has had on the software situation in the industry.

The procedures for buying and selling electricity are complicated by several circumstances. The basic problem is how to measure the amount of electric energy each producer produces and each consumer consumes. Because the price changes continually with the supply and demand, the measurements need to be performed each hour. Before the deregulation, consumers were dependent on the electricity company owning the concession for the geographical area in which the customer was located. The measurement and settlement problem was thus an internal problem of the electricity company, being both retailer and network owner. After the deregulation, customers can choose any retailer. The concession-owning network company now needs to measure the consumers' energy consumption (as well as any generator's production within the concession area), make sure that the measurements are correct, estimate them if they were not registered, compare the inflow and outflow of power into and out of its network with neighboring network companies, and finally distribute the information to representatives of the retailers (actors responsible for keeping the balance), and to Svenska Kraftnät. These measurement and settlement procedures have proven to be difficult to implement and the primary problem has been related to the software systems needed to perform the required tasks.

Examples of the specific problems encountered by the network companies are [Ene00]:

- The hourly metering of the many low-consumption customers has been complicated by the fact that the meters have been too costly to produce. After many twists and turns, it has finally been decided that these metering values can be estimated instead of measured.

- Problems of estimating missing meter values due to a lack of estimation functionality in the metering and settlement systems have been reported. These problems have required manual interventions that have been both costly and time-consuming, delaying the whole settlement procedure.

- After the decision to allow estimation of the meter values of low-consumption customers, this estimation in itself has become problematic due to a lack of functionality in the metering and settlement systems.

- Problems of localizing consumption points in the network due to problems of asset identification have been reported. This has typically been attributed to a lack of integration between metering and settlement systems on the one hand and customer information and asset management systems on the other.

- There have been problems managing customers attempting to switch from one retailer to another. These problems are to be expected, since every organization, and thus every information system previously managed only its own customers. Functionality for retailer switching and the required inter-organizational communication procedures has thus been lacking.

- Many actors have encountered communication problems with external actors. Although the communication of detailed metering information is new per se, problems are mainly due to new message format standards. According to regulations, the format Ediel [SvK02] should be employed for measurement information, registrations of customers changing retailers, production predictions, etc.

The problems of electricity metering have thus been partly due to a lack of software system functionality, and partly due to software integration problems.

3.3.3 NEW BUSINESS OPPORTUNITIES

For the segment of the electricity industry that is experiencing competition as a result of the deregulation, lower margins are expected. To increase the potential for profit, these companies – in particular those with an interface to the consumers – have been searching for completely new business opportunities [Bäc98]. Electricity companies have for instance attempted to move into telephony, broadband access, insurance, and home automation markets. Of course, all such moves have required new sets of operation support systems.

3.4 SOFTWARE SYSTEMS IN THE ELECTRICITY INDUSTRY

Any attempt to describe the software systems of a complete industrial sector is doomed to be grossly incomplete. This section can at best conjure a vague image of the software milieu of the electricity industry.

Software systems are often divided into *operation support systems* and *business support systems* [Hag02]. This division is primarily due to historic reasons. Companies developing general (non-digital) machines for management of, e.g., the electricity process (machines such as generators and transformers), successively added digital computing capabilities for control and supervision of the process and its machines. Due to their environment and tasks, these operation support systems are typically characterized by real-time performance requirements, high reliability, availability, robustness and safety. In contrast to operation support systems, the development of business support systems fairly early grew into an industry in its own right, decoupled from the specificities of the user organizations. These systems are stereotypically characterized as mainly batch processing, with requirements on high throughput rather than real-time performance. Requirements on extreme availability or robustness do not belong to the general characterization of these systems. Although these requirements may explain how the two system classes came to be, the evolution of both the systems and the context in which they are used have greatly diffused the borders between them.

In this section, typical systems employed in the electricity industry are presented in an (intuitive) order from “true” operation support systems to “true” business support systems [Ceg86] [Che97] [Eng99] [Hag02] [Pit01]. For many of the systems, there is an ongoing discussion concerning terminology, debating what functions should be included under what system names. In this section, no justifications are provided to the categorization, and it is of little importance.

Local monitoring and control systems. Software-based systems employed for local, low-level monitoring and control are used in conjunction with a great number of devices in the power process. In their most basic form, these systems collect process data, relay it to some centrally located agents, and implement instructions received from the central systems. Additionally, local systems may be used for data buffering, time stamping and data filtering. Process control operations that do not require central system intervention and that are time-critical may be directly controlled by a local system. In the power system, local systems are used for data acquisition and control of devices such as protection relays, capacitor banks, breakers, automatic reclosers, sectionalizers, and more. According to Ericsson and Rahkonen [Eri95], the interfaces provided by these local power control systems are to a great extent proprietary.

Central monitoring and control systems. Providing real-time data acquisition and remote control, SCADA (Supervisory Control And Data Acquisition) systems are normally considered the core controllers of the power system. These systems collect data from widely geographically distributed local systems, present relevant information to the operators, relay operator commands to the local systems, analyze the state of the process, react to anomalies by automatic control of local systems and operator alarming. Additionally, SCADA systems store data, and more. To the central monitoring and control systems, load management systems should be included. These systems may be used to centrally control electricity consumption. Between control center applications, communication standards are typically proprietary, while significant harmonization has been achieved between control centers in the form of standards such as ELCOM and ICCP [Eri95].

Automatic meter reading systems. Data about the consumption and generation of electric energy is a necessary base not only for generation and load control, but also for the economic transactions related to the use of electricity. According to regulations, this information should be collected for each (larger) consumer and generator, each hour. Due to the deregulation, this data collection system has recently become centralized and more extensive. Because of the geographical distribution of electricity consumers, the communication between electricity meters and the central system is managed in a number of innovative ways, including the use of telephone lines, power lines, mobile telephony, and satellites as communication infrastructure. A standard for electronic interchange of (among other things) metering values is currently employed in Sweden, Ediel. Ediel is based on the EDIFACT (Electronic Data Interchange for Administration, Commerce and Transport) standard for message presentation, and the X.400 electronic mail protocol for transmission. Under X.400, several lower-layer communication facilities are allowed. Ediel is jointly developed by the Scandinavian electricity industry in a standardization committee called EDIEL Nordic Forum.

Trading systems. Both the NordPool spot market and the Svenska Kraftnät regulation market provide possibilities of trading electricity on short notice (from half an hour to a day ahead). Since these kinds of market places were not relevant before the deregulation, these systems have been developed fairly recently. These systems are closely related to systems for trade of futures and other financial instruments employed for financial risk management. Ediel is employed as a message standard for communication of bids etc.

Settlement systems. In the deregulated market, the economic settlement following the generation and consumption of electricity is a complicated affair. Hourly metering values are collected by network owners, missing values are estimated, values are controlled, compared with neighboring network owner's values, distributed to actors with balance-keeping responsibilities as well as to the system operator. Balance responsible actors as well as the system operator control and perform calculations upon the metering values to determine the total consumption, consumption per balance responsible, etc. Values calculated by different actors are compared to each other; inconsistencies require backtracking and error location schemes. The whole process is so complicated that it has taken several years for the industry just to comply with the coarsest directives of the regulations [Ene00]. A set of the Ediel standard is used for settlement data transfer between actors.

Distribution and production management systems. Typically operating on top of the central monitoring and control systems, distribution management systems include decision support systems for operation and control, trouble call analysis and management systems, trouble crew dispatch, and similar pseudo-real-time functionality.

Geographical information systems. Because the power process is widely geographically distributed, there is a need for systems linking assets and devices to a geographical position. This is the purpose of the geographical information systems (GIS).

Planning and engineering systems. Planning and engineering systems are employed to analyze, optimize, modify and plan operation and maintenance. These systems may perform different kinds of network analyses, forecast loads, predict reservoir level development for hydro power, simulate trade markets, etc. They may also support system design, for instance with CAD tools.

Work management systems. Systems are available for keeping track of the work flow, including functions for work planning, estimation, orders, progress, resources, contractors, etc. When a job is completed, the work order is closed and reported to other involved systems.

Customer information systems. Customer information systems are employed to keep track of the customers. This includes contract management, supply points billing, payment control, and so on. According to general wisdom, the more the company knows about its customers, the likelier it is to keep them. Therefore, successful integration of the customer information system with metering systems, GIS systems, trouble call systems, is considered a competitive advantage.

Asset management systems. There are systems for keeping track of the company's assets. These systems typically contain information about the types of assets, their age, condition, when they were last serviced, etc.

Enterprise resource planning systems. Enterprise resource planning systems are intended as company-wide administrative systems for managing everything from accounting to human resources, including customer information systems, procurement tracking systems, payroll management systems, time management systems, project and program management systems, quality management systems, and more. Several of the systems presented above can be located under the umbrella of the enterprise resource system. Enterprise Resource Planning Systems have long been criticized for their non-standardized communication mechanisms and have only recently begun opening up their interfaces. The market leader, SAP's (Systeme, Anwendungen, Produkte in der Datenverarbeitung) system, employs an old IBM middleware layer called CPI-C for program-to-program communication. In an attempt to hide the complexities of CPI-C, an in-house mechanism called RFC (Remote Function Call) has been conceived. On top of RFC, an object-oriented mechanism called BAPI (Business Application Programming Interface) is available. SAP also defines a message format, similar to EDI but proprietary, called IDOC (Intermediate Document) [Lin00]. To a large extent due to a lack of standardization, much of the integration between non-real-time systems is based on flat file transfer, where data files are exported (possibly employing customized data extraction software) from the source application, transported by e.g. FTP (File Transfer Protocol) batch converted, and subsequently imported (possibly using load programs) into the destination system.

3.5 A NEED FOR SOFTWARE INTEGRATION

Formerly common, idiosyncratic green-field development of organization-specific systems is nowadays highly unusual. For all of the above system types, there are today suppliers with more or less complete product packages. These may be used as a base when introducing new functionality into the organization. Customization of the base products in delivery projects is, however, a common activity, and often used by the developing organizations to add functionality to their products.

A major task for electric utilities has then become to introduce and evolve a large set of products provided by uncoordinated suppliers. The software system management of the utilities is not so much concerned with development of new functionality, since this normally is already available in some supplier's product. Instead, the selection of products and suppliers, the introduction of new products into the existing enterprise-wide software system, and the evolution of this enterprise-wide system, become the primary tasks of the electric utility.

In the old days, systems were built as "stovepipes", i.e., they were not designed to interact with other systems. This was reasonable at the time, since the need for interaction was unrecognized. During the last decades, the demands for software system integration have, however, become much more urgent. It has become evident that the systems presented above have a great number of relations to each other; in particular, the data present in one system may be of interest to another system.

As discussed above, the deregulation has added to these general integration demands. Firstly, the increase in mergers and acquisitions has created a need for software system harmonization in the companies resulting from the organizational mergers. Secondly, the new regime has increased the need for communication, both internally, within companies, and between organizations. Thirdly, new business opportunities have resulted in investments in completely new systems, which also need to be integrated into the enterprise software system.

As a closing example, we consider the network owner's metering system, it's data, and where it may be of use. The data is relevant to the settlement system as well as to neighboring network owner's, system operator's, balance-keeping actor's settlement systems; the data is also relevant to the retailers billing systems and customer information systems. Furthermore, metering data is related to a metering device. The metering device is registered in the asset management system, so it might be useful to have some relation between these systems. Furthermore, like most assets, the meter has a geographical position, which qualifies it for the GIS system. Assets associated with geographical information are in turn useful for the work management system. The load forecasting of the planning system requires historical load data, which is collected by the metering system. The results of the load forecasting are relevant to the distribution management system as well as for other planning systems. And so on.

The requirements as well as the possibilities for meaningful software integration are thus many. In the next chapter, the problems encountered when attempting to integrate software systems and proposed approaches to these problems are discussed in some detail.

Chapter 4
Software Integration

4.1 INTRODUCTION

This chapter reviews the concept of software integration as presented in literature. The main purpose of the chapter is to provide a base for the subsequent evaluation of the applicability of deduction-based software architecture analysis techniques to enterprise software system integration. As presented in Chapter 6, a number of methods for architectural analysis of integration-related issues are available. By reviewing a set of typical integration technologies, this chapter aims at providing a categorization of the issues that need to be managed in enterprise software system integration projects. The underlying assumption of this approach is that those issues that need to be managed for software integration in fact are managed by at least one of the technologies presented herein. It is therefore not important that all available technologies for software integration are reviewed (an impossible task), but a sufficient number and range to ensure that no important issues have been omitted. A supplemental purpose of the review is to provide a general overview of the field of software integration.

Three different perspectives on integration are considered, *monarchical* integration approaches, *oligarchical* integration approaches, and *anarchical* integration approaches. The reason for using these three perspectives is two-fold. Firstly, literature on software integration seems to be divided into three categories. Secondly, software integration is in much concerned with agreements between component developers, and there appears to be three common software integration situations depending on what agreements have been made between developers. In analogy to political science, the three sections describe the integration problems encountered and the solutions employed in a world dominated by a monarchical software developer, oligarchical software developers, and anarchical software developers.

Because of the versatility of automatic computing, the laws that govern software development are rarely related to physical limitations, but rather to rules created by people. For instance, in a multitasking operating system, one process can typically not directly access the address space of another process. This is not because it would be difficult to allow this, but because it is a restriction deemed useful by the community. Similar man-made rules determine how object-oriented objects can be accessed, how computers communicate over networks, etc. Integration of software systems is to a large degree concerned with understanding these rules, in which context they where created, whether they can be changed or not, or whether they can be sidestepped.

One way of viewing the three approaches of this chapter is by considering the assumptions they make about what is modifiable, when it is modifiable and by whom. The monarchical approach assumes that the same agent has complete access to all components all the time, and is able to synchronize their development. The oligarchical approach assumes that there is coordination between component developers before the components are designed. The anarchical approach assumes that there is no coordination between the components developers at all. The job of the integrator thus varies considerably between the approaches. A similar way of viewing the three approaches is in terms of agreements. In the monarchical scenario, there is no need of agreements, since only one actor is involved. In the oligarchical scenario, agreements between component developers constitute the base for integration. In the anarchical scenario, although agreements would have been beneficial, none are in place.

The section *Monarchical integration approaches* describes how software integration typically is achieved at a fairly fundamental technical level, considering concepts such as shared memory, interprocess communication, remote procedure calls etc. The concepts of the section are typically treated in literature on operating systems, compilers, programming languages, etc. [Dei84] [Aho86] [Bac87] [Tan87] [Tan95] [Mal84] [Foi85] [Rey98]. Most of the encountered software integration problems may be reduced to the problems presented in this section, and most problems are eventually solved with the corresponding techniques. The issues presented in this section do, however, only constitute one view on the problems and solutions of software integration. Important issues that are ignored are the results of organizational interaction, or lack thereof. In the monarchical scenario, the same actor is both component integrator and component developer; stereotypically thus, these issues concern the lonesome green-field programmer.

The section *Oligarchical integration approaches* presents the concepts of integration standards. The standards are considered in the Open Systems Interconnection reference model (OSI) of the International Organization for Standardization (ISO) [Tan81]. Integration standards are based on agreements among developers to employ a certain integration solution, not necessarily because it is the most efficient solution, but simply because it can be agreed upon. An unreasonable number of integration standards have been defined, and no single text can describe even a fraction of them. This section is therefore confined to consider these standards in a brief and conceptual manner, with some references to specific cases when appropriate. In the oligarchical scenario, different actors develop the components, but agree on the integration solution; stereotypically, these issues are relevant for cooperating, large-scale, green-field development organizations.

The section *Anarchical integration approaches* reviews methods for application integration. The standard literature on these methods is normally denoted Enterprise Application Integration (EAI), and concerns techniques and devices such as middleware, adapters, message brokers, etc. [But99] [Lin00] [McG00] [Lin01a] [Lin01b] [Mor01] [Ruh01]. Application integration is typically performed by integrators of independent third-party software components. Stereotypically, the anarchical view is relevant when components constructed by developers that never agreed on anything, are to be integrated. Although the anarchical approach is most commonly considered in the context of enterprise software system integration, the two other approaches are of equal importance.

The final section of the chapter compiles the issues managed by the considered approaches. As mentioned, this compilation is subsequently used as a framework in the chapter on architectural analysis approaches. Seven aspects, or categories, of software integration constitute the framework in this chapter. They need not to be considered as a final categorization of software integration issues, but together they should encompass the important issues.

4.2 INTEGRATION AND INTEGRABILITY

Before considering general approaches to software integration, it is useful to consider some definitions of common terms.

According to Bass et al. [Bas98], *integrability* is the ability to make the separately developed components of a system work correctly together. A special kind of integrability is (still according to Bass et al.) *interoperability*. Integrability measures the ability of parts of a system to work together; interoperability measures the ability of a group of parts (constituting a system) to work with another system. According to [Pol01], integration results in tightly coupled, while interoperability creates loosely coupled systems. The difference between the terms is thus fairly unclear. In the context of the present work, integrability is employed, partly since integration of enterprise software system components is considered (although these components often are viewed as systems in other contexts). Perhaps more importantly, interoperability has rather narrow connotations, primarily concerned with interface compatibility and the creation of a communication link between two software systems where previously there was none. Integrability is thus employed as a slightly broader term, also concerned with some of the overarching qualities of the resulting system (such as data consistency and security). In this text, *software integration* thus denotes the task of making separately developed components of (enterprise software) systems work correctly together, in analogy with the term integrability. In the end of this chapter, a framework attempting to operationalize the concepts of integration and integrability is presented.

4.3 MONARCHICAL INTEGRATION APPROACHES

In the monarchical development situation, the integrator is also the component developer. Being the developer of the components provides the integrator with an uncontested choice of integration solution. The totalitarian control over integration solution is thus in analogy with monarchical governance. Continuing with the analogy, the integrator follows the rules of the platform developer (typically including hardware, operating system, virtual machine or compiler, etc.). He is thus the subject of the platform's regulations, decided on beforehand, as were they the constitution of a monarchic state.

The issues considered in this section are mainly treated in computer science literature on operating system, compilers, and (imperative) programming languages [Dei84] [Aho86] [Tan87] [Bac87] [Tan95] [Mal84] [Foi85] [Rey98].

The section begins by the simplest communication between components – the communication between successive statements in a program – and then increases the granularity of the components. For each component type, the introduced communication problems are described and typical technologies to overcome them are presented. Issues that are not automatically managed by the platform become the concern of the developer; these responsibilities are also considered.

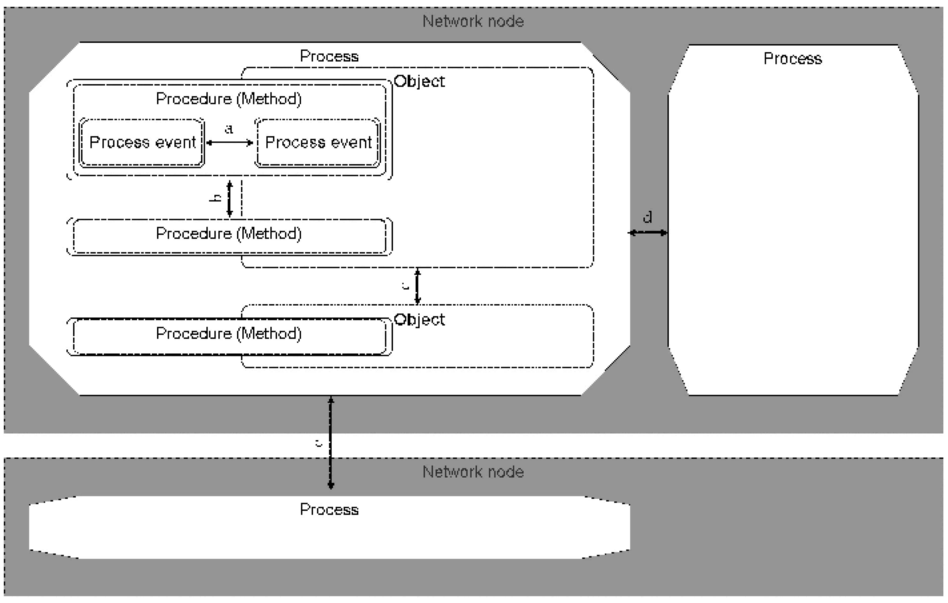

Figure 2. Monarchical integration on increasing levels of abstraction (notation according to [Bas98]).

Although elaborated on elsewhere in the thesis, for the purposes of this section, the term *component* represents an active software entity that is capable of some kind of communication with its peers (this is in line with [Sha96a]). Components are in the monarchical context defined by the supporting soft- and hardware. The semantics of objects, for instance, are defined by the compiler while processes are defined by the operating system. In this section, the supporting soft- and hardware is referred to as the *platform*. At first glance, this section may seem to ignore the integration between the components and the platform, which may be considered a component in its own right. However, the platform is to a large extent the creator of the component abstractions and the manager of the integration issues, and viewing it as a component would result in an unsound kind of nesting. The platform is thus addressed as an underlying abstraction creator and integration manager.

4.3.1 PROCESS EVENT INTEGRATION

The term *process event* is used here to denote the execution of a program statement[2]. Thus, as programs are composed of statements, processes are composed of events (cf. [Hoa85]). An event may be viewed as an atomic component; it is in this context hardly meaningful to imagine a smaller component (according to the above definition) than a single process event. Let us therefore, as a reference, begin by considering how process events communicate with each other (cf. Figure 2-a). There are two issues that need to be communicated from one event to the next, control (i.e., execution) and data.

Considering transfer of control, in the simplest of processes, the events follow each other sequentially in time as specified by the program. The program pointer, indicating the address of next the statement to be executed by the processor, is incremented for each executed statement, thereby transferring control from one statement to the next. When writing a simple program, the developer thus need not worry about how control is transferred from one event to the next, since the platform manages this. However, he does need to follow the convention of arranging the statements in the same order in program text space as he wants the process events to be executed in time. He also needs to follow the syntax conventions of platform-defined statements.

For data transfer, shared memory is employed. The execution of an assignment statement stores a value to an area of the memory region. The memory for the process is common to all statements; so two statements in the same program can access the same data if they know the address. The developer need not be concerned with how data is written to and read from memory. However, he is responsible for the format of the data. The developer also needs to ensure that the two communicating statements refer to the same data address space as well as the meaning of the data.

4.3.2 IN-PROCESS PROCEDURE INTEGRATION

A next level in component granularity considers procedures (subroutines or functions) as components. A procedure is a bundle of statements that can be invoked several times by the program. The procedure brings not only structure to a program but also economy,

[2] In this section, a very simple language is considered, without subroutines, jump statements or variables.

since repetition of program text may be avoided. Procedure integration (cf. Figure 2-b) considers how control and data is transferred to and from procedures[3]. Jumps and variables are therefore considered here.

The main problem of transferring control between components (in comparison with the previous example) is how to make the program counter jump from the process call to the process definition and back. When a procedure is invoked, its first statement will probably not be located exactly after the invoking call. In the simple program (above), the program counter is incremented for each executed statement, but clearly, when procedures are present the program counter needs to be able to point to other addresses. Furthermore, when the procedure has reached its final statement, the program counter should return to the statement following the invoking statement, again making a jump. Procedures are invoked by procedure calls. Automated control transfer in a procedure call is managed by the platform first storing the current value of the program counter in the process's stack, then changing the value of the program counter from the current address to the address of the first statement of the called procedure. When the called procedure is finished executing, the return address is returned to the program counter. This is also the typical way of performing system calls. In the context of procedure calls, the developer need not be concerned with the memory address of the invoked procedure, but only the proper procedure name. In this sense, the platform, taking the name as input and yielding address as output, implements a location directory service as well as actual control routing. The developer is restricted by the procedure call convention prescribing the complete execution of the procedure before returning to the statement following the procedure call. Furthermore, to make a procedure call control transfer, the developer needs to match the name of the procedure in the procedure definition and the procedure call. Any sequences beyond call-return are the responsibility of the developer (e.g., Call-return A must follow Call-return B).

Considering data transfer; in the process event integration scenario, data was accessed by direct memory addressing. This is problematic since the programmer needs to directly manage memory addresses. Instead of addressing memory directly, variables may be employed. These are normally both memory references and format directives. Additionally, a procedure may have local variables, it may be invoked using parameters, and it may return variables. In principle, all these variables could be global variables accessible to all statements in the program; this is however not desirable. Instead, local variables, residing in a reserved memory segment, are accessible only to the statements within the procedure. Procedure parameters and return variables thus need to be passed between the invoking statement and the procedure. This is accomplished by pushing these parameters onto a designated place in the stack, available to the called procedure. Either the parameter values or the parameter addresses are pushed onto the stack, call-by-value and call-by-reference respectively. Return variables are treated in a manner similar to parameters, i.e. using the

[3] The present discussion is equally applicable to system calls, i.e., procedures defined by the operating system.

stack. When the procedure is finished executing, the local variable and parameters are freed from the stack. The calling statement may assign the return variable to some (more) persistent variable. Summarizing the data transfer responsibility distribution between developer and platform, transport of data from call to procedure and back is thus managed by the platform (the stack is hidden from the developer). Furthermore, data transport is simplified by parameters and return variables. Using variables, the developer's responsibility over location is decreased since the platform implements a directory service, and data representation is standardized by the platform. The developer still does need to adhere to the representational conventions defined by the variable types, and ensure that the variable names (address references) are correct, designating the intended variables. To some extent, with the statements operating on variables (e.g. arithmetic operations on integers), the platform determines the meaning of certain data. The developer is further responsible for this meaning by any developer-defined operations.

4.3.3 IN-PROCESS OBJECT INTEGRATION

Objects are instantiated classes, and as such bundles of procedures (methods) and variables (attributes). Bundling is among other things intended to lead to higher cohesion and lower coupling [Bri97] [Bri99] [Ede94] [Hit95] in the program. Objects preferably communicate (cf. Figure 2-c) by means of method invocations. In (single-threaded) object-oriented systems, control is passed between components by method invocations, much in the same way as in procedure-oriented systems. When an object is instantiated, a special method (constructor) is automatically invoked. Attributes and methods may be accessible to other objects (public) or not (private). Although attributes in one object may be directly accessible by another object, it is generally preferred that local methods manipulate local data.

Method invocation, being the typical solution to control and data transfer between objects, basically functions in the same way as procedure calls. The main additional problem of control transfer between objects is related to object creation. Objects are dynamic and therefore do not have an address at compile time as procedures do. Since an object's address become known only at run-time, it needs to be transferred run-time to any object that wishes to invoke that particular object's methods or access its data. This problem is solved by the platform making object address references accessible to the creating object at the time of creation. The references are subsequently used for invoking methods in the object as well as for accessing data. Object references often are typed variables, which implies that the syntax is determined by the platform. Correctly representing the reference is however the developer's responsibility. Furthermore, run-time location, i.e., passing object references to the appropriate instances (other interested objects), is mainly the developer's task. Finally, some aspects of the sequence of interactions between objects are regulated by the platform, while others are the responsibility of the developer. For instance, an object must be created before its methods are invoked, but the order of method invocations between creation and destruction is up to the developer.

4.3.4 LOCAL PROCESS INTEGRATION

On most modern operating systems, several processes can execute (seemingly) simultaneously. A major purpose of these concurrent processes is to facilitate management of multiple independent control threads, a kind of separation of concerns [Dij76]. A process is here defined as an executing program with a single thread of control and associated data (while for some operating systems, a process may contain several threads of control). Processes typically have address space protection, making it impossible for one process to directly access the address space of concurrently executing processes. To further isolate processes, they normally only communicate with the environment (I/O, the file system, etc.) using system calls provided by the operating system.

As for procedure and object integration, the objectives of process integration (cf. Figure 2-d) are control and data transfer. However, control transfer takes on a new meaning since each process already has a thread of control. The address space protection mechanism makes it impossible for one process to set its program counter to the address space of another process. This restriction is a major benefit of the process concept, but it makes it impossible to transfer control as in the previous examples (e.g. object integration). The substitute of control transfer then becomes synchronization of execution. Since synchronization is performed via data transfer, the issue of control is dependent on data transfer. But also direct data access is prevented by address space protection. Other means of data transfer are thus necessary. Interprocess communication mechanisms are implemented in the platform using a variety of specific technologies, but in two distinct ways: by providing means for processes to share a process-external memory area and by providing some message passing mechanism.

Message passing has been implemented in numerous ways, including pipes, remote procedure calls, remote method invocations, sockets, etc. Message-passing systems may be synchronous, resulting in processes running in lockstep, or asynchronous, using mailboxes or other buffers for temporary storage of messages. As an example of a message passing mechanism, remote procedure calls [Kha95] [Rud96] simulate synchronous control transfer in the sense that one process acts as a server, waiting for a client process to invoke a procedure, and while the procedure is being executed on the server, the client is blocked. Considering the responsibility distribution between developer and platform, remote procedure calls are designed to have much of the same properties as ordinary procedure calls. However, location of communicating parties is a special problem in the inter-process version. These location issues may be aided by platform-provided directory services. For RPC, it is not enough to locate the correct process, but also the correct procedure. The platform may support interface discovery mechanisms to this effect. Another problem that may occur in the multi-process context is that the called process is unavailable. These situations typically need to be managed by the developer, although the platform might support some error control, e.g. time-outs and error notification. Although some message-passing systems do not specify the data representation of the messages at all, for remote procedure calls – like ordinary procedure calls – the developer determines the numbers and types of parameters, but the variable representation is determined by the platform .

Shared memory is typically implemented as a process-external memory buffer accessible for reading and writing of processes by means of some system calls. Normally, shared memory refers to direct access memory, but the properties of e.g. file sharing are similar. The first problem of shared memory is how to notify the involved processes of where it is located. In general, there are three ways to determine the location of the shared memory area: a) there may be only one (as is the case for the typical clipboard file); b) the shared memory area may be determined at design time (as in file-sharing, when the location and name of the shared file is determined by the developer); or c) the location may be determined run-time (as when the shared file is determined by a user or some other third party). In the first case, the platform manages the location issue completely; in the remaining cases, the developer needs to take some responsibility over this issue. Typically, shared memory mechanisms provide variables or pointers to the shared data. These mainly function as references to the data but may also be typed, thereby allowing type checking. Furthermore, processes reading and writing on a shared memory area need to agree on the representation and meaning of the data. To some extent, this may be determined by the platform (e.g. by setting the data types), but often, the developer is responsible for these issues. Yet another important issue is synchronization. Two processes sharing a common resource risk deadlock, starvation and other undesirable situations. Computer science has evolved a niche on these issues including solutions such as the TEST AND SET LOCK instruction, semaphores, monitors, Peterson's solution, and more [Tan87]. Some platforms implement these kinds of techniques for process synchronization. Finally, as with most technologies, errors may occur in shared memory mechanisms (e.g. a process may attempt to open a non-existent file). The error control provided by the platform is typically restricted to error notification. Finally, if several possible communication mechanisms are available, the processes will need to agree on which to use. This is typically the responsibility of the developer.

4.3.5 REMOTE PROCESS INTEGRATION

The term remote process denotes a process located on one network node that may communicate with processes on other nodes. Remote process integration (cf. Figure 2-e) introduces a number of new integration problems as compared to local process integration. Node integration is heavily based on network communication protocols. Distributed object technologies, for instance, may use TCP/IP as underlying communication protocol. The issues managed by these protocols are discussed in the section on oligarchical integration approaches. In this section, we briefly consider some concerns relevant for remote process integration, such as DCE RPC (Distributed Computing Environment's Remote Procedure Call), Java RMI method invocations, COM and CORBA [Kha95] [Cha96] [Smi98] [Pri99] [Mon00] [MSD01] [Obj01] [Rap01]. It may be well worth noting that these remote process integration mechanisms, as a number of other mechanisms of this chapter, are not clearly classifiable in one category or another; they have both monarchical and oligarchical characteristics. For instance, interface definition languages are standards suitable for the oligarchical context. However, because of their similarities to local process integration mechanisms, they are considered in this section.

When two processes on different nodes attempt to communicate, the problems are related to the introduction of the network and its associated devices, and the division of memory into two distinct entities. The division of memory prohibits shared memory[4] (therefore only message passing is considered). The introduction of the network introduces problems related to addressing and data transfer. Remote process communication tends to be unreliable, which introduces a potential for transmission errors.

Communicating processes on a single computer need to be able to find each other, e.g. by a process identifier. If the processes are located on different nodes on a network, the address of the node is also required. A problem of location is related to how the address is obtained by the process initiating the communication. Node addressing (e.g. by use of IP numbers, MAC addresses or other machine identifiers) is a means for managing the network location of processes. However, even with an addressing system, the calling process may not know the address to the called process. In these cases, directory services may be required, matching e.g. a machine name, application name or other reference with an address. COM, CORBA and DCE RPC all provide these directory services. COM and CORBA additionally provide interface exploration facilities for location of specific methods.

Concerning transmission errors, over a network these typically including jumbled messages, duplicated messages, lost messages, etc. Many of these issues are dealt with on the network protocol level and there is much theory available for error detection and correction. Normally, user processes need only be concerned with presumably lost messages, i.e., expected messages or invocations that do not arrive. This, however, becomes especially problematic in synchronous message passing systems such as the traditional remote procedure call, since the receiving process nominally remains blocked until the expected message arrives. The problem is often managed by a platform-supported timeout for the blocked procedure call.

The integration issues managed by monarchical approaches are consolidated in the end of this chapter. In the next section, oligarchical approaches are considered.

4.4 OLIGARCHICAL INTEGRATION APPROACHES

In the oligarchical development situation the integrators are multiple component developers; this is thus the case when two systems with different developers are to be constructed, explicitly considering their future integration. The rules governing the transfer of data and

[4] An abstraction of shared memory can of course be created, cf. file sharing, but this abstraction needs to be based on message passing.

control are then decided upon by negotiation and agreement. When an agreement is in place, the systems can be designed to include the agreed upon integration capabilities. These oligarchical agreements are the integration standards of the software world. This subsection considers those integration issues that are considered by the Open Systems Interconnection (OSI) reference model [Tan81]. Together with the previous and the next section, the most pertinent integration issues will arguably have been addressed.

For component integration, there is an extreme amount of standards, including well-known acronyms such as TCP [Tan89], HTTP [Fie97], CSMA [IEE00a], EDI [Uni02], RS-232 [Tan89], IP [Tan89], UDP [Pos80], FTP [Pos85], X.25 [Tan89], X/Open [All99], ASN.1 [Tan89], SOAP [Jep01], XML [Bra00] to mention but a fraction. In an attempt to abstract from these specific protocols to general responsibilities, the OSI reference model has been proposed. In this section, the OSI is employed for the same purpose, abstraction. Comparing this section with the previous, the OSI layers here replace the platform as abstraction creator and manager of integration issues. The OSI is a layered model, where higher-layer protocols are dependent on the services provided by those below.

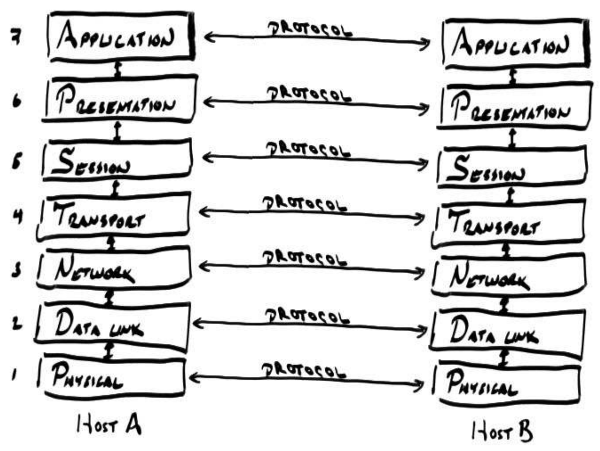

Figure 3. The Open Systems Interconnection reference model.

The seven layers proposed by the OSI are the physical layer, the data link layer, the network layer, the transport layer, the session layer, the presentation layer, and the application layer. One of the critical comments about the OSI model is that few protocols abide by it; for instance, the Internet standard TCP/IP does not conform to it. In the context of this text, this is not important, since the concern is of responsibilities in general rather than of their location in the OSI stack. Another critique against the OSI model is its increasing vagueness with layer number. Especially the application layer (the top layer) is considered to contain everything that the other layers do not address. Attempts to define an eighth layer of the OSI model have therefore been proposed (e.g. [Gla98]). For the purposes of the present work, the application layer thus provides limited support. Furthermore, the physical layer (the bottom layer) is concerned with mechanical and electrical integration rather than software integration. It will therefore not be elaborated on.

4.4.1 THE DATA LINK LAYER

The main task of the *data link layer* is to take the raw transmission facility provided by the physical layer and transform it into a line that appears free of transmission errors to the network layer [Tan89]. Error control is thus a main issue in for this layer. An additional problem managed by the data link layer – specifically by the Media Access Control (MAC) sublayer– is how to allocate a single broadcast channel among competing users, i.e., how to determine who gets to use, for instance, a LAN at a given time [Lit01]. If several network nodes attempt to use a single LAN simultaneously, the messages may garble, resulting in error. There are many protocols available for managing this issue, e.g. CSMA/CD (Carrier Sense Multiple Access with Collision Detection) [IEE00a]. Furthermore, when several nodes share communication medium, the machines must be uniquely identifiable. This is thus a question of addressing.

Between nodes, communication may be connection-oriented or connectionless. Especially in the connection-oriented communication, error control becomes important. One problem that the data link layer needs to manage is the potential bit-level errors that the physical layer may introduce in the communication. By breaking up the data into frames and using different checksums, the data link layer may detect whether bits have been lost or added during transmission. The faulty frames may then be retransmitted. But introducing frames introduces potential frame-related errors. Therefore, techniques for managing lost, garbled and duplicated frames are introduced. Yet another problem encountered on the data link layer is uneven flow of messages. If the sender transmits faster than the receiver can accept, frames will be lost. This is typically managed by introducing some feed-back from the receiver to the sender, thus controlling the flow.

4.4.2 THE NETWORK LAYER

The *network layer* is concerned with controlling the operation of network [Tan89]. As with the data link layer, communication may be connection-oriented or connectionless in the network layer. Although the primary focus is routing, also congestion control and internetworking are relevant to this layer. Routing becomes an issue when a package requires multiple hops from source to destination. Furthermore, each machine over a potentially large network needs a unique identifier. In the well-known IP protocol, this identifier is the nodes IP number. Innumerable routing algorithms have been devised, attempting to provide correctness, simplicity, robustness, stability, fairness and other desirable qualities. One result of bad routing may be network congestion, i.e., that the amount of traffic becomes higher than the network can manage. Congestion control algorithms may be implemented to address this concern. These algorithms function by discarding packets, choking off input, or by similar techniques, when congestion occurs.

Internetworking becomes an issue when communication is required between different subnets, perhaps based on different protocols. Strictly speaking, these issues are relevant for more layers than the network layer. Bridges, gateways, and protocol converters are normally employed to manage the network integration issues that may occur. Bridges are employed to manage connections with differing protocols on the data link layer, gateways are employed when connecting dissimilar network layer networks and protocol converters are used for higher-layer integration. These devices need to manage a number of compatibility issues, including frame and packet reformatting and differing protocol speeds (a kind of flow control). In specific cases, other issues may become problematic. For instance, one protocol may not incorporate information required in the other, e.g., priority or security properties. Therefore, in some cases, integration cannot be complete.

4.4.3 THE TRANSPORT LAYER

The *transport layer* is to a large extent a local abstraction of the network layer; the bottom four OSI layers can be seen as the transport service provider, whereas the upper three layers are the transport service user [Tan89]. Since the network layer hides many of the complexities of the network (such as routing) the transport layer resembles the data link layer, again managing communication between two nodes. Error and flow control therefore resurface in the transport layer. An issue that does not appear in the data link, but in the transport layer, is addressing. How does a process find the address to a server providing the desired service? Typically, directory services are employed for these purposes, i.e., yellow pages, linking address to service reference. The transport layer also provides a process abstraction for host machines. For example, the TCP protocol contains ports in addition to IP addresses, thereby allowing several parallel connections from one host.

The subnet as seen by the transport layer, as opposed to the physical connection as seen by the data link layer, is capable of storing and even duplicating messages. Duplicated messages may result in errors if they are not detected. The transport layer therefore requires facilities for managing this problem. A final issue that is dealt with in the transport layer is crash recovery. A host that crashes is particularly precarious, but if it restarts, there are possibilities of resuming communication where it ended.

4.4.4 THE SESSION LAYER

The *session layer* is considered a thin layer, mainly providing facilities for managing and coordinating dialogs between two application processes [Mod91]. For instance, the standard synchronous remote procedure call prohibits a client from making a second call before the first has been completed. If the remote procedure call concept [Bir84] were to fit anywhere in the OSI model (which it does not), it would be in the session layer or possibly the application layer [Tan89]. Dialog management is thus a matter of sequencing. Another issue supported by the session layer is the resynchronization of a bungled transmission. For instance, when large file transfers are unexpectedly interrupted, the use of synchronization points inserted in the file can be used to avoid retransmitting the entire file. A similar concern is activity management, which is supported by a set of primitives in the session layer. Activities are used to isolate certain communications, e.g. transferred files or other related information. Activities may also be interrupted, thus introducing a kind of task swapping into the communications. Finally, the session layer provides a feature for reporting unexpected errors between communicating parties.

4.4.5 THE PRESENTATION LAYER

The *presentation layer* is concerned with the representation of the data that is to be exchanged between two communicating processes [Tan89]. It is common that communicating peers do not employ the same data representation for the same concept. Examples of differing representations range from low-level stuff such as EBCDIC vs. ASCII character representation and big and little endian byte numbering, to representations of complex structures such as maps or CAD drawings. Because of these incompatibilities, data transformations are necessary. A popular approach for managing transformations is by using a canonical form for describing data structures. Instead of devising a transformation between every two formats, it suffices to devise a single transformation to the canonical form for every format. The drawback of this approach is that all actors will need to agree on which format to use. Another popular, and less ambitious, approach is to not only send the data in some implicit format, but to attach the data structure description to the data. The receiver then has a fair chance of correctly interpreting the data.

Using a common standard for data structure specification can be characterized as an oligarchical approach to an anarchical problem, because now the format of the data structure description needs to be agreed upon. Everyone agrees to disagree on actual format, and instead agrees to agree on how to characterize the disagreements. However, since even this agreement has been difficult to achieve, several standards have been proposed. Two such standards are Abstract Syntax Notation 1 (ASN.1) [Bar92], specified as a part of the OSI development work, and the more recent eXtensible Markup Language (XML) [Mor01]. Briefly, ASN.1 is designed for efficient communication, while XML is designed primarily for readability. Mappings between the two standards have been proposed [Ima00]. It may be well worth noting that even when two actors settle for on e.g. XML, further mutual understanding is necessary for a correct interpretation of the meaning of the communicated data.

Two other issues related to data representation, namely data compression and encryption, fall into the domain of the presentation layer. There are innumerable technologies for these issues, but in this text, we restrict ourselves to noting their inclusion in the presentation layer.

4.4.6 THE APPLICATION LAYER

The *application layer* holds the application processes [Tan89]. The OSI has standardized a number of common applications, such as e-mail, virtual terminals, file transfer, file access, directory services, transaction management, and more. Application layer protocols include DNS, Finger, FTP, HTTP, IMAP, POP, SMTP, SNMP, SSL, TraceRoute, and WhoIs. Because of the diversity of applications, the application layer is broad and heterogeneous. Here, only a few particularly interesting application layer issues are considered. A first observation is that although the shift is subtle, the protocol specifications cover much of the component functionality in this layer, as compared to the lower layers, which mainly consider the components interface. A second relevant observation is that there are some applications in the layer that explicitly address integration concerns, e.g., directory services and CCR (Commitment, Concurrency, and Recovery) [Hen92]. Directory services may allow the run-time discovery of components to interact with [MGr00] [Ric00], e.g., the Domain Name System (DNS) [Sha01]. CCR primarily implements the two-phase-commit protocol (cf. transactional middleware in Section 4.5.4), thus ensuring atomicity of transactions.

It is interesting to note some seemingly irrational deviations from the layering of the OSI model. As such, the Simple Object Access Protocol, SOAP [Jep01], is a prime example. SOAP primarily provides remote method invocation functionality. As such, it is (at least to some extent) located in the session layer. However, SOAP is standardized on top of XML (if anything, a presentation layer standard), which is embedded in HTTP, an application layer protocol. The HTTP probably runs over TCP/IP. In the SOAP example, then, the layering order seems unnecessarily complex. The reason for parts of this design is that the HTTP protocol typically is permitted to pass through firewalls, while RPC calls are blocked. By embedding RPC in HTTP, the firewalls are thus overcome. Instead of removing or reconfiguring the firewalls, new protocols are introduced. In a sense, the evolution is thus similar to an armaments race: the security people might respond by introducing more firewalls, and the people interested in functionality retaliate by devising new, even more complicated protocol stacks, able to penetrate the new firewalls.

The issues managed by the oligarchical integration approaches are compiled with the mon- and anarchical approaches at the end of this chapter.

4.5 ANARCHICAL INTEGRATION APPROACHES

In a competitive and rapidly evolving marketplace, standards are rarely generally agreed upon. It is sometimes good business to develop a competing standard rather than adhering to an existing, and once a technology has been standardized, there is generally room for improvement by deviation. Oftentimes, there is no standard at all. It is thus not uncommon for user organizations to attempt the integration of software systems that were built with no, or few, common interaction assumptions. This is the anarchical development situation, where every man is for himself. The component developers have not agreed on anything and the integrator must accept the components as they are, with the integration facilities that they happen to provide. Furthermore, the components have typically been considered to be independent systems in their own right before the idea of their integration was introduced. Typical components are therefore accounting systems, geographical information systems, process control systems, production planning systems, etc. This is the context of Enterprise Application Integration (EAI) field [Gol99] [Lin00] [Mor01] [McG00] [Ruh01].

EAI may be divided into three main issues. The first issue concerns access to data and functionality in the different components. The second issue deals with the actual interconnection needs to be managed, including data transformation, message routing, etc. Finally, the properties of the resulting "system of systems" need to be managed. There are several system-level properties that often become problematic, e.g. security and data consistency.

4.5.1 ACCESS

Many business applications are designed in a three-layered architectural style, with a database on the lowest layer, business logic on the middle layer, and a user interface on the top layer. The rationality of this division is similar to that of the object-oriented paradigm, dividing objects into attributes (data), methods (logic), and interface. When attempting to access the data or functionality of a system, there is often a choice of which layer to integrate to.

Oftentimes, business systems have a separately accessible database – perhaps developed by a third party. Database-oriented integration allows access to data but not functionality. Many software systems provide interfaces to allow invocation of functionality. The types of interfaces (and of course the functionality) offered vary significantly. The main concern of application-oriented integration is to discover and employ the syntax and semantics of these interfaces. Some systems offer only one interface to data or functionality, namely the user interface. These are typically legacy systems constructed without any consideration of future integration. User-interface integration is concerned with exploiting this singular interface.

In the fortunate case, a system acting as a server has an interface abiding by the exact rules that a client system expects; this is the monarchical and oligarchical scenario. In the unfortunate (and unfortunately common) case, the interface of the server system does not match the expectations of the client system. This is the case normally considered by the EAI literature. The typical approach for management of these mismatches is to employ wrappers, adapters, connectors, gateways, bridges, etc.

4.5.2 INTERCONNECTION

Even though the interfaces may be accessible, some problems remain for a successful integration of software systems. Firstly, a message from a data provider needs to have a data consumer, and it is not always the case that the provider knows the address of the consumer. Secondly, the request must be transported from the provider to the consumer. Thirdly, the data representation of the message may not match that expected by the consumer (or even of the consumer's adapter). Adapters sometimes manage some of these issues, but an underlying infrastructure is required and furthermore, it is becoming more common that specialized integration mediators handle some of the traditional adapter concerns.

4.5.3 EXTRA-FUNCTIONAL PROPERTIES

The success of an integration project is not solely measured by the success of access and interconnection between components. There are several situations that may arise when the components function perfectly in isolation and when access and interconnection have been duly managed, but the resulting system is unsatisfactory. The system-wide qualities that appear in the cooperation of the components are sometimes called extra-functional properties.

There are a number of extra-functional properties that tend to be particularly problematic in EAI projects. Data inconsistency is one such issue, to a large extent related to the assumption of many components that they are alone in manipulating the data in their database and that this data is only stored in their database. When this assumption becomes untrue, inconsistencies may appear. Another common issue is security, since integration often implies opening new ways into a component. Yet another issue is performance. An example of this is when many components are to employ one database, originally designed for only one or a few connections. The list of extra-functional problems goes on, including reliability, atomicity, durability, and more.

4.5.4 EAI TECHNOLOGIES

In this section, a number of common EAI technologies for the problems described above are presented. These technologies include adapters, data-oriented middleware, transactional middleware, message-based middleware and message brokers. Finally, the popular concept of web services is considered.

ADAPTERS

The most obvious issue in software integration is the need to somehow access the data and functionality that is contained within an application. The standard way of doing this is by employing interfaces to the application. There are two main interface types to an application, the application interface and the user interface. Application interfaces come in several forms, implemented as language-specific import libraries to be linked into the calling application's code, as component framework interfaces (such as CORBA interfaces), etc. They may offer complete access to functionality and data of the application, or only to a restricted subset. Some applications have no application interface at all.

Adapters are employed when an interface needs to be transformed into another form. For instance, if a component provides an interface based on the CORBA standard is to be integrated into a system of COM-based components, an adapter may be employed. There is no general agreement of what, exactly, is the difference between adapters, connectors and wrappers (sometimes called bridges and gateways), so this section will treat them as belonging to the same group (adapters) [Sne96] [Luc97] [Sne97] [Cim98] [Sne98] [Gan00] [Ber00] [Chi00]. The main purpose of adapters is to present the interface that the client system expects on the one end, and convert this into manipulations that the server system interface expects.

The most rudimentary adapters simply translate between data representations of, for instance, procedure calls, renaming procedures, reordering parameters, etc. More complicated adapters may additionally translate data formats on other levels (e.g. between big-endian and little-endian format or database record structures). Adapters may also be sensitive to certain calls from the application, and they may listen for, and trigger on, application events. In some cases, the adapter may need to wait for several external systems to provide sufficient parameters for a call to the application, and vice versa, one call from an external system may require several calls from the adapter to the application. In the complicated cases, the adapter may need to perform several invocations, queue messages, retain its state, manage security, error control, monitor and log events, manage timing, and so on [But99]. Furthermore, sophisticated adapters may dynamically discover certain aspects about the application they are connecting to, such as database schema. Dellarocas thesis [Del96] contains an in-depth study of the construction of adapters.

If an application does not provide any application interface, screen scraper adapters may be employed to interpret and manipulate the user interface in accordance with functions of the adapters application interface, thereby providing software access to the application. A final (generally unpopular) option is to modify the source code of the application to allow interfacing.

DATABASE-ORIENTED MIDDLEWARE

Database-oriented middleware, including database federation software, database gateways, virtual databases, call-level interfaces, and database replication software, are employed to simplify data access from heterogeneous data sources. There are two typical problems that these products address. Firstly, an application may encounter difficulties when requiring data access from more than one database, since the database developers oftentimes use native data formats and application interfaces. From the point of view of the client application, the best thing would be if it could perceive the multiple heterogeneous databases as a single homogeneous one. By introducing a software layer between the application and the databases, this illusion can be created. A similar problem is when two databases for some reason need to contain the same data. This may be the case when two unmodifiable systems with useful functionality require the same data, but are designed to only access the data in its own database. A mediating component may then be introduced, keeping the data consistent between the systems.

To manage these functions, database-oriented middleware typically performs two main functions, connection and translation. Connection to native databases is performed with database-specific adapters, in this context usually denoted database drivers. Adapters are discussed above. Translation denotes the transformation of the format of the actual database data. Depending on the situation, the transformation may be between idiosyncratic representations, or to a third format suitable for external access. The compilation of several databases to one virtual database is also mainly a representational concern.

TRANSACTIONAL MIDDLEWARE

Transactional middleware products are not single-problem solutions, but rather wide-ranging concepts. The main issue approached by these products is coordination between several resources in performing restricted tasks called transactions. A basic characteristic of a transaction is its "all or nothing" character; either the transaction is completed or not, but never left in an undetermined state. The reservation of a theatre ticket, for instance, often includes the preliminary booking of a seat while the transaction is carried through. If the transaction is terminated before completion, the preliminary booking is cleared. This property of a transaction is called *atomicity*. Additional transaction requirements include *consistency* (e.g., two databases never end up with inconsistent data), *isolation* (e.g., two customers never book the same seat) and *durability* (e.g., once committed, the ticket reservation survives system failures). Jointly, these properties are referred to as the ACID properties.

Furthermore, transactional middleware is often responsible for additional features, such as increasing scalability by e.g. using load balancing (dynamic processing load distribution) and database connection pooling (multiplexing transactions over database connections), security and fault management.

The X/Open Distributed Transaction Processing standard [All99] defines three main components for transaction processing: applications, transaction managers, and resource managers. Briefly, application requests are accepted by the transaction manager, which invokes a number of resource managers, which in turn commit data to databases or perform some functionality. Transaction processing monitors were for a long time the prime transaction managers, but recently, Java- and Web-enabled application servers are challenging their hegemony, providing a location for application logic [Lin01b].

Transactional middleware thus potentially manages all kinds of integration-related issues. The resource managers function as adapters, connecting to the databases or other resources, both transforming APIs and translating data. Location of communicating parties and routing is performed by the transaction manager as well as error control related to the ACID properties (e.g. by two-phase commit and dynamic switching). The transaction manager also determines how messages and invocations should be ordered and under what circumstances they should be directed to one party rather than another. Additional middleware is however normally employed for data transport.

MESSAGE-ORIENTED MIDDLEWARE AND MESSAGE BROKERS

Message-oriented middleware denotes middleware solutions based on the dispatch and reception of messages between applications [Lin00]. From a functional point of view, it does not differ significantly from message-based, monarchical, operating-system-provided, inter-process communication, simply passing messages between processes, possibly utilizing queues. In the anarchical context, however, the main benefit of message-oriented middleware is the platform-independence that these solutions often bring about. Two common models for messaging is point-to-point, where each component sends messages directly to its communication parties, and publish-subscribe, where a component may publish messages on a centrally located hub and other components may retrieve the messages by subscribing to a certain topic. Further kinds of messaging options are queued and direct messaging. In the queued model, the recipient does not necessarily need to take care of the message immediately, while this is required in the direct version. Platform independence is in no way inherent in the message-oriented concept; it is instead the result of an explicit effort by the developers of these systems. The most well-known messaging middleware is probably IBM's MQSeries (recently renamed WebSphere MQ) [Tho99], with support for an impressive number of platforms, including IBM OS/390, Pyramid, Open VMS, Unix, Solaris, OS/2, Windows NT, MacOS and more.

On top of message-oriented middleware (and sometimes on top of other middleware) a message broker may be located. The message broker communicates with a number of applications using application-specific adapters that accept messages from the broker, feed them to the application, collect outgoing data from the application and submit them to the broker. The message broker consequently functions as a central hub, receiving and dispatching messages from a number of applications. When a message arrives from one application, the broker determines where it should be passed, transforms it into a format suitable for the receiving application, and forwards it. Successful use of message brokers in heterogeneous environments is heavily dependent on application adapters. Message broker vendors therefore provide a wide range of adapters to common systems as well as adapter development environments, to be used for new applications. Typically, message broker adapters are noninvasive.

A second key function of message brokers is message transformation. The preferred solution uses a canonical form to which all message formats have a mapping. From the canonical form, any message can be generated as long as its data has the expected semantics. Conversion applies to the schema level as well as the bit-level of data representation. In a sense, the conversion rules of a message broker contain much of the semantics of the data in an enterprise system, since each data item is presented in many forms, each indicating some aspect of the item.

The third function of message brokers is called intelligent routing. A message entering the message broker from an application is parsed, decomposed, and interpreted. Based on the interpretation, i.e. the contents of the message, it is transformed into some other representation, possibly combined with some other message, and finally dispatched to recipients according to the message brokers internal logic. The forwarding of messages based on their contents is called intelligent, or content-based, routing. As the message transformation, intelligent routing indicates some form of understanding rather than mindless, indiscriminate processing.

Many of the variable functions performed by message brokers are based on rules. These rules, specifying when a message should be forwarded to application A or application B, to what format it should be converted, with what other data it should be merged, etc., describe much of an organization's processes. Therefore, yet another layer of functionality has been added in the enterprise application integration world, the process automation layer. Process automation tools may be employed to visualize and control how information flows through the organization's systems.

WEB SERVICES

The fairly new concept of web services [Che01] [Sne01] addresses anarchical integration from an oligarchical viewpoint. It is recognized that many existing software components that could generate value if integrated were built without agreements on interoperability standards. The solution to this problem is, according to the web services concept, to standardize the descriptions of the components. Briefly, via a central registry (called a Universal Description Discovery and Integration, UDDI, registry), standardized information (authored in the Web Service Description Language, WSDL) describing software components (called services) is located. When in need of a particular service, autonomous software components may search the registry for the relevant service, find sufficient information on offered functionality and interfaces, accepted data formats, protocols, etc., to be able to use the service [Mou01].

Marketed as the ultimate solution to software integration over the Internet, web services do however disappoint in the perhaps most crucial issue: the semantics of the services. The descriptions available via the UDDI registry contain machine-readable information on method signatures, protocols etc., but no information on the actual functions performed by the service. These service descriptions are instead available in natural language. Therefore, the most probable readers of the UDDI registry and the WDSL descriptions are software developers rather than software, and seamless and automatic integration of components is still a thing for the future.

4.6 CONSOLIDATING THE APPROACHES

In this section the integration issues managed by the three approaches described above are consolidated. This consolidation serves as a foundation for the concluding evaluation of architectural analysis approaches of Chapter 6. Table 1 to Table 3 summarize the above technology reviews from the perspective of the below presented categorization. For each technology and integration issue, the tables indicate whether the platform/technology manages the issue and to what extent.

Data representation. Data representation refers to syntactic issues. In the context of integration, this particularly concerns data translation; if the data representation issue is not managed, communicated data will not be readable by the recipient [Kim91] [Bri92] [Här99]. Furthermore, data representation is typically an issue on several layers of abstraction. In the remote procedure call, for instance, the developer is responsible for following the syntax as prescribed by the platform. Within the limits of the platform, the developer is free to select procedure and parameter names. With the freedom comes the responsibility of ensuring that procedure definitions and procedure calls match. In the underlying communication supporting the remote procedure call, network protocols require correct representation of packet headers and such.

In the OSI stack, all layers are to some extent concerned with data representation, since packet headers sent between peers need to be understood in terms of addresses and the like, but layer 6 is responsible for the data representation of the actual content of a message, managing issues such as translation, compression and encryption. Application integration adapters often manage issues of data representation, wrapping one interface with another, translating parameters and messages. A prime example of a data representation technology is data-oriented middleware, which is employed explicitly for the translation of large amounts of data.

Data semantics. Data semantics refers to the meaning of data. If the data semantics issue is not managed, the data may be readable, but it will be interpreted incorrectly. For instance, one component may send the integer 10 to another component. The first component may have measured 10 as the temperature in degrees Celsius, while the second component interprets it as the number of books to buy. It is often the responsibility of the developers to make sure that the meaning of data is preserved, but also the platform prescribes meaning to data. For instance, the integer in the example above is partially defined by arithmetic operations defined by the platform. Examples of management of data semantics also include message broker's intelligent routing: the message broker interprets the data of a message and decides on its destination based on its contents. Furthermore, an XML message includes a tag for each data item, indicating its meaning. A weakness of XML is of course that it is necessary for the components to agree on the meaning of the tag instead of the data per se. A categorization of increasing maturity levels of semantics capabilities in communication is presented by Ericsson and Schubert in [Eri96].

Connector semantics. Connector semantics refers to the behavior of connectors, i.e., synchronization and sequencing of interactions between components [Bar95b] [Hua98] [Jma00]. Typical results of failure to manage connector semantics include deadlocks, starvation, etc. Connector semantics are, as data representation issues, defined on several layers. For instance, in a remote procedure call, the platform ensures that the server responds to the client. The remote call may, however, on a lower level be communicated with TCP/IP, which in itself contains and manages a number of synchronization issues between peers.

On top of the remote procedure call, the developers may have implemented additional sequencing or synchronization rules.

Component semantics. The term component semantics refers to the behavior of components [Moo97]. If the component semantics issue is not managed, a component invoked by another will not behave as expected by the invoker, even though the invocation was syntactically impeccable. In the reviewed technologies, the developer is primarily responsible for component semantics. To a minor extent, dynamic adapters may probe the components to which they are attached to explore certain aspects of how they work. Web service descriptions also manage component semantics to some extent, since they describe the functions of the component. However, these descriptions are primarily of an intuitive nature, and require human interpretation. The application layer of the OSI stack, does however, specify a considerable portion of the component semantics for the applications considered by the ISO. Generally, and particularly for the lower OSI layers, the distinction between component and connector semantics is vague, since the component is restricted by the connector semantics. Arguably, for the lower layers, however, only a small portion of the desirable functionality of a component is specified by the connector semantics.

Error control. Error control refers to the management of undesired behavior. If error control is not implemented, everything will work fine under optimal circumstances, but once a disturbance is introduced, the system execution is in danger. For instance, layer 2 of the OSI stack manages bit-level transmission errors that may be caused at the physical layer. Layer 3 in the OSI stack manages network congestions and reroutes traffic when network nodes are lost. Transactional middleware typically implements two-phase commit protocols, ensuring multi-component synchronization by a sequence of assurances between the components. Much of error control is also managed by the developers.

Location. Location refers to the identification, location, addressing of and routing to of communicating parties. If the location issue is not managed, a message sent by one party might reach some recipient, but not the intended. Layer 2 and 3 of the OSI stack are concerned with addressing and routing. For remote procedure calls, the platform may provide a directory service mapping procedure names to network addresses. It is, however, normally the developers' responsibility to locate the service reference (e.g. the node, process or procedure name). Message brokers typically route messages to the appropriate destination based on a set of criteria, e.g. the source and the message contents.

Extra-functional properties. Extra-functional properties, or quality attributes, refer to an array of "ilities" that often need explicit consideration in software integration projects. These include security, data consistency, performance, modifiability, reliability, and more. Extra-functional properties are in the practical case often tightly linked to functionality. For instance, reliability is enhanced with mechanisms for error control and performance is increased with load balancing and connection pooling. Moreover, they are only partly related to the actual integration solution, since they are heavily dependent on the nature of the components per se. Furthermore, there is no end to the number of extra-functional properties that could be considered. It is thus difficult to clearly delimit the specific properties to take into consideration here. The set considered in this text is determined by the individual properties prominence in literature. For instance, those technologies that explicitly address error control, also address reliability. Furthermore, the OSI presentation layer explicitly considers security by means of encryption, and performance by means of compression. Transactional middleware featuring two-phase-commit explicitly address atomicity.

The integration issues presented above may be compared to similar classifications, such as [But99], [DeL99] and [Yak99b]. Although the issues described above all are important for software integration, they are in many respects of different types. For instance, the five first issues are related to what components expect of one another. A component expects data on a certain form and it expects the data to mean something, it expects certain communication sequences, it expects collaborating components to behave in particular ways, and it may expect certain kinds of disturbances. Extra-functional properties are in this context a horse of a different color, expressing characteristics of the system as a whole. Also, there is overlap between certain issues. For instance, the extra-functional property reliability and the error control issue are closely related. These overlaps are to be expected due to the vagueness of extra-functional properties. Despite these overlaps, the issues considered are – and need to be – managed when integrating software components. To the author's knowledge, there is no more appropriate classification of integration concerns.

4.7 SUMMARY

This chapter has reviewed and consolidated three different approaches to software integration. The primary reason for the review was to identify the issues that different integration mechanisms tackle. These issues are, arguably, those that generally need to be addressed when integrating software systems. A compilation of integration issues thus concludes the chapter.

Software integration is monarchical when the same actor (be it a single developer, a software developing organization, or a user organization) develops the components to be integrated as well as performs the actual integration. Examples of monarchical integration mechanisms include procedure integration, object integration, and process integration. Literature on operating systems, compilers and programming languages dominate in the monarchical approach.

Software integration is oligarchical when the component developers agree on integration mechanism before developing the components. Examples of oligarchical integration mechanisms include TCP/IP, XML, FTP, and many other similar standards. In this chapter, these agreements between component developers were considered within the framework of the OSI reference model.

Software integration is anarchical when there are several component developers, but they have not agreed on integration mechanism. The software integrator is then forced to resort to unstandardized mechanisms for accomplishing the task. Examples of anarchical integration mechanism include adapters, virtual databases, application servers, message brokers, and more. Literature on anarchical integration approaches is typically found in the enterprise application integration (EAI) field.

The three integration approaches are all concerned with a number of problems of integration. Communicated or shared data needs to be presented in a form that is readable as well as understandable to all components; communication must take place in a manner that is expected by the components; the components must have correct expectations on each others behavior; certain potential errors need to be managed; components need to be able to find each other; and the collaborating components should display certain system-wide properties. These issues, data representation, data semantics, connector semantics, component semantics, error control, location, and extra-functional properties, are considered by the reviewed integration mechanisms. In Chapter 6, deduction-based approaches to architectural analysis of software integration are evaluated on the basis of the same integration issues.

	Monarchical integration				
	Process events	**Procedures**	**Objects**	**Local Processes**	**Remote processes**
Data representation	Platform determines syntax of statements.	Platform determines syntax of (general) procedure call and partially determines syntax of variables.	Platform determines syntax of object references. Platform determines syntax of (general) method invocation and partially determines syntax of variables.	RPC: Platform determines syntax of (general) procedure call and partially determines syntax of variables. Shared memory: Platform partially determines syntax of variables.	RPC: Platform determines syntax of (general) procedure call and partially determines syntax of variables. Shared memory: Platform partially determines syntax of variables.
Data semantics	None.	Platform prescribes some meaning to variables (integer, character) by statements operating on variables.	Platform prescribes some meaning to variables (integer, character) by statements operating on variables.	Platform prescribes some meaning to variables (integer, character) by statements operating on variables.	Platform prescribes some meaning to variables (integer, character) by statements operating on variables.
Connector semantics	Platform determines sequence of statement execution.	Platform determines call-return model.	Platform determines creation-usage-destruction sequence. Platform determines call-return model.	RPC: Platform determines call-return model. Shared memory: Platform determines creation-usage-destruction sequence. Platform may be responsible for synchronization.	RPC: Platform determines call-return model. Shared memory: Platform determines creation-usage-destruction sequence. Platform may be responsible for synchronization.
Component semantics	Platform defines meaning of statements.	Platform defines meaning of statements. Connector semantics.	Platform defines meaning of statements. Connector semantics.	Platform defines meaning of statements. Connector semantics.	Platform defines meaning of statements. Connector semantics.
Error control	The management of unexpected situations is typically shared between platform and developer.	The management of unexpected situations is typically shared between platform and developer.	The management of unexpected situations is typically shared between platform and developer.	Unavailable communicating RPC-party, shared-memory-synchronization partially managed by platform (error notification), partly by developer.	Transmission error partially managed by platform (correction or error notification, e.g. on time-out), partly by developer.
Location	Platform prescribes data addressing conventions.	Platform provides reference service for procedures as well as variables (procedure-name-to-address; variable-to-address).	Platform provides directory service for object references (object-reference-to-address). Platform provides directory service for procedures as well as variables (procedure-name-to-address; variable-to-address).	Platform may provide directory service for process references as well as interface discovery support. Shared memory: Platform may be responsible or prescribe conventions for addressing. Variables may be employed.	Platform prescribes node (and port) addressing system. Platform may provide directory service for node and port references as well as interface discovery support.

Table 1. Integration issues managed by monarchical approaches

	Oligarchical integration					
	OSI 2	**OSI 3**	**OSI 4**	**OSI 5**	**OSI 6**	**OSI 7**
Data representa-tion	Communication primitives prescribed by layer.	Communication primitives prescribed by layer. Gateways reformat packets between subnets.	Communication primitives prescribed by layer.	Communication primitives prescribed by layer.	Communication primitives prescribed by layer. Layer 6 is explicitly concerned with data representation (including compres-sion and encryp-tion).	Communication primitives prescribed by layer.
Data semantics	Layer prescribes semantics to communication primitives.	Layer prescribes semantics to communication primitives.	Layer prescribes semantics to communication primitives.	Layer prescribes semantics to communication primitives.	Layer prescribes semantics to communication primitives. Limited semantics via data representa-tion.	Layer prescribes semantics to communication primitives.
Connector semantics	Layer partially prescribes connector semantics.	Layer partially prescribes connector semantics.	Layer partially prescribes connector semantics.	Layer partially prescribes connector semantics. Specifically, layer 5 concerned with dialog management and “activities” for task distinction and swapping.	Layer partially prescribes connec-tor semantics.	Layer partially prescribes connector semantics.
Component semantics	Only by connector semantics.	Only by connector semantics.	Only by connector semantics.	Only by connector semantics.	Only by connector semantics.	To a considerable degree.
Error control	Layer 2 manages bit-level as well as frame-level transmission errors. Flow control.	Congestion control. Bridges and gateways implement flow control between subnets.	Flow control. Manages duplicate message errors. Crash recovery (network or hosts).	Synchronization points for resynching bungled transmis-sion. Error-reporting facilities.	No explicit provi-sions.	CCR-protocol controls consis-tency-related errors
Location	MAC sublayer manages LAN node addressing.	Layer 3 manages network address-ing (e.g. IP-numbering) and routing.	Directory services may be provided. Port addresses are typically provided.	None.	None.	Directory services (e.g. DNS).
Extra-functional properties	Reliability explicitly addressed by error control.	Reliability explicitly addressed by error control.	Reliability explicitly addressed by error control.	Reliability explicitly addressed by error control.	Performance explicitly addressed by compression. Security explicitly addressed by encryption.	Atomicity and reliability in CCR.

Table 2. integration issues managed by oligarchical approaches.

	Anarchical integration				
	Adapters	**Database-oriented middleware**	**Transactional middleware**	**Message-oriented middleware**	**Web services**
Data representation	Renaming of procedures, reordering of parameters, translation of database records, manipulation of database schemas.	Translation of database schemas, records, additional representation. Cf. Adapters.	Resource managers function as adapters.	Translation between representations. Cf. Adapters.	Describes invocation syntax.
Data semantics	Translation between representations.	Translation between representations. Cf. Adapters.	Translation between representations. Cf. Adapters.	Translation rules for many representations. Intelligent routing. Cf. Adapters.	By tagged variables.
Connector semantics	Follows source and destination prescriptions. Application event triggers, call synchronization and pooling.	Cf. Adapters.	Two-phase commit, Cf. Adapters.	Intelligent routing. Cf. Adapters.	Specifies protocols.
Component semantics	Dynamic component discovery.	No.	Only by connector semantics.	Intelligent routing.	In natural language.
Error control	No generic, in some cases.	No generic.	Two-phase commit, additional fault-tolerance (rerouting / dynamic switching, redundancy). Task distribution.	No generic.	None.
Location	No.	No.	Task distribution / routing. Load balancing.	Intelligent routing.	The UDDI registry.
Extra-functional properties	No generic. Sometimes security.	Data consistency	Performance, Security, Reliability, Atomicity, Consistency, Isolation, Durability		

Table 3. Integration issues managed by anarchical integration approaches.

Chapter 5

Software Architecture

5.1 BRIEF INTRODUCTION TO SOFTWARE ARCHITECTURE

Software architecture is the name of a particular form of abstraction, or model, of software systems. In a general sense, models of software systems are, of course, not new; these kinds of models have existed as long as software has. To distinguish software architecture from other software abstractions, a number of specific characteristics have been proposed: software architecture is concerned with a *higher-level abstraction*; software architecture is related to *more complex systems*; software architectures consist of *components and connectors*; software architecture describes the *structure*, or *topology*, of a software system; software architecture is particularly concerned with the *external properties of components* and their *relations*; software architecture is located in the *early design phases*.

However, before the concept of software architecture was conceived, module interconnection languages (MILs) [Ric94], interface definition languages (IDLs) [Pri99], object-oriented modeling languages [Boo99], hardware description languages such as VHDL[5] [IEE00b] [Sha86], Ada packages [Dia93], Modula-3 modules [Kin93], structured analysis notations and techniques [Huß97], process interaction notations such as Hoare's Communicating Sequential Processes (CSP) [Hoa85], statecharts [Dav93], and many other techniques, concepts and notations had been developed for much the same purposes as software architecture. Furthermore, industry has for a long time been bubbling with idiosyncratic techniques for describing software systems. In this context, software architecture as a concept seems to provide little that was not already in existence.

In fact, there is nothing really new with software architecture. Software architecture is simply a new name for the same old thing: to on a conceptual level try to understand the essential aspects of a software system. It is therefore not always meaningful to attempt to

[5] Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

separate software architecture from other design paradigms or notations. High-level, conceptual or early design has simply been renamed software architecture.

What is important with the emergence of software architecture as a distinct academic field is the problem area as such: how software systems can be represented and understood. Software architecture can be viewed as an attempt to refocus on this question and to gather those research groups that have been considering these issues in their own context.

5.2 Definitions of software architecture

Although software architecture as a term may be found in texts from the seventies [Bro75] and eighties [San81] [Bjö82], the first generally cited attempts to legitimize software architecture as an academic discipline, were authored by David Garlan, Mary Shaw, Dewayne Perry and Alexander Wolf [Sha89] [Per92].

There seems to be some agreement what software architecture is and is not, yet it is not a term well defined [Bar98]. From the top-level, one-liner definition down to the specifics of component interface structures, there are alternative definitions and representations. This is not necessarily all bad, since it allows for many uses of software architecture, but it does allow for a whole lot of confusion. This whole chapter is therefore devoted to presenting the meaning of software architecture as used in this thesis. In particular, this section considers common definitions of software architecture as well as the definitions employed herein.

5.2.1 Classical definitions of software architecture

Some definitions of software architecture have become more cited than others. Below, the four arguably most cited one-liners are considered.

According Garlan and Shaw, at the Computer Science Department at Carnegie Mellon University (CMU) [Sha96a],

software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Bass, Clements and Kazman, at the Software Engineering Institute (SEI) [Bas98] propose that

the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Perry and Wolf, at the AT&T Bell Laboratories [Per92] suggest

a model of software architecture that consists of three components: elements, form, and rationale.

According to Gacek, Abd-Allah, Clark, and Boehm, at the Center of Software Engineering at the University of Southern California (USC) [Gac95],

a software system architecture comprises:

- *A collection of software and system components, connections, and constraints.*
- *A collection of system stakeholders' need statements.*
- *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.*

Although these definitions, under fortunate circumstances, might have referred the same concept, they do not. Software architecture is among other things concerned with the structure/form/composition of components/entities. General consensus, unfortunately, seems to end with this diplomatic (and permutable) statement. It is not generally agreed upon what a component or entity is, it is not generally agreed upon what a structure is, or even if it is to be called structure, and it is not generally agreed upon what else comprises software architecture. There is also a debate on whether software architecture is restricted to project-early, high-level abstractions [Sha89] or not [Rie99].

Perhaps the clearest division in the world of software architecture is between formalists and practitioners. Also software engineering in general displays this division between hard [Ala98] [Boy99] [Cla96] [Hut00] [Par83] [Rey98] [Sch86] [Spi89] and soft [Foi85] [Gam98] [Jac99] [Roy70] [Wal01] approaches. The formalistic approach, primarily represented by the architectural description language community (e.g. [Mor97b] and [Luc95a]), is based on precise specifications of architectures and deduction-based reasoning applied to these specifications. The formalistic approach is based on a number of fundamental assumptions, such as correct refinement [Mor95] between related specifications. When these assumptions are true, the formalistic approach is often able to produce convincing results. When the assumptions are questionable, the practitioners (e.g. [Bas98]) enter the arena, confronting the uncertainties of the real world, generally with less conclusive results.

The above distribution of definitions already on the one-liner stage leaves the discipline of software architecture in an awkward position. In many senses it might therefore have been more informative to speak of individual authors rather than of software architecture in general. Surprisingly and fortunately, however, it seems that the definition debate of software architecture in some senses highlights the differences rather than the similarities of the area. Most involved parties accept that at least a few precisely defined architectural description languages indeed belong to the discipline; for instance, overviews of software architecture such as [Ves93], [Cle96], [Med97], [Gar98], [Med00] and [Abd96] all agree on MetaH [Ves98] and Rapide [Luc95a] as architecture description languages. Both of these languages define software architecture precisely. There are more common grounds within the software architecture discipline, for instance, architectural styles are generally considered to belong to the core concerns. Moreover, a number of shared assumptions permeate the architectural community. There is a common belief that a description of the relations between the entities of a software system is beneficial. It is furthermore generally believed that an architectural description can be used as a base for reasoning about certain properties of the depicted system; the architecture is considered closely related to requirements, and in particular to extra-functional, emergent or quality requirements. Considering the authors of the above definitions, work at the CMU on the architecture description language Wright involves analysis of run-time properties of software architectures [All97]. As the name indicates, the Architecture Trade-off Analysis Method (ATAM), developed by SEI, is concerned with architecture-based analysis of multiple quality attributes [Kaz98]. The USC has analyzed software architectures for architectural mismatches [Abd96] [Gac98]. In [Hei01a], Wolf highlights the use of architectures for analysis of extra-functional attributes.

Thus, although the exact nature of software architecture is debated, there is a fairly general consensus that an architectural description can be used as a base for reasoning about certain properties of the underlying system. Furthermore, there is agreement on some examples of precise architectural formalisms.

5.2.2 EMPLOYED DEFINITIONS

This section, and much of this chapter, elaborates on what the term software architecture denotes in the presented research. This thesis is mainly based on the definition of software architecture found in [Sha96a]:

Software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.

Considering this definition, we find a number of key terms that require further elaboration. The main elements referred to are, according to [Sha96a], *components*, while the interactions are modeled with *connectors*. Patterns of composition and pattern constraints are mainly related to *architectural styles*. These concepts are considered below. Furthermore, a number of issues are ignored by this definition, including stakeholders' needs and design rationale. These are, arguably, important and closely related concerns, but are in this text not treated as part of the definition of software architecture.

When deviations from the definitions of this section are made, this will be pointed out. In particular, the term *enterprise software architecture*, differing in some significant ways from software architecture, will be elaborated on below. Furthermore, when reviewing architectural approaches proposed by others, their respective definitions will be employed. Finally, as has been pointed out earlier, the software architecture community is divided into a formalist and a practitioner's camp. Although formal methods are considered, the origins of this text is closer to those of the practitioner.

5.3 VIEWS, COMPONENTS AND CONNECTORS

Since the definition of components and connectors are dependent on the chosen view, these three concepts are considered jointly. In the next section, architectural styles are discussed.

Architectural views. The concept of architectural views is closely related to the concepts of components and connectors. A component may represent a process or a source code file or some specific functionality, for instance. All of these component types may be useful to the software developer and might be used as a meaningful base for reasoning about the system. By rigidly defining the meaning of the term component once and for all, many uses of software architecture would be lost. The concept of architectural views has therefore been proposed [Kru95]. A view is a set of definitions of what components and connectors represent in a specific model. Views suggested by Kruschten [Kru95] (albeit for an object-oriented context) include *the logical view*, where a component typically is an object or a class, *the process view*, where a component is a process or a thread, *the physical view*, where a component typically represents a processing node, and *the development view*, where a component may be a library or other source code entity.

The concept of views is based on the idea of separation of concerns [Dij76], where a view is intended to address certain aspects of the system while ignoring others. For example, the process view is considered suitable for matters relating to performance, while the development view may be employed for reasoning about reuse and maintainability [Kru95].

Not all researchers employ the architectural view concept. Firstly, some researchers seem to be particularly interested in a certain view; there seems to be a greater general interest in the logical and process views, for instance in [All97], [Sha96], [Luc95a], [Abd96], [Gac98], [Mor97]. Secondly, in some senses the view concept overlaps with architectural styles, discussed below. As views, styles allow for several definitions of components and connectors.

Components. The word *component* is as vague as the word *system*. It could mean almost anything, depending on the context. Also when reducing the context to software engineering, or even software architecture, component denotes a number of different entities.

According to Garlan and Shaw [Sha96a], a component is a *locus of computation and state*, exemplified by subroutines, interpreters, databases, etc. These components display their external behavior in their *ports* (or interfaces). The ADL Wright describes a component mainly as a computational process (including states) linked to port processes [All97]. Bass et al. at the SEI [Bas98] are more liberal in their interpretation of component, allowing processes, computational components, active and passive data components, classes, object methods, processors, processor groups, and systems. Examples of components, according to Bass et al., include web servers, operating systems, object request brokers, databases, functions such as target tracking, human-computer interfaces, and machines, such as workstations.

Extending into related fields, in the component framework area [Cha96] [Obj01] [Pri99], a component is mainly defined by its interfaces. A CORBA object specifies its interfaces in an Interface Definition Language (IDL) and incorporates stubs or skeletons to allow external interaction with its methods. When dealing with commercial off-the-shelf (COTS) products, components are to a large extent determined by vendors and their products. Microsoft Office might be considered a component since it is provided as a package by a vendor. In the COTS area, black-boxing constitutes another natural component boundary. A chunk of functionality that the developer cannot decompose may be considered a "de facto" component. Decomposition may be hindered by legal agreements, by an inability to access the source code, or by the lack of interest in the structure of the component. According to [Wal01], a commercial software component is released by a vendor in binary form with an interface for integration.

In this thesis, we allow different definitions of components depending on the context. In general, the context is defined by the architectural view, which in turn is determined by the concern at hand, i.e. what aspects we are interested in reasoning about. In Paper C, we strictly adhere to the definitions of Wright [All97]. In Paper A, Paper B and Paper D, the definitions presented in Section 4.6 are employed.

Connectors. Whatever components are, they seem more tangible than *connectors*. A connector is, according to [Sha96a] a *locus of relations*[6], and as such it does not represent a process, a processor, a library, or something else that in some sense is localized. Examples of connectors are method invocations, UNIX pipes, interprocess communication mechanisms, etc. Relating to architectural views [Kru95], connectors have primarily been defined for the logical and/or process views. A connector, such as a remote method invocation (RMI), includes the interaction between the client and its stub, the stub and an operating system, the operating system and the middleware (if middleware supports the RMI), internal processes in the operating system and/or middleware, perhaps network communication, interaction between the server skeleton and an operating system, the server and its skeleton, etc. A connector crosscuts all of the processes, libraries, and processors that support it. In this sense, a connector is, maybe more obviously than a component, a construction of the mind.

Perhaps because of this potential overlap– where a connector, to a large extent, is composed of components – the connector is a debated abstraction. Many definitions of software architecture do not include connectors, but settle for less prominent concepts, such as *relations*, or *interactions*, where the interaction issues are primarily specified in the components interfaces.

Where applicable in this work, connectors are used explicitly. The benefits of software architecture are, arguably, more dependent on the *contents* of interface, port and connector

[6] According to [All94a], *protocols that capture the expected patterns of communication.*

descriptions, than on their mere existence. Nevertheless, the introduction of the connector allows many issues related to the context, such as communication-related parts of operating system, to be explicitly considered.

5.4 ARCHITECTURAL STYLES

Arguably one of the main concepts behind software architecture as a discipline, assessment of the "goodness" of an architecture description is nevertheless difficult. The complexity of the underlying system, the complexity of the development process, together with the poor understanding of the relation between goodness and software structure turns architectural assessment into a formidable quest. Perhaps as a result of the problems of deduction-based prediction of system quality, an alternative approach has evolved, namely architectural styles. Instead of deducing the properties of an architecture, these properties are with styles induced. In other words, instead of attempting to determine the properties by applying a set of evaluation criteria on the architecture and its entities, the qualities of the architecture is assessed by comparison to other architectures with well-known properties.

Architectural styles are proven solutions to common problems [Gam98]. Typical styles are the pipes-and-filters, blackboard, client/server, layered, and main-program-and-subroutine styles [Sha96a] [Bas98]. These are all common ways of building software systems. As such they have, arguably, through evolutionary selection proven their value. A main argument is that there is little reason to be creative when approaching problem areas for which accepted solutions already exist. The whole point of architectural styles is thus that they are not new; contrary to many software engineering research products, the older an architectural style gets, the better.

Architectural styles are the same thing as high-level design patterns, although design patterns [Bec87] [Vli98] are usually found in an object-oriented context[7]. As is the rule in software architecture, there are diverging opinions on what one-liner best define the concept of architectural style (or patterns). The Gang of Four, generally considered the main reference of design patterns for software systems, define patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [Gam98]. In this definition, objects and classes need to be extended to architectural entities to make patterns equivalent with architectural styles. Garlan and Shaw [Sha96a] (representing a more formalistic approach to styles) define architectural style in terms of vocabulary (component and connector types), configuration rules (or topological constraints), and possibly semantic interpretations (whereby compositions have well-defined meanings), and analyses [Gar95a]. The definition of the SEI [Bas98] is similar, including component and connector types, topology, and semantic constraints.

[7] According to [Bas98], architectural style is another name for *system patterns*. System patterns and *design patterns* are subcategories to *architectural patterns*. According to [Mon97], styles can (more or less) be viewed as patterns.

There is also an ongoing discussion on what descriptions of architectural styles, or patterns, should contain. This discussion has to a large extent focused on the format of the style description; popular style formats include the *Alexandrian form* (cf. [Ale74]) and the *GoF format* (GoF denotes the Gang of Four, the authors of [Gam98]). Disregarding the format, certain information is considered necessary to represent a pattern or style, including name; problem to which the pattern is applicable; context in which the pattern is relevant; "forces", i.e., constraints and/or trade-offs; the solution that constitutes the actual pattern; examples, consequences of applying the pattern; rationale or justification; related patterns; and finally known uses [App00].

Styles as a base for architectural analysis are further elaborated on in the next chapter as well as in Paper D.

5.5 ARCHITECTURE DESCRIPTION LANGUAGES

According to the myth, software architecture was conceived in practice. Software developers, wishing to convey the structure of their system, depicted it with box-and-line diagrams, where a box constituted some kind of software component, and a line some kind of relation between the components. In this sense, software architecture has graphical origins. In the search for more stringency, these diagrams have been formalized in different ways. An architectural description language (ADL) is a graphical or textual language, with more or less formalized syntax and semantics, for describing these software architectures.

As usual, exactly what is to be considered an ADL is not agreed upon [Med00] [Cle96] [Cat95] [Kog95]. Generally, ADLs should support modeling of components and their communication via interfaces. Furthermore, communication integrity (i.e. that components only may communicate with connected components) [Luc95b], support for hierarchical composition [Sha96a], the ability to model dynamic architectures [Luc95b], property assertions [Sha96a], and analysis support [Sha96a] are considered desirable properties.

ADLs normally consist of sets of sub-languages addressing different concerns (cf. views). For a language to be classifiable as architectural, a structural sub-language is typically required, enabling the description of entities (components and possibly connectors) and their relations. Oftentimes, type sub-languages are also included, enabling the description of types of entities. Constraint sub-languages enable the description of e.g. style constraints. Finally, behavioral languages describe the temporal workings of the architectural entities, such as event sequences and synchronization. Evidently, ADLs are mainly the domain of the formalistic software architects, allowing stringent inferences based on explicit models. As such, they are a locus of the deduction-based analysis approach. In Chapter 6, ADLs and their analysis potential are covered in more detail.

5.6 ARCHITECTURE IN THE SOFTWARE PROCESS

The software development process became a topic of research in 1970, when Winston Royce wrote an article on the management of the development of large software systems, describing the well-known Waterfall method [Roy70]. Still constituting the basic process model of software development, the Waterfall model specifies a number successive phases, including system requirements, software requirements, analysis, program design, coding, testing, and operations. This model has, however, since its inception been much criticized, among other things, for its rigidity [Boe88] [Boo99]. Improvements that have been suggested include iterative development, prototyping, concurrent development, incremental development, and many more.

The architectural design phase is generally considered to be located in the early parts of the design phase or possibly the late parts of the analysis phase. According to Hofmeister et al. [Hof99], representing the Rational Unified Process (RUP) standpoint, the software architecture phase comes after domain analysis, requirements analysis, and risk analysis, and before detailed design, coding, integration, and testing. In some texts, however, the role of software architecture seems surprisingly prominent (Figure 4).

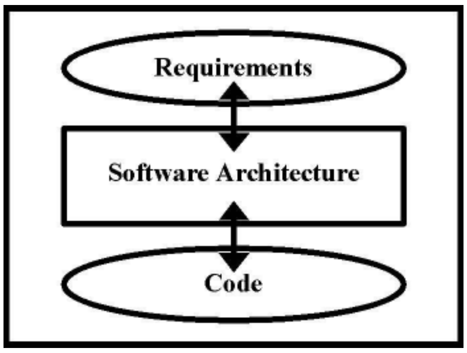

Figure 4. Software architecture in the software process, according to [Gar00].

A main idea of software architecture is to mitigate different kinds of technical problems and risks early on in the software process. Problem identification is typically performed by some kind of reasoning based on the architectural description, ranging from gut feeling to formal analysis. An often-cited architectural analysis process is the Architecture Tradeoff Analysis Method (ATAM) [Kaz98] [Kaz99]. The name is slightly misleading, since the ATAM per se does not contain specific analysis methods or techniques, but rather proposes an engineering process in which the analyses may be performed. The most prominent feature of the ATAM is its scenario-based approach, where scenarios are elicited and subsequently analyzed by some appropriate analysis technique. Similar methods have been proposed by other authors [Ben00] [Las02]. A benefit of the scenario-based approach is that the need for universal definitions of extra-functional requirements are substituted by concrete situations. For instance, soft requirements (e.g. modifiability), well-known for their unquantifiable nature, do not require specific metrics but can instead be operationalized in the form of change scenarios[8].

As most processes, the ATAM is presented in sequential activities, or steps. These include 1) scenario collection, where potential usage and change scenarios are elicited from involved stakeholders; 2) architecture representation, where the current architecture is described; 3) property-specific analysis, where, by available means, the potential impact of the scenarios on the architecture is assessed; 4) trade-off analysis, where sensitive architectural elements are identified by their frequency in the property-specific analyses; and finally 5) propose modifications, where the analysis results are used as a base for architectural improvement.

Although software architecture may be relevant in other situations, such as software modifications, reengineering, etc., this thesis in focused on the above considered, process-early, phase. The analysis process is further considered in Paper B, where scenario-based analysis processes are considered in the light of enterprise software system integration.

5.7 Enterprise software architecture

Traditional software architecture has mainly been preoccupied with green-field development of systems typically developed by a single vendor. For example, the standard reference on the subject, by Shaw and Garlan [Sha96a], includes case studies on Parnas's KWIC system [Par72], an oscilloscope instrumentation system, a hypothetical robot controller, a cruise-control system, and a chemical process control system. Hofmeister et al. [Hof99] includes case studies on an image acquisition and processing system, an instrumentation and control system, an embedded real-time patient monitoring system, and a central patient monitoring system. All of these systems are developed by (at most) a single vendor, and are reasoned about with the implicit assumption that the system can be developed from scratch, or at least that all of the source code is available. For many software developers, the green-field assumption may be viable, but for many others, management of legacy systems and COTS constitutes a major part of the development process. In particular, user (or customer) organizations face a management process focused on procurement and integration, and with little in-house development. In this thesis, the term *enterprise software system* refers to the systems of these organizations. The architecture of these systems is here denoted *enterprise software architecture.*

In this section, the characteristics of enterprise software systems and architectures are elaborated on. Systems and their architectural descriptions are considered jointly, as an architectural vocabulary is employed to describe the systems. The relations to the (acronymified) areas of enterprise application integration (EAI) [Lin00], component-based soft-

[8] Similarly, usage scenarios have been proposed for operationalizing usability [Bas01b].

ware engineering (CBSE) [Hei01a] and commercial off-the-shelf (COTS) systems [Wal01] are also considered.

5.7.1 ENTERPRISE SOFTWARE SYSTEM EVOLUTION

Considering the software systems of typical medium-sized to large enterprises in the industrialized world, they are surprisingly similar. Most enterprises need software systems to manage their economy, their employees, their customers, and their subcontractors. They may have customer information systems, call-centers, time management systems, and payroll systems. They require software systems to track inventories, to manage billing, to manage their assets. They may employ systems for analysis and forecasting. If they are producing companies, they may need systems for product design, systems to develop, to plan, to track, and to supervise and control the production. If they have a geographically distributed operation, they may employ geographical information systems. The list goes on. There are generic systems relevant for most enterprises, there are industry-specific systems, and perhaps, in some cases, there is a need for a few company-specific systems.

The evolution of these systems of systems can be described in four eras. First, there were the "stovepipe" systems; isolated systems that had little need to communicate with their neighboring systems and where unprepared to do so. If the output from one system occasionally was needed in another, the information transfer could be manually accomplished.

As the companies automated their business and more information was digitalized, this approach, however, became inefficient. Those systems that needed to communicate where then integrated if possible, but since the systems generally were independently developed, with no provisions for future integration, expensive customized solutions between in-house interfaces were the typical result. This was the point-to-point era. As the point-to-point approach was adopted, however, it became clear that the cost for system integration was high, in many cases too high to justify the integration. Furthermore, the emerging enterprise software system became difficult to manage with many customized connections between systems. With the point-to-point approach, the introduction of one new system typically requires many specialized connections to the existing systems.

In an attempt to reduce the complexity of the enterprise software system, the Enterprise Resource Planning (ERP) systems were introduced. Exploiting the similarities of most companies of the computerized world, vendors such as SAP [Sap02] and Baan [Baa02] offer giant systems, covering many of the functions that previously needed to be procured separately. A main benefit of the ERP systems, from an architectural point of view, was that the components/systems were developed by one vendor, and prepared for integration with each other. Communication between the pay roll system and the accounting system was thus prepared, and cost little, if anything, for the customer. This was the ERP era. During recent years, however, more and more failed ERP implementation projects have been reported. This has been attributed to several causes, including poor organizational fit, an unhealthy dependence on the vendor, and too extensive customizations [Hon01]. ERP vendors have also been criticized for providing non-standardized in-house interfaces, thereby allowing integration only with their own products, hindering their customer from picking the best components from different suppliers [Lin00].

Now the hype is for Enterprise Application Integration (EAI) solutions, including message brokers, application servers and the like. These products are specifically designed for facilitating the integration of legacy systems, ERP systems, and other COTS systems that are notoriously difficult to access. This is thus the EAI era. It may be worth noting that very few companies have managed to fit themselves into one specific era; most enterprises have elements of all eras in their enterprise software architecture.

In parallel with the evolution described above, enterprises have moved from custom-developed software projects to COTS procurement. The costs of developing the desired systems are generally higher than a single company can afford, and the time from the investment decision to a working system is too long. By procuring off-the-shelf software, the enterprises loose control over the software, but hope to gain in economy and time. However, few are the procured COTS systems that are not customized to a smaller or larger extent in the implementation phase.

The remaining part of this section considers what characterizes today's enterprise software systems and architectures and how they compare to the traditional notion of software architecture. Chapter 3 briefly describes the typical enterprise software system of electric utilities on the deregulated Scandinavian electricity market, the primary empirical base of the present text.

5.7.2 CHARACTERISTICS OF ENTERPRISE SOFTWARE ARCHITECTURE

In traditional software architecture, a component may be a procedure, a process, and object-oriented object, etc (cf. section 4.3). These kinds of components are, however, difficult to employ to describe the enterprise software system for several reasons. Firstly, the sheer size of the enterprise software system in terms of processes or objects is overwhelming. The use of architecture as a high-level abstraction would thus be lost. Secondly, in most enterprise software systems, it is not possible to discover components of this granularity. This is due to what could be called "organizational encapsulation", i.e., much software employed by a user organization is not accessible for modifications or even for inspection. There are several reasons for this encapsulation. In the (common) case of procurement of externally developed systems, the source code is often not available, nor are any design specifications. Even if this information were available, chances are that access would be legally restricted by the developing organization. In the case of legacy systems, relevant documentation is often lacking, and the people who developed the systems are often no longer available. This results in an involuntary encapsulation of software into "atoms", "indivisibles", or "components". In many cases, the boundaries of these components are thus set by what chunks of software the vendor decides to provide as a product; in other cases, the boundaries are set by tradition. Below, the characteristics of enterprise software systems are further elaborated on, considering components, connectors, and system-level properties.

Components are not modifiable. The determinants of what are components in the enterprise software system have a number of effects. One such effect is that components oftentimes cannot be modified. Without access to source code and documentation, and possibly with legal hindrances, commercial components become black-boxes. For all practical purposes, so do poorly understood legacy components. Therefore, changes to the system are preferably handled indirectly, either by influencing the software vendor to adapt its packaged product in coming releases, or by implementing non-intrusive modifications, e.g. by wrapping a component in order to change its external behavior.

Components are heterogeneous. In traditional software architecture, components are homogeneous. This means that all components are based on a number of common assumptions. In an object-oriented system, for instance, all components are objects. They have the same fundamental structure with data and behavior separated, with private and public attributes and methods, they are created and destroyed in the same manner, they interact with the operating system in the same manner, they probably even execute on the same platform (or at least on platforms providing common services). In the enterprise software system context, however, components are heterogeneous. The buyer of a COTS system does not normally decide how many processes are running in parallel in the procured component, nor does the buyer typically determine whether the component features a CORBA interface or not. Furthermore, today's enterprise architect did not determine the properties of yesterday's legacy systems. Thus, in an enterprise software system that to a large extent is composed of the combination of packaged and legacy components from a wide range of vendors, epochs, and intended purposes, uniform system components normally proves hard, costly, and probably even inappropriate to realize.

Components are large-grained. The enterprise software system is a "system of systems" in the sense that the components of the enterprise system are normally considered as systems in the (developer-oriented) traditional software architecture. Using any measure, (e.g. number of objects, number of processes, number of filters), the enterprise components become large-grained. For instance, complete systems such as customer information systems and geographical information systems might be considered as components in enterprise software architectures.

The supply of packaged components is limited. Traditional development allows the specification and development of any technologically feasible component. The software architect thus distributes requirements over a set of components, and makes sure that the required functions and properties are implemented according to the architectural specification. In the enterprise software context, this is not a viable option. Green-field development is generally a far too expensive undertaking, and the component options then become those available on the market. Unfortunately, the number of large-grained components available on the market is limited. Often components with the desired functionality do not exist, forcing enterprises to combine packaged software to fulfill the requirements.

Connectors are heterogeneous. In traditional software architecture, connectors are preferably determined once and for all when the developer selects the architectural style. If a pipes-and-filter style is selected, then the connectors must be pipes. If only the components are filters, this presents no problem since the two are compatible. In most enterprises, however, the choice of connectors is influenced by the interfaces provided by the acquired base components. Unfortunately, the probability that two independently developed components will feature compatible interfaces is annoyingly small. There is a great number of different kinds of interfaces (e.g. remote object interfaces), and for each kind, there are typically competing standards (e.g. CORBA and COM). Furthermore, although the situation arguably has ameliorated during the latest decade, many components follow no interface standard at all. And it is not enough that only two components provide the same interface, for typically, a new component requires integration with a number of existing ones. The consequence of this interface disharmony is heterogeneous connectors. Connectors in an enterprise software system are thus by nature diverse since their main purpose is to glue heterogeneous component interfaces together. Moreover, connectors do not only interface with components within the same enterprise software system, they also provide interfaces to other organizations' software systems bringing even more heterogeneity into the enterprise's total battery of connectors.

The enterprise software system may contain both data and functional redundancy. In a system controlled by a single developer, one piece of information is typically stored in one place and one piece of user functionality is only coded once. The integration of packaged components, however, increases complexity since those components rarely correspond completely to the organization's requirements. In order to grasp the bulk of the requirements, different packaged components are typically combined, often resulting in both functional and data redundancy. Data redundancy is typically the result of the integration of several related components including their own databases. An electricity metering system, for instance, may hold information on the type and identification of metering equipment. This information may also be present in an asset management system. More or less by default, data redundancy is subject to the risk of inconsistency. Similarly, for instance in the case of a company merger, if two different asset management systems are integrated on the database level, duplicate functionality may operate on the common data. Ignoring potential concurrency problems, functional redundancy is normally not a problem if the functions are in fact identical, but if they are slightly different, they may cause data inconsistency as well as inconsistent behavior.

The legacy architecture constitutes the starting point of the enterprise software system development effort. Although it is not always the case, traditional software architecture often assumes the possibility of development from scratch. Occasionally, this is the case for software developing organizations, and in these cases, the design latitude is fairly unrestricted. In the context of enterprise software systems, the legacy is often huge and the legacy components often constitute a significant asset that may not be easily replaceable without severe disturbances to business operations. Enterprise software system development is thus rather a modification activity than a clean-slate development project.

It is fairly clear that these characteristics of the enterprise software system affect the task of architectural design and analysis. The design space [Lan90] becomes discrete, component are selected rather than designed, and integration becomes a primary activity. The implications are many. These issues are further elaborated on in Paper A, Paper B, and Paper D.

5.7.3 CBSE AND COTS

The industrial revolution was characterized by "the transition from cut-to-fit craftsmanship to the automated mass-production of goods from interchangeable parts" [Cza00]. The fundamental idea of component-based software engineering (CBSE) is to revolutionize the software industry in the same manner. This is at least one interpretation of component-based software engineering. A second interpretation is the commercial-off-the-shelf view of component-based software engineering, which rather focuses on "the manual production of software systems from components that typically need to be modified to be interchangeable." Although admittedly less visionary, this second view is perhaps more applicable to today's software milieu. Generalizing, there are thus two major interpretations of the term "component" in CBSE: component-framework component [Lon01], and COTS (commercial-of-the-shelf) component [Obe98]. Component-framework components are standardized with respect to (at least) their interfaces and are exemplified by CORBA, COM and EJB components. Referring to the categorization of Chapter 4, the integration approach is thus monarchical or oligarchical. COTS components, on the other hand, are not necessarily standardized at all, placing COTS component-based software engineering in the anarchical integration category.

From the perspective of this text, the views on and methods for software development represented by the component-framework CBSE area (e.g. as represented by [Szy98], [Hei01b] and [Lon01]) differ little from traditional software architecture. Although the differences may be significant from other viewpoints, most of the differences between enterprise software architecture and traditional software architecture discussed above are equally applicable to the differences between enterprise software architecture and component-framework CBSE. In particular, enterprise software systems allow heterogeneous connectors. This is exactly what component frameworks attempt to overcome by standardization.

COTS development, in particular as considered by Kurt Wallnau et al. [Wal01], is much closer to the enterprise software systems discussed herein. For instance, the components considered by the COTS field are often unmodifiable, connectors may very well be heterogeneous, and sufficient component information is often lacking. In relation to the perspective of the present text, the COTS field does, however, not focus on the software architecture of these systems.

Both of these component-based perspectives on software engineering are affecting the views of software architecture. Traditional software architecture is now moving from the green-field assumption towards the explicit recognition of component-based software engineering as a main influence on the architectural design process (cf. e.g. the reuse considerations of product-lines [Bos00] [Bas98] [Jaz00]).

5.7.4 EAI

As the name indicates, enterprise application integration (EAI) [Lin00] [Lin01a] [Lin01b] [Mor01] [Ruh01] concerns the integration of enterprise applications. This field is covered in greater detail in section 4.5. The purpose of this section is to highlight the similarities and differences between the enterprise application integration field and enterprise software architectures. Firstly however, there is a distinction to be made between EAI as a practice and EAI as a technology. As a technology, EAI typically refers to non-intrusive application integration techniques aimed at creating loosely coupled enterprise software systems. Message brokers, adapters, process automation tools, and similar modern products are hallmarks of the EAI technology. As a practice, however, EAI is basically concerned with the integration of systems that were not developed for integration (anarchical integration in the terminology of Chapter 4).

The addressed problem area is thus similar to that of enterprise software architecture integration, considering the integration of software systems based on legacy and externally developed, packaged components. All of the characteristics of enterprise software systems are also generally applicable to enterprise application integration. Generally, however, enterprise application integration is not particularly concerned with software architecture. Neither is it concerned with structured forms of analysis of architectures or systems. The focus is rather on specific practical approaches for integration, as presented in Section 4.5.

5.8 SUMMARY

This chapter has presented the concepts of software architecture and enterprise software architecture. There are currently several competing definitions of software architecture. In particular, there is a division between formal, theory-oriented, or deduction-based software architecture on the one hand, and informal, practice-oriented, or induction-based software architecture on the other. Most authors do however agree that important concepts in software architecture include components, connectors, views, and styles, typically represented in an architectural description language. It is also generally agreed that architecture may be relevant in a number of places in the software process, but a particular focus is typically placed on the project-early phases. This chapter has considered these central concepts as well as alternative interpretations of them.

The chapter also proposes the concepts of enterprise software systems and architectures. These are characterized by a limited supply of unmodifiable, heterogeneous, large-grained components linked by heterogeneous connectors. The enterprise software system may contain both functional and data redundancy, and the legacy architecture constitutes the starting-point of the system development effort.

Papers A and B contain supplemental descriptions of the enterprise software architecture. Deduction-based architectural analysis of enterprise software system integration is further considered in the next chapter. The software process aspects of architectural analysis (considered in this chapter) are further explored for the context of enterprise software systems in Paper B. Paper D considers architectural styles for enterprise software systems.

Chapter 6
Architectural Analysis

6.1 INTRODUCTION

There are many questions that a software developer would like to know the answer to before a system is developed. Will the system be sufficiently fast? Will it be reliable? What happens if the user behaves unexpectedly? Will the system be secure? Will it be difficult to port it to a new platform?

In this chapter, two conceptually different approaches for assessing these properties are reviewed, induction-based and deduction-based architectural analysis. Deduction-based architectural analysis is a reductionistic method, attempting to calculate the attributes of the modeled system by analyzing how its constituents and their interactions jointly display certain system-wide properties, such as reliability. This approach is the same as is used when attempting to predict the behavior of a molecule by analyzing the properties and interactions of the nuclei and electrons of which it is composed. In software architecture, the deduction-based analysis approach is most prominently represented by the formally established methods. These are methods based on well-defined architecture description languages with associated formal analyses, such as Wright [All97], Darwin [Mag95], Rapide [Luc95a], and more.

Induction-based analysis is based on a statistical argument, attempting to, based on certain criteria, classify a specific system into a category and predict further characteristics of the system based on the category to which it belongs. In the molecular analogy, as soon as a molecule has been classified as ethanol, it can be predicted flammable, since all other ethanol molecules are flammable. In software architecture, the induction-based approach is most prominently represented by the architectural style community [Bus96] [Sch00]. Styles, representing classes of systems found in the real world, are typically defined by their architectural traits and associated with certain properties (such as modifiability), implying that any system belonging to a certain style, will display those properties that are generic for the style.

The chapter begins with a general discussion on induction-based approaches, but since Paper D discusses architectural styles in some detail, the main focus of the chapter is on deduction-based approaches. In line with the thesis objectives, the deduction-based sections are particularly concerned with the analysis of integrability. Section 5.4 presents an evaluation of the applicability of current approaches to this problem. Section 5.5 concludes the chapter with a discussion on the appropriateness of these methods in the context of enterprise software systems.

6.2 Induction-based analysis methods

The software engineering world is full of more or less established relations between the software design and the qualities of the implemented system. For instance, a *modularized* system is considered more flexible and comprehensive than a monolithic system [Par72]. The more *go to* statements a program contains, the messier it is, according to Dijkstra [Dij68a]. A *conceptually integrated* system is arguably faster to build and to test than it's opposite [Bro75]. *Nested factorization* of a program is believed to improve the correctness. The *simpler* the composition scheme is, the more intellectually manageable is the program [Wir74]. A *hierarchical software structure* aids design verification, thereby improving the correctness [Dij68b]. Low *coupling* improves understandability, correctness and changeability and high *cohesion* improve ease of development, maintainability, reusability, and reduces fault-proneness [Ste74][9]. Pipe-and-filter systems support reuse and are easy to maintain [Sha96a].

Many of the relations presented above refer to the organization of software components, be they layers, modules, or other. Not surprisingly, a software architecture description depicts this: the organization of software components. According to the relations presented, then, some architectures are more reusable, maintainable, manageable, etc. than others. In this spirit, induction-based a*rchitectural analysis* is the attempt to evaluate the "goodness" of a software system's architecture. As mentioned, this section focuses on architectural styles as prime example of induction-based analysis. Architectural styles were introduced in Section 5.4.

Induction-based methods address the task of architectural analysis quite differently from their deduction-based relatives, drawing conclusions by analogy to similar architectures rather than based on formally expressed rule sets. Causation is thus substituted by association, in the sense that the main concern no longer is why an architecture demonstrates certain properties, only that the properties appear when the architecture takes on a particular form (i.e., style). Induction-based analysis methods are thus based on empirical relations rather than theoretical constructs. The only theory relevant for an architectural style is the relation between the style and the system properties that are associated with the it, e.g. the

[9] [Ste74] according to [Bri99].

relation between the black-board style and the extra-functional property modifiability. Because the underlying theory is so meager, there are few restrictions on the employed abstractions. Similarly, induction-based methods do per se not make extensive assumptions about the considered system. The only characteristics of the system that are explicitly assumed are those that define the style description. Because of their statistical nature, induction-based methods are generally related to externally measurable properties, e.g. integration cost or throughput.

6.2.1 EXTRA-FUNCTIONAL PROPERTIES

"Goodness" is often measured in quality attributes [Bar95a], also known as emergent properties, non-functional properties, "ilities", extra-functional properties or simply qualities. This multitude of terms is not a coincidence, for quality attributes are difficult to define. The term "non-functional" is devised as an antonym to functional requirements. Most requirements specifications describe functionality in a standardized way, for instance by using use cases. Non-functional requirements are in this context all those requirements that cannot be specified by use cases. The proposition of the term "extra-functional properties" is an attempt to nuance the relation between function and quality, indicating that these qualities are not the opposite to functional properties, but rather something beyond them. "Emergent properties" indicate those properties of a system that are not directly related to the components or functions, but emerge when they are composed into a system. "Ilities" refers to all properties that end with the suffix "ility". "Quality attribute" indicates that the property is related to the quality, or "goodness," of the system.

One way of defining quality attributes is by examples (cf. Table 4). Typical quality attributes of software systems are performance, scalability, maintainability, portability, interoperability, and reliability. These properties of the system are typically not satisfied by the introduction of a singular function or component, so they are extra-functional and emergent. Five of six of them end with the suffix "ility", and they are all aspects of goodness.

Quality attributes are often categorized into run-time (or operational) quality attributes, and design-time (or developmental) quality attributes [Bos99]. Run-time attributes are observable when executing the system, such as performance or reliability. Design-time attributes are typically related to the software process, e.g. modifiability, portability, etc.

Accuracy	The precision of computations and control.
Audibility	The ease with which conformance to standards can be checked
Availability	The probability that the system functions correctly.
Communication commonality	The degree to which standard interfaces, protocols, and bandwidth are used.
Completeness	The degree to which full implementation of required function has been achieved.
Conciseness	The compactness of the program in terms of lines of code.
Consistency	The absence of contradictory data in the system. The use of uniform design and documentation techniques throughout the software development project.
Correctness	The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
Data commonality	The use of standard data structures and types throughout the program.
Ease of creation	The difficulty of constructing the system. This is often measured in labor hours.
Efficiency	The amount of computing resources and code required by a program to perform its function.
Error tolerance	The damage that occurs when the program encounters an error.
Execution efficiency	The run-time performance of a program.
Expandability	The degree to which architectural, data, or procedural design can be extended.
Flexibility	The effort required to modify an operational program, or, the ease with which the systems can be adapted to changes.
Generality	The breadth of potential application of program components.
Hardware independence	The degree to which the software is decoupled from the hardware on which it operates.
Instrumentation	The degree to which the program monitors its own operation and identifies errors that do occur.
Integrability	The ability to make the separately developed components of a system work correctly together. The ability to make the separately developed systems work correctly together
Integrity	The extent to which access to software or data by unauthorized persons can be controlled.
Interoperability	The effort required to couple one system to another. The ability of a system to work with another system
Maintainability	The effort required to locate and fix an error in a program (this is a very limited definition).
Modifiability	The ability of a system to be extended to accomplish additional functionality.
Modularity	The functional independence of program components.
Operability	The ease of operation of a program.
Performance	The measure of how well the computer system responds to its inputs. Common measures are response time, resource utilization, and throughput.
Portability	The effort required to transfer the program from one hardware and/or software system environment to another. The ability of a system to execute on different hardware and software platforms.
Reliability	The extent to which a program can be expected to perform its intended function with required precision.
Reliability	The ability of he system to sustain operations. A common measure is mean time between failures.
Reusability	The extent to which a program [or parts of a program] can be reused in other applications.
Scalability	The ability of a system to support modifications that dramatically increase the size of the system.
Security	The availability of mechanisms that control and protect programs and data.
Self-documentation	The degree to which the source code provides meaningful documentation.
Simplicity	The degree to which a program can be understood without difficulty.
Software system independence	The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints.
Testability	The effort required to test a program to ensure that it performs its intended function.
Traceability	The ability to trace a design representation or actual program component back to requirements.
Training	The degree to which the software assists in enabling new users to apply the system.
Usability	The effort required to learn, operate, prepare input, and interpret output of a program.

Table 4. Quality attributes, "ilities", non-functional properties, or emergent properties according to [Bos99] [Kaz94a] [Kaz94b] [Las99] [McC77].

There are few, if any, software quality attributes that are well defined in the sense that they have a single generally agreed upon measurement. Availability could be measured as up-time per year in percent, number of failures per year, down-time per day, up-time per year in percent weighted according to function usage frequency, the probability that the system functions correctly at time t, as t approaches infinity [Bar95a], and so on. Furthermore, several properties are qualitative and subjective to their nature. It is for instance difficult to imagine that usability would be universally and objectively quantifiable.

6.2.2 ARCHITECTURAL STYLES AND EXTRA-FUNCTIONAL PROPERTIES

One of the main reasons for employing architectural styles is thus that their application is considered to result in particular qualities of the system. To propose some further examples: according to Schmidt et al. [Sch00], application of the *interceptor* architectural pattern enhances extensibility, flexibility, and reusability. Buschmann et al. [Bus96] argue that the *layers* architectural pattern results in reusability and exchangeability. The Gang of Four [Gam98] claims that the consequences of the *interpreter* design pattern include changeability, extensibility, and ease of implementation. The list goes on, for every architectural style in literature, affected quality attributes are listed.

So is there any reason to believe that architectural styles actually do what they claim? Perhaps the strongest argument is the general community agreement on induction-based rules as those proposed by Dijkstra, Parnas, and Wirth (presented in the introduction to this section). The large acceptance that styles have found in the software community is a related indication; the Amazon.com internet book shop lists over 40 books on the subject[10] and dedicated international conferences are held, mainly concerned with the presentation of new patterns [Plo02]. Furthermore, although the main approach is clearly induction-based, the relations between quality attributes and architectural style are normally supplemented by more deduction-based arguments (in some cases, formal approaches are combined with styles or patterns [Mik98]). To some extent, these arguments strengthen the credibility of the styles[11]. However, few statistical surveys have been conducted on the relations between styles and qualities. Moreover, architectural styles are typically defined as proven solutions to common problems. As software is an artificial science, one might argue that this definition lends itself to self-fulfillment. A developer determined to build a modifiable system will probably choose an architectural style that is generally considered modifiable, but he or she will also make a number of other design choices to this effect. The relation between styles (or patters) and qualities might thus be spurious. This argument is tightly linked with the ongoing discussion on the possibilities of codifying knowledge [Add02]; to apply a style successfully, expertise may be required, and if expertise is available anyways, its codification in styles is unnecessary and perhaps even unsuccessful. The relation is further complicated by the fact that few researchers, if any, argue that architectural style is a sole determinant of any quality attribute. Even the most robust architecture can be thwarted by a malicious implementer. This means that the relation "good architectures yield good systems" is not

[10] Styles and design patterns.

[11] Perhaps somewhat surprisingly, the architectural formalists primarily consider styles from a deduction-based perspective. Styles are in this approach specified formally to guarantee certain properties. Specific architectures are then checked for style-conformance. These styles are thus deduction- rather than induction-based. When this text speaks of styles, it primarily refers to the induction-based kinds, i.e., empirically, rather than theoretically, proven solutions to common problems.

generally valid. Instead one must settle for weaker positions like “bad architectures yield bad systems” or “good architectures *permit* good systems”.

Summarizing, although the relation between styles and qualities is not proven beyond doubt, and although there are obvious limits to the effects styles may have on system characteristics, there is a fairly strong case for a link between the concepts. Architectural styles applied to enterprise software systems are further elaborated on in Paper D.

6.3 DEDUCTION-BASED ANALYSIS METHODS

In this section, a number of architectural deduction-based analysis methods are presented on a conceptual level. In the subsequent chapter, the abilities of these methods to address software integration issues are considered. In the final section, the applicability in the enterprise software system context is considered.

There are several reviews of (formal) architectural description languages (architectural description languages are introduced in Section 5.5), including [Cle96] [Cat95] [Kog95] and [Med00]. These reviews consider slightly different sets of languages. The present review is contains a language set similar to that found in [Med00], and includes Aesop [Gar94b] [Gar95b], C2Sadel [Med96] [Med98] [Med99] [Rob98], Darwin [Mag95] [Mag96] [Mag97a] [Mag97b], MetaH [Ves98], Rapide [Luc95a] [Luc95b], SADL [Mor95] [Mor97a] [Mor97b] [Rie99], UniCon [Sha95b] [Sha96b] [DeL99] [Zel96], and Wright [All97]. Note that several of the analyses performed using one language could be performed with some other language as a base. Here, the languages are mainly used as a convenient categorization of research efforts; the potential of the languages per se is considered only to a limited extent. In addition to the approaches considered by [Med00], we also include the architectural mismatch analysis presented by Abd-Allah and Gacek [Abd96] [Gac98], since this is a similar architectural deduction-based approach of a novel issue.

6.3.1 C2SADEL

Based on type theory and particularly directed at applications with a graphical user interface aspect, C2SADEL (Chiron-2 Software Architecture Description and Evolution Language) is an architectural description language constructed in conjunction with the C2 architectural style [Med98] [Med99]. The base constructs of the language are *component types, connector types* and *topology*. Components have *names*, *interface elements*, *behavior,* and possibly an *implementation*. Interface elements are either *provided* or *required* by the component, they have *names*, *parameter* sets, and possibly a *result*. The behavioral semantics are described by component *invariants* and provided or required *operations* with *pre-* and *postconditions*. The language is formalized using the Z notation [Spi89].

With this formalism it is possible to specify type-checking predicates. Specifically, a service provided by component will satisfy the service required by another component if their interfaces match as well as the pre- and postconditions of the involved operations. In analogy with Moormann Zaremski and Wing [Moo95] [Moo97], these matches may be more or less relaxed.

```
Component DeliveryPort is
   Subtype CargoRouteEntity (int \and beh) {
      State {
         Cargo: \set Shipment;
         Selected: Integer;
         …
      }
      invariant {
         (cap >= 0) \and (cap >= max_cap);
      }
      interface {
         prov ip_selshp: Select (sel: Integer);
         req ir_clktck: ClockTick();
         …
      }
      operations {
         prov op_selshp: {
            let num : Integer;
            pre num <= #cargo;
            post ~selected = num;
         }
         req or_clktck: {
            let time : STATE_VARIABLE;
            post ~time = time + 1;
         }
         …
      }
      map {
         ip_selshp -> op_selshp (sel -> num);
         ir_clktck -> or_clktck ();
         …
      }
}
```

Figure 5. Sample component type specified in C2SADEL [Med99].

For C2SADEL, consistency checks are also possible, determining whether components and connectors are properly specified, instantiated, and connected, whether component interfaces are correctly mapped to operations, and so on. Also constraint checks are available, e.g. ensuring that the system adheres to a specific style.

From a C2SADEL specification, skeleton code may be generated (currently in C++, Ada and Java), where pre- and postconditions are commented into the program text in appropriate places. In fact, C2SADEL provides a framework of abstract classes for its components, connectors, etc. in the supported languages. Several off-the-shelf middleware technologies have been integrated with the framework to enable interactions between C2 components in different languages.

6.3.2 SADL

The Structural Architecture Definition Language (SADL) is built on logic – specifications can be systematically translated into logical theories of (an extended) first-order logic – and bases its analysis capabilities on theorem proving. A software architecture in SADL is represented by components with types and an interface of ports, connectors, configurations, mappings, and architectural styles. Connectors are, as components, typed and treated as first-class entities (refineable). Accepted data types of connectors are specified. Ports have direction, type, typed parameters and return values. Configurations contain connections between connectors and ports and constraints. Mappings, a central element to SADL, define the relation between abstract and more concrete architectures. Styles include defined types, constraints, and connector semantics. These basic entities constitute the core structural model. On top of this model, arbitrary semantic layers may be introduced for representing behavioral or non-behavioral aspects of the design, such as latency, or dataflow. The main concern of the developers of SADL is correct architecture refinement [Mor95], but also architectural analysis is central.

```
compiler_L1: ARCHITECTURE
       [char_iport: SEQ(character) -> code_oport: code]
   IMPORTING character, code, token, binding, ast FROM compiler_types
   IMPORTING Function FROM Functional_Style
   IMPORTING Dataflow_Channel, Connects FROM Dataflow_Style
BEGIN
COMPONENTS
   lexical_analyzer: Function
       [char_iport: SEQ(character)
          -> token_oport: SEQ(token), bind_oport: SEQ(binding)]
   parser: Function
       [token_iport: SEQ(token) -> base_ast_oport: ast]
   analyzer_optimizer: Function
       [base_ast_iport: ast, bind_iport: SEQ(binding)
          -> full_ast_oport: ast]
   code_generator: Function [full_ast_iport: ast -> code_oport: code]
   CONNECTORS
       token_channel:   Dataflow_Channel<SEQ(token)>
       bind_channel:       Dataflow_Channel<SEQ(binding)>
       base_ast_channel:   Dataflow_Channel<SEQ(ast)>
       full_ast_channel:   Dataflow_Channel<SEQ(ast)>
   CONFIGURATION
       token_flow: CONNECTION
          = Connects(token_channel, token_oport, token_iport)
       bind_flow: CONNECTION
          = Connects(bind_channel, bind_oport, bind_iport)
       base_ast_flow: CONNECTION
          = Connects(base_ast_channel, base_ast_oport, base_ast_iport)
       full_ast_flow: CONNECTION
          = Connects(full_ast_channel, full_ast_oport, full_ast_iport)
   END compiler_L1
```

Figure 6. Sample component type specified in SADL [Mor97b].

General consistency and type checking[12] is employed to ensure the well-formedness of the architecture. Further analysis of architectures is typically performed by applying proofs of property constraints. Because of the extensible semantic layer structure, the number of potential analyses is unlimited and the potential for analysis of specific properties is case-dependent. Security [Mor97a] analyses have been performed for specific architectural

12 Type checking is a matter of showing that type constraints are satisfied.

styles. In a more ambitious continuation, assessment "intrusion tolerance" is under investigation [Sta01].

The security analysis of SADL performed by proving that the Bell-LaPadula [Mor97a] multilevel security (MLS) policy is not violated. Briefly, the policy allows read and write access only to subjects with the proper clearance level, thus all inappropriate data flows in the architecture are forbidden. The analysis is carried out by matching mechanisms (i.e. the Bell-LaPadula policy) to the general security property, thus decomposing the main property into simpler and verifiable architectural properties[13] (the Bell-LaPadula policy is presented as two verifiable properties[14]) [Sta01]. An important limitation of this work is that the Bell-LaPadula policy is not a definition of security, but an example. Thus, an implementation of the Bell-LaPadula policy may be deemed secure, but employing solely Bell-LaPadula as evaluation criterion for the security property of unknown systems will undoubtedly rule out many perfectly secure architectures. Furthermore (and related), whether the Bell-LaPadula policy is in fact enforced in an implemented system does of course depend on a number of unconsidered factors, including *how* access is to be prohibited in components and connectors.

6.3.3 Darwin

Darwin is, according to its authors, a declarative binding language (or configuration language) with a precise operational semantics defined in π-calculus [Mag95], and providing multiple views. On top of the basic structural view, a behavioral and a construction[15] view have been elaborated.

The structural view is based on components as first-class, typed and named entities, providing services to the environment using interfaces. Component services interact via *bindings*, which are less than connectors, without types, names or semantics. The behavioral view employs Finite State Processes[16] (FSP) to specify the behavior of components. As opposed to the construction view, no commitment is made as to the location of the implementation, the calling direction (requires or provides), or the data type of the communication. FSP is a textual notation of Labeled Transition Systems (LTS). Components communicate with each other by synchronizing on shared actions and composite components are constructed by parallel composition of LTS's. The communication semantics is thus similar to that used in CSP [Che94]. In the construction view, components are adorned with typed service interfaces that are either provided or required. The main purpose of this view is to allow code generation.

---

[13] More concretely, component interface ports are augmented with a parameter for clearance level. Constraints, relating calling and receiving clearance levels according to the policy, are stipulated.

[14] The Simple Security Property and the *-Property.

[15] In the construction/implementation/service view, components provide and require services at their interfaces and implementations are defined for the primitive components.

[16] A CSP-like notation [Mag97a].

```
component SENSOR {
   portal command; cancel; ack; sight;
inst TX; RX;
   bind
      command   -- TX.command;
      cancel    -- TX.cancel;
      ack       -- TX.ack;
      sight     -- RX.sight;
}
```

Figure 7. Sample component specified in Darwin [Mag97a].

Analysis of the construction view is susceptible to type checking. Thus, the Darwin compiler checks the compatibility of bound required and provided service interfaces [Mag97b]. As mentioned, behavior of Darwin systems is specified in LTS; it is accordingly analyzed using the Labeled Transition System Analyzer (LTSA). Specifically, certain safety[17] and liveliness[18] properties may be analyzed in the considered systems[19] by specifying the properties as automata (using LTS) and composing them with the system. Furthermore, exhaustive state space exploration may detect deadlocks and error states. Finally, specific test cases may be traced for manual evaluation.

6.3.4 METAH

Developed at Honeywell Technology Center, the MetaH architectural description language is focused on automatic development as well as architectural analysis [Ves98]. To maximize the analytic and code generating capabilities, the context in which the language and associated tool set may be employed is restricted. In particular, the preferred programming language is Ada (although there is support also for C), and the set of accepted components and connectors is pre-defined. Hard- and software entities are specified in the language, and they are used for analysis as well as deployment. Components include subprograms, processes, processors and devices, while ports and channels exemplify connectors. Entities are associated with predefined semantics and may be detailed using (predefined) properties. The above are entities in the structural view of MetaH; behavioral aspects maybe specified using *paths* for sequencing behavior, *events* for triggering purposes, and a number of entity attributes for other aspects (e.g. time between process dispatches, computation deadline, etc.).

The basic software components are programmed in Ada (or C) and must abide by certain rules for inclusion in the architecture[20]. The tool set consists of a syntactic analyzer, a

[17] Something bad will never happen

[18] Something good will eventually happen

[19] As for Wright, analysis requires finite state models while the notation allows infinite ones (using parameters/action subscripts).

[20] E.g. for the scheduling to function, all processes code modules must contain a master loop yielding control to the MetaH executive by calling the subroutine MetaH.Await_Dispatch.

hardware/software binder, a code generator and application builder, and a number of analyzers.

```
with type package DOMAIN_TYPES;
process P1 is
    FROM_P2 : in port DOMAIN_TYPES.INTEGER_TYPE;
    TO_P2 : out port DOMAIN_TYPES. INTEGER_TYPE;
end P1;

periodic process implementation P1.SIMPLE is
attributes
    self'SourceTime := 100 us;
    self'Period := 1 sec;
    self'SourceFile := "p1.a";
end P1.SIMPLE
```

Figure 8. Sample interface and implementation specified in MetaH [Ves98].

The analyses performed by the MetaH tool set include schedulability/performance, reliability and safety/security. The schedulability analysis of MetaH is based on so called *compute paths* and *source time* attributes. Paths define the sequencing behavior of entities. For instance, a thread of execution may sequence through a set of subprogram components in a process. The source time attributes specifies the execution time for the code in the code module. Together with additional information, such as the time between dispatches of processes and the execution time budget, the schedulability analysis calculates whether an application can be feasibly scheduled.

The reliability analysis proposed for MetaH is based on error models, i.e., models of component responses to errors and subsequent. Error models specify fault events and error states. Together with error arrival rates, propagation rates, error paths, and a number of other attributes, the error model constitutes the input to the reliability analysis. Results typically allow specification of mission duration and solve for the probability of being in each particular application error state [Ves98].

Finally, safety/security analysis is based on the assignment of safety levels and security classes. Briefly, an entity with a lower safety level should, according to the MetaH policy, not be allowed to affect the operation of any entities with higher safety levels. Similarly, entities may only receive information from other entities if they have the proper security classification. MetaH thus defines safety and security policies to which applications must abide (c.f. the SADL security analysis). The analysis per se operates as a semantic check, declaring whether an application is safe/secure or not.

6.3.5 RAPIDE

Rapide is by its inventors described as an event-based, concurrent and object-oriented language for system architectures [Luc95a]. As such, it is a combination of five sub-languages aimed at different concerns. The *types language* describes the component interfaces as data types, the *architecture language* describes the flow of events between components, the *specification language* specifies abstract constraints on component behavior, the *executable* *language* is used for module programming, and the *pattern language* describes patterns of events.

Architectures in Rapide consist of first-class interfaces and second-class connections. Defining types, interfaces contain typed functions and actions, constraints and behavior, and they are satisfied by (executable code) modules. Connections, belonging to the architecture language, bind different interface functions (synchronous) or actions (asynchronous) to each other.

```
architecture X/Open_Architecture()
   return X/Open is
      AP: Application;
      RM: Transaction_Manager;
      ...
connect
   AP.TX to TM.TX;
   ...
end architecture X/Open_Architecture;
```

Figure 9. Sample architecture specified in Rapide [Luc95a].

Behavioral aspects are specified using *posets* (partially ordered event sets), which are event sequences with both a temporal and a causal ordering. This allows a differentiation between “a caused b” and “a occurred before b”.

```
type Resource is interface
   public action Receive(Msg: String);
   extern action Results(Msg: String);
constraint
   match
      ((?S in String) (Receive(?S) ->Results(?S)))^(*~);
end Resource;
```

Figure 10. Sample interface type specified in Rapide [Luc95a].

The type language allows type checking. In particular, code modules written in the execution language may be checked for conformance to interfaces using static semantic analysis. Similarly, in an architecture, bound functions and actions may be type checked for compatibility in communication.

If modules are implemented in the executable language, run-time analysis (or simulation) is possible. Simulation in combination with conformance analysis allows architectures to be checked for interface and connector consistency as well as concurrency, resource and timing issues. Furthermore, conformance of modules to interfaces and architectural (communication) constraints may be checked. According to [Luc95a], code generation in C++, Verilog and Ada constitute further research. Whether modules implemented in these languages will be checkable by the above means is not clear.

The assessment of the extra-functional property *atomicity* is considered in [Luc95a]. Atomicity is related to distributed transactions and means that after a transaction executes, either all or none of its operations take effect. Similar to SADL's treatment of security, Rapide's assessment of atomicity is performed by specifying a system type that guarantees the property and checking the conformance of instances to the type. In [Luc95a], the X/Open distributed transaction processing (DTP) standard is specified and, coarsely, if an architecture conforms to the standard, then it preserves transaction atomicity (in the SADL case, the Bell-LaPadula multi-level security policy is employed). This approach is, of course, in many ways unsatisfactory, since it rather specifies a solution than a requirement. Many good architectures would not pass the atomicity check of [Luc95a] or the security check of [Mor95].

6.3.6 UniCon

UniCon, developed by the Carnegie-Mellon University, is primarily concerned with glue generation for integration of existing components [Sha95b] [Sha96b] [DeL99] [Zel96]. The view supported by the language is primarily the structural view, specifying components, connectors and their relations. As most of the CMU ADLs, connectors as well as components constitute first class entities, with types, names, etc. Connectors (defined by their *protocols*) have roles and the component ports are called players. All entities are adored with properties. In the current version, UniCon provides a fixed set of component types. The semantics of these are to a certain extent implicit in the UniCon tool set implementation [Abd96]. As such, the UniCon language is tightly linked to the tool set and underlying platform.

```
component Real_Time_System
   interface is
      type General
   end interface

   implementation is
      uses client interface rtclient
         PRIORITY(10)
         ENTRYPOINT (CLIENT)
         end client
   ...
   establish RTM-realtime-sched with
      client.application1 as load
      client.application2 as load
      server.services as load
      ALGORITHM (rate_monotonic)
      PROCESSOR ("TESTBED.XX:EDU")
      ...
      end RTM-real-time-sched
   ...
end Real_Time_System
```

Figure 11. Sample component specified in UniCon [Sha95b].

The (explicit) non-structural information is specified in properties of components, connectors, players and roles. Properties are either required or optional. Required properties are those necessary for glue generation, while optional may include any information of interest to for instance architectural analysis. The basic architectural analysis is thus primarily based on the structural view, allowing e.g. type checking and (limited) compatibility checking. However, any additional analyses may be performed if their required information is expressible in the form of properties. As such, two analyses are considered here. Firstly, real-time schedulability analysis has been implemented according to a set of techniques developed at the Software Engineering Institute, called rate monotonic analysis (RMA) [Kle91]. Secondly, performance analysis has been implemented using queuing network theory [Spi98].

```
connector RTM-realtime-sched
   protocol is
       type RTScheduler
       role load is load
   end protocol

   implementation is builtin
   end implementation
end RTM-realtime-sched
```

Figure 12. Sample connector specified in UniCon [Sha95b].

Real-time schedulability analysis has been implemented for UniCon specifications according to a set of techniques developed at the Software Engineering Institute, called rate monotonic analysis (RMA). Input to the analysis includes execution times, periods, relative priorities, and event tracks. The analysis predicts whether all processes will meet their deadlines or not. The analysis assumes pre-emptive, fixed-priority systems, and the UniCon tool set is restricted to the Real-Time Mach operating system [Tok90]. The allowed UniCon components are *processes*.

6.3.7 AESOP

Aesop [Gar94b] [Gar95b] is more a development environment than an architectural description language. However, developed by the CMU, Aesop shares several of the features of Wright and other CMU languages (e.g. ACME). Basic entities are components with ports, connectors with roles, configurations, and representations and bindings. In the Aesop system, these generic entities are sub-typed to form specific architectural styles. The Aesop system is currently not under development as focus of the CMU people has shifted to ACME. Aesop does provide several analysis tools, including type checkers and schedulability analyzers. Here, the presentation is restricted to the performance analysis of Spitznagel et al. [Spi98].

Performance analysis, based on queuing network theory, of architectures described in the Aesop ADL is proposed by Spitznagel and Garlan [Spi98]. Queuing network theory is based on queues (or buffers) and service centers. [Spi98] presents an initial adaptation of the theory to the context of software architecture. Input to the analysis is job service time (how long it takes for a component to complete a job) and time between job arrivals distributions for each component. Output includes component utilization, queue length, latency, throughput, and more. The presented example is only applied to the distributed message passing style, which is similar to queuing networks; extension of the theory to other styles constitutes further work. Furthermore, in the current state, there is no differentiation between types of messages (or jobs). Also, processes are considered as primitive components, disallowing concurrent processing within one component.

6.3.8 WRIGHT

Wright [All94b] [All97] [All98] is concerned with the static structure of software components as well as their interactions. In a Wright specification, a system consists of components using connectors for communication. The behavior of the system entities is described as processes and the interactions between entities thereby becomes the interactions of communicating processes. Fortunately, a well-known formalism for describing communicating processes already exists, suitably named Communicating Sequential Processes (CSP) [Hoa85]. CSP is thus incorporated in Wright for the description of behavior.

Component Server-type
Port Client = $\text{DefineSet}_{\text{oa, ca}}$
  Port Admin = $\text{DefineSet}_{\text{na, ra}}$
  Computation = $\text{State}_{\text{NoAccount}}$
    where $\text{State}_{\text{NoAccount}} = (\text{DSST}_{\text{na, AnAccount}}$ □ DUST_{ra} □ DUST_{oa} □ DUST_{ca}) □ §
    $\text{State}_{\text{AnAccount}} = (\text{DUST}_{\text{na}}$ □ $\text{DSST}_{\text{ra, NoAccount}}$ □ $\text{DSST}_{\text{oa, OpAccount}}$ □ DUST_{ca}) □ §
    $\text{State}_{\text{OpAccount}} = (\text{DUST}_{\text{na}}$ □ DUST_{ra} □ DUST_{oa} □ $\text{DSST}_{\text{ca, AnAccount}}$) □ §

Figure 13. Sample component type specified in Wright [Eks02].

Components contain a computation and one or several ports (interfaces), while connectors contain one or more roles and glue. Computations describe the component behavior, ports describe the components externally visible behavior and assumptions about the environment, roles are connected to ports, and glue specifies the relations between roles. Computations, ports, roles and glue are all viewed as processes and specified in a slightly modified CSP. A CSP process is a sequence of observed and initiated events and external or internal choices. CSP also supports states.

Connector MethodInvocation(E : $\mathbb{P}\,\Sigma$)
  Role Definer = $\text{DefineSet}_{\text{E}}$
  Role User = UseSet_{E}
  Glue = $\text{NameMatch}_{\text{Definer, User}}$

Figure 14. Sample connector type specified in Wright [Eks02].

Wright specifications may be analyzed by performing a number of generic tests. These include port/computation consistency, port/role compatibility, connector deadlock-freedom[21], initiator commits, single initiator. Briefly, a port is consistent with a computation if it allows the computation to initiate the events that it might want to initiate. A computation is consistent with a port if it always is prepared to observe those events that the port may observe. In other words, *port/computation consistency* ensures that the computation won't do anything that the port won't allow, and that the port won't allow anything from the environment that the computation can't do. *Role/port compatibility* is similar to the port/computation consistency in that it ensures that the role only initiates those events that the port can accept and that the port only initiates those events that the role can accept. *Connector-deadlock-freedom* guarantees that the roles always will agree on the next event. *Initiator commits* ensures that a process that initiates an event does not simultaneously allow the environment to affect its execution. *Single initiator*, finally, guarantees that an event in a connector is initiated by a single role (all other roles engaged in the event need to be observers).

6.3.9 ABD-ALLAH AND GACEK

Although not viewed as an architectural description language in its own right, the Z-based [Spi89] formalism of Abd-Allah [Abd96] and Gacek [Gac98] is used to specify and analyze architectures. Architectures according to Abd-Allah and Gacek are composed of control and data components, control and data connectors, component ports, triggers, objects and systems. These base entities are further specified by a set of attributes and constraints, including their data types, network nodes, and resource usage, arguments, buffer sizes, platforms, class belongings, and a number of constraints.

The work is primarily concerned with detecting "architectural mismatches". Architectural mismatch is defined as "logical inconsistencies between constraints of different architectures being composed," which (coarsely interpreted) are the integration problems that can be detected by means of architectural style analysis. At the basis of the mismatch detection scheme lays the concepts of architectural styles. An architectural style defines a family of systems based on a common structural organization [Sha96a]. Typical architectural styles are for instance implicit invocation, pipes-and-filters, black-board, interpreter, main-program-and-subroutine, and layered architectural styles [Bas98]. Abd-Allah and Gacek attempt to characterize some common architectural styles using what they call *conceptual features*. Conceptual features are considered as more fundamental constructs than architectural styles and are exemplified by the following list: concurrency, distribution, encapsulation, layering, triggering capability, and preemption.

[21] The definition of deadlock in [All97] and [Hoa85] is "when participants in an interaction cannot agree on the next appropriate event" or differently, a "process is said to deadlock when it may refuse to participate in all events, but has not yet terminated successfully." This is broader than the traditional definition of deadlock as a state where actors sharing the same resources are mutually preventing each other's resource access. The deadlock tests suggested by [All97] include port/computation consistency and port/role compatibility.

Before introducing the actual architecture mismatch detection scheme, it is necessary to present the different types of component interactions that are treated by the authors. According to Abd-Allah, components and systems may have *bridging connectors* of the following kinds: call, spawn, data connector, shared data, triggered call, triggered spawn, triggered data transfer, shared resources, statically declare, dynamically declare, import, export. Of these, Gacek and Abd-Allah are only concerned with *call, spawn, data connector, shared data, trigger,* and *shared resources*.

Architectural mismatch can arise when two systems with certain conceptual features are joined with a specific bridging connector. For instance, two *concurrent* (conceptual feature) systems *share data* (bridging connector), with potential synchronization problems. Or, a *trigger* (bridging connector) refers to a system which forbids explicit or implicit *data connectors* (conceptual feature), hence triggering may never occur. The list of architectural mismatches detected by this scheme consists of 46 possible errors [Gac98].

6.4 DEDUCTION-BASED INTEGRABILITY ANALYSIS

In this section, the above described deduction-based architectural analysis approaches are considered explicitly in the light of integrability. In practice, the approaches are evaluated for their capabilities of predicting potential integration problems. The integration issues elicited in Chapter 3 are employed as a base for the evaluation. The main question considered in Chapter 3 for each technology was whether the platform/technology or the developer was responsible for the issue, or if it was managed jointly. In this evaluation, we ask whether the reviewed architectural analysis methods are capable of detecting if the issue is managed or not in a given architectural specification. As an example, a data representation issue managed by an application integration adapter is component operation signature transformation, e.g. by renaming a procedure call into the expected name. If given an architectural description of a component requesting an operation with one name and another providing an operation with another in the C2SADEL language, the associated type checking mechanism (as implemented in DRADEL [Med99]) will generate a warning. The results of the evaluation are compiled in Table 5 and Table 6 at the end of this section.

6.4.1 DATA REPRESENTATION

Data representation refers to syntactic issues. In the context of integration analysis, this particularly concerns data representation compatibility. A complete analysis of the data representation issue captures all syntax incompatibilities between components to be integrated. Data representation is typically an issue on several layers of abstraction. In the (remote) procedure call, number representation may be big- or little-endian. On top of this, a procedure call may require that variables are separated by commas and enclosed within parentheses. On the next level, the developer may specify that the parameter *age* should come before the parameter *beauty*. If the communication takes place over a network, further data representation issues are introduced in the network protocol. All of these representations must match between procedure caller and definer. Furthermore, higher-level data representations need to be considered, including data records and schemas. In most examples, only one level of data representation (e.g. procedure signatures) is specified, whereas the underlying levels (e.g. network protocol) are left unconsidered.

Several architectural description languages address data representation issues by typing and type checking data. Typically, the type of the data of ports, roles and connectors may be specified and checked. C2Sadel specifications, for instance, include typed provided and requested procedure signatures with typed parameters and return variables which, when bound, may be checked for type compatibility. Normally, the types are simply defined by tags (e.g. "integer" or "char"). Code generating environments, such as the MetaH toolset, require the specification of types in the underlying programming languages, thus relying on their typing facilities for allowing the definition of complex data structures. Excluding associated programming languages, the definition and checking of complex data structures is not possible.

Languages tightly linked to a development environment may allow only certain types of interactions, thereby limiting the possible data exchange formats. For instance, C2Sadel is based on a specific architectural style, and therefore limits these exchange options (as styles are intended to do). Some languages, such as Wright, do not type data. Of course, in these languages, data representation compatibility cannot be checked.

6.4.2 DATA SEMANTICS

Data semantics refers to the meaning of data. If the data semantics issue is not managed, the data may be readable, but it will be interpreted incorrectly. Data semantics compatibility checking should thus detect whether the data will be interpreted in the same manner in the both the providing and receiving ends. This issue cannot be managed completely, as the possibilities for misunderstandings are endless. Data typing does however provide a rudimentary support. By defining the types of data allowed in different operations, the risks for misinterpretations are arguably reduced. Meta-data (c.f. XML), describing and relating data items to each other, provides further support.

As mentioned, in many languages, communicated data is typed. No languages do, however, provide constructs for more extensive meta-data specification or compatibility checking. Some languages (e.g. Rapide) allow the specification of data manipulation (primarily routing). While this does allow certain reasoning about a component's or system's interpretation of the data, it is difficult to imagine automation of this kind of analysis.

In general, data compatibility is not considered the primary domain of software architecture, so it is not remarkable that the subject is only superficially treated. However, in this text, the concern becomes relevant since we are interested in integration issues.

6.4.3 *Connector Semantics*

Connector semantics refers to the behavior of connectors, i.e., synchronization and sequencing of interactions between components. Typical results of failure to manage connector semantics include deadlock and starvation. Connector semantics are, as data representation issues, defined on several layers. For instance, in a remote procedure call, the platform ensures that the server responds to the client. The remote call may, however, on a lower level be communicated with TCP/IP, which in itself contains and manages a number of synchronization issues between peers. On top of the remote procedure call, the developers may have implemented additional sequencing or synchronization rules. Analyses of connector semantics should detect sequence incompatibilities between ports (and in applicable cases between ports and roles, and between roles within connectors) as well as system-level effects such as deadlock.

There is an often-spoken-of difference between those languages that support explicit connectors and those that do not. This distinction does, however, not seem to influence the types of connector-semantic analyses possible. Detailed modeling of concurrent processes is supported by several languages (e.g. Darwin, Rapide, Wright) and well-known formalisms are available (e.g. process algebras and finite state automata), therefore many common issues related to timing and synchronization may be analyzed, including safety properties (such as deadlock-freedom), liveliness properties (such as starvation- and livelock-freedom), and architecture-specific constraints. C2Sadel employs pre- and postconditions, which also may be employed to specify connector semantics [Med99]. For these languages, matching of processes according to different systems allows for more or less relaxed compatibility checks.

Surprisingly often, the level of abstraction in the behavioral descriptions is low. The employed abstractions are completely based on existing abstractions (e.g. the call sequence of a remote procedure call, hiding the underlying sequencing in e.g. the network protocol), which are typically on the programming language level.

6.4.4 *Component Semantics*

Component semantics refer to the behavior of components. If the component semantics issue is not managed, a component invoked by another will not behave as expected by the invoker, even though the invocation was syntactically impeccable. There is a close relationship between component semantics and connector semantics, since the external behavior of components is manifested in its interfaces, which in turn are bound to connectors. Connector semantics, however, is not concerned with the internal workings of components, or potential side-effects (when the component influences the environment, e.g. by printing to a screen, launching a rocket, etc.). These internal states or side-effects may however be primary criteria for integration, (as, presumably, is the case for a call to the "launch_rocket_procedure").

Several of the reviewed ADLs consider component semantics. C2Sadel matches pre- and postconditions of requested and provided services. This approach allows full component semantic analysis, since the requestor describes the results it is seeking. Wright, Rapide and Darwin check that process algebra component port specifications match. This approach thus concerns the part of component semantics that is in common with the connector semantics, and ensures that those aspects of a components behavior that are visible in the interface are the requested ones. Internal states and other interfaces[22] are, However, also represented in different manners in these languages and could be used as a basis for connection constraints.

6.4.5 ERROR CONTROL

Error control refers to the mitigation of undesired behavior. If error control is not implemented, everything will work fine under optimal circumstances, but once a disturbance is introduced, the system execution is in danger. Architectural analysis of errors is normally performed as reliability analysis. The purpose of the analysis is to predict how the system will react to some set of (expected) errors. Since many things in life may go wrong in many ways, error control cannot be completely managed. Also, error control and reliability are closely related.

Darwin and MetaH provide specific constructs for error control. Darwin does this by providing special error states, while MetaH defines error models, describing error states, error state transitions and error propagation. MetaH further allows for the analysis of the error model by introduction of fault event probability distributions (cf. Reliability below). In other languages providing behavioral modeling, such as Wright, error control is not subjected to special treatment, but error states may be defined by the user.

6.4.6 LOCATION

Location refers to the identification, location, addressing and routing of communicating parties. If the location issue is not managed, a message sent by one party might reach some recipient, but not the intended. Analysis of location issues should detect whether there is a recipient at all, and if so, whether this recipient is the intended. In the case of routing, the analysis would need to consider several consecutive senders and recipients.

A major benefit of software architecture is the explicit consideration of component relations. As such, architectural descriptions highlight location issues. It is easy to detect if a component port is bound or not to a connector. Component semantic analysis may further aid in determining whether the communicating party is the intended one. However, a problem of several languages is their static nature. Component semantics are specified and checked in advance, but if a component run-time attempts to find and interact with a new

[22] It is an issue of system boundaries whether a (physical) rocket launch should be considered an internal component state, a side-effect, or an event in the components "rocket" interface.

component (as is the purpose of web services), this cannot always be described. Darwin, Rapide, and Wright (in a revised version, cf. [All98]) supports constrained dynamism, which briefly means that all potential interactions must be known in advance. C2Sadel supports unconstrained dynamism.

Furthermore, none of the languages consider addressing issues. In dynamic systems, components may find addresses to other components via directory services, they employ brokers or routers for relaying, etc. Possibly, these solutions could be considered as architectural styles, and much in the way that security properties are analyzed in SADL (below), these location issues might be specifically modeled and analyzed for different solutions. This has, however, not been done. Finally, none of the languages explicitly consider the availability of communicating parties.

6.4.7 EXTRA-FUCTIONAL PROPERTIES

Extra-functional properties, or quality attributes, refer to an array of "ilities" that often need explicit consideration in software integration projects. These include security, data consistency, performance, reliability, and more. Extra-functional properties are in the practical case often tightly linked to functionality. For instance, reliability is enhanced with mechanisms for error control and performance is increased with load balancing and connection pooling. The set considered herein is determined by the individual properties prominence in the reviewed literature.

PERFORMANCE

Performance analysis, based on queuing network theory, of architectures described in the Aesop ADL is proposed by Spitznagel and Garlan [Spi98]. Queuing network theory is based on queues (or buffers) and service centers. Spitznagel and Garlan present an initial adaptation of the theory to the context of software architecture. Input to the analysis is job service time (how long it takes for a component to complete a job) and time between job arrivals distributions for each component. Output includes component utilization, queue length, latency, throughput, and more. The presented example is only applied to the distributed message passing style, which is similar in structure to queuing networks; extension of the theory to other styles constitutes further work. Furthermore, in the current state, there is no differentiation between types of messages or types of jobs. A similar approach to performance analysis using queuing network theory is presented in [Aqu01].

Closely related to performance is real-time schedulability analysis, which has been implemented for UniCon specifications according to a set of techniques developed at the Software Engineering Institute, called rate monotonic analysis (RMA) [Kle91]. Input to the analysis includes execution times, periods, relative priorities, and event tracks. The analysis predicts whether all processes will meet specified deadlines or not. The analysis assumes pre-emptive, fixed-priority systems, and the UniCon tool set is restricted to the Real-Time Mach operating system [Tok90]. The allowed UniCon components are *processes*.

MetaH also features a schedulability analysis, based on so-called compute paths and source time attributes. Paths define the sequencing behavior of entities. For instance, a thread of execution may sequence through a set of subprogram components in a process. The source time attributes specifies the execution time for the code in the code module. Together with additional information, such as the time between dispatches of processes and the execution time budget, the schedulability analysis calculates whether an application can be feasibly scheduled.

CONCURRENCY ISSUES

Several languages contain sub-languages for describing behavior. Darwin uses labeled transition systems; Rapide and Wright employ process algebras. These formalisms are not new, and have been explored in depth prior to the dawn of software architecture (cf. e.g. [Hoa85]). Properties such as safety (including deadlock-freedom) and liveliness (e.g. livelock and starvation) are typically possible to assess for finite state models. Rapide also incorporates clock time as an explicit parameter, and supports simulation, to allow analysis of timing issues.

RELIABILITY

The reliability analysis proposed for MetaH is based on error models, i.e., models of component responses to errors and subsequent state transitions. Error models specify fault events and error states. Together with probability distributions of the error arrival rates, propagation rates, error paths, and a number of other attributes, the error model constitutes the input to the reliability analysis. Results typically allow specification of mission duration and solve for the probability of being in each particular application error state [Ves98].

SECURITY

The security analysis of SADL performed by proving that the Bell-LaPadula [Mor97a] multilevel security (MLS) policy is not violated. Briefly, the policy allows read and write access only to subjects with the proper clearance level for a specific resource; all other data flows in the architecture are forbidden. The analysis is fundamentally different from the previously described. It is based on the idea of matching specific security mechanisms (i.e. the Bell-LaPadula policy) to the general security property, thus decomposing the main property into simpler and verifiable architectural properties. In this specific example, component interface ports are augmented with a parameter signifying clearance level. Constraints, relating calling and receiving clearance levels according to the policy, are then stipulated and their conformance is subsequently proven [Sta01]. The Bell-LaPadula policy is presented as two verifiable properties[23].

[23] The Simple Security Property and the *-Property.

An important limitation of this work is that the Bell-LaPadula policy is not a definition of security, but an example. Thus, an implementation of the Bell-LaPadula policy may be deemed secure, but employing solely Bell-LaPadula as evaluation criterion for the security property of unknown systems will undoubtedly rule out many perfectly secure architectures.

Furthermore (and related), whether the Bell-LaPadula policy is in fact enforced in an implemented system does of course depend on a number of unconsidered factors, including *how* access is to be prohibited in components and connectors.

ATOMICITY

The assessment of the extra-functional property *atomicity* with Rapide is considered in [Luc95a]. Atomicity is related to distributed transactions and means that after a transaction executes, either all or none of its operations take effect. Similar to SADL's treatment of security, Rapide's assessment of atomicity is performed by specifying a system type that guarantees the property and checking the conformance of instances to the type. In [Luc95a], the X/Open distributed transaction processing (DTP) standard is specified and, coarsely, if an architecture conforms to the standard, then it preserves transaction atomicity (in the SADL case, the Bell-LaPadula multi-level security policy is employed). This approach is, of course, in many ways unsatisfactory, since it rather specifies a solution than a requirement. Many good architectures would not pass the atomicity check of [Luc95a] or the security check of [Mor95].

A compilation of the above is presented in Table 5 and Table 6.

	C2	**Wright**	**SADL**	**Darwin**	**MetaH**	**Rapide**	**UniCon**	**Abd-Allah & Gacek**	**Aesop**
Data representation	Communicated data is typed. Employs type checking, matching providing and requesting component names, operation names, and parameter names and types. Limits ports to procedure calls. No possibility to specify or check data structures such as schemas.	Passed data is not typed. No possibility to specify or check data structures such as schemas.	Port parameters (passed data) also have data types. Allows port type compatibility checking. Seems to limit ports to procedure calls? No possibility to specify or check data structures.	Service interface data type is specified. Allows service (data) type compatibility checking. No possibility to specify or check data structures.	Ports are typed in underlying programming language (Ada). As such, data structures may be declared as e.g. records.	Functions and actions are typed. Data either as function or action parameters. No possibility to specify or check data structures.	Some players and roles specify allowed data type in the form of arguments. Some do not. No possibility to specify or check data structures.	Ports, data components and data connectors specify allowed data types. These could be, but are not, checked. Data structures could be, but are not, specified or checked.	Style-specific.
Data semantics	Variables and parameters have predefined types. Developer is responsible for additional semantics.	No data types. To a small extent, the transport of data may implicitly be specified in component semantics.	Port and connector parameters have data types. Component semantics and system constraints may implicitly further define data semantics (e.g. security clearance), and analyzed for conformance.	Service interfaces have data types. Component semantics may implicitly define data semantics.	Ports are typed in underlying programming language (Ada). Data does not appear in component semantics (except in programming code).	Function and action parameters are typed. Data semantics is further implicitly defined by component (interface) behavior and constraints.	Players and roles specify allowed data types.	Ports, data components and data connectors specify allowed data types.	Style-specific.
Connector semantics	To small extent implicit in component semantics (pre and postconditions could be employed to specify operation invocation sequencing.	Ports and connectors have operational semantics. Employs compatibility checks on port, role and glue specifications,	Connectors have no semantics	No connectors, but interaction semantics is defined using parallel composition of LTS's with shared actions.	Not implicit in component semantics. Only present in underlying programming language specification of components.	Implicit in interface definitions. Interfaces are compatibility checked both statically and run-time.	Connectors, ports and roles have some (sometimes implicit) predefined semantics. Any additional may be added as properties. Predefined composition rules for the available entities that may be checked.	No behavioral. Style and type restrictions.	Style-specific (CSP suggested for pipe-and-filter).
Component semantics	Employs type checking, matching pre- and postconditions of operation specifications.	CSP component port specification must match connector role specification.	Specified in logic.	Specified in LTS or π-calculus.	Paths, events and several predefined properties.	Specified using posets as well as execution language.	Only as property list (possibly uninterpreted). Specifically for schedulability analysis.	No behavioral. Style and type restrictions.	Style-specific and implicitly specified in corresponding Aesop class.

Table 5. Integration issues addressed by reviewed analyses.

	C2	Wright	SADL	Darwin	MetaH	Rapide	UniCon	Abd-Allah & Gacek	Aesop
Error control	No explicit provisions.	No explicit provisions.	No explicit provisions.	Explicit error states allowing state exploration for checking.	Explicit modeling of error states and error propagation.	No explicit provisions.	No explicit provisions.	No explicit provisions.	No explicit provisions.
Location	"Unanticipated" dynamism.	Checks style conformance (typically includes correct component-connector attachments) Constrained dynamism.	Allows style conformance checks (typically includes correct component-connector attachments) Constrained dynamism.	Checks correct provider-requester interface bindings. A Darwin specification declares how components are bound, and these bindings are automatically generated into the code. Provisions for constrained dynamic architectures (all runtime changes must be known a priori).	No dynamism.	Constrained dynamic architectures (all runtime changes must be known a priori).	No dynamism.	No dynamism.	Checks style conformance. No dynamism.
Extra-functional properties	-	Deadlock.	Certain support for security analysis. Potential (limited) future support for intrusion tolerance.	Safety and liveliness by including property automata. Deadlock and error states by exhaustive state space exploration.	Reliability analysis, safety/security analysis, and schedulability analysis.	-	Schedulability analysis.	Architectural mismatch detection.	Performance analysis.
Critical assumptions	Connectors are (specifiable as) procedure calls. Detail is available.	Low-level detail is available.	Complete formal specifications of (COTS) components (detail is available).	Tightly bound to development environment. Low-level detail is available.	Tightly linked to program-ming in Ada or C. Full access to programming code. Low-level detail is available.	Module specifications in execution language for several analyses. Low-level detail is available.	RT Mach operating system, code generation according to UniCon system. Only entities as coded by system. Detail is available.	Systems are reducible to conceptual features. Connections are reducible to connectors. Certain detail is available.	Style-specific.

Table 6. Integration issues addressed by reviewed analyses (continued).

6.5 Applicability to Enterprise Software Systems

Obviously, some issues are analyzable using existing formalisms and methods, while others are not. This section summarizes the findings and considers them in the light of enterprise software systems. In particular, the section considers 1) large components, 2) unmodifiable black-box components, and 3) heterogeneous (independently developed) components and connectors.

The main method for analysis of *data representation* issues is by type checking. The checks rely on either the assumption of a common understanding of data types between interacting components, or checks of explicit type specifications. The reviewed approaches rely on the assumption of a common understanding, which may be viable for small systems directly generated from the architectural specification. In the context of enterprise software systems, where components typically are independently developed, it is, however, unlikely that this assumption is valid. Therefore, explicit type specifications checks seem necessary. However, due to the size and black-box nature of an enterprise software system, the information-gathering task associated with such an undertaking is liable to become extremely cumbersome. This presumably inherent trade-off problem between analysis reliability and effort makes the type checking approach to the data representation issue, which in principle is workable, of limited applicability in practice.

The *data semantic* issue is, broadly speaking, not supported by the reviewed approaches. As mentioned, these kinds of analyses may, for instance, be undertaken based on meta-data models. In the reviewed work, only the semantics implicit in the data representation is checkable. In enterprise software systems, these issues are often of great importance due to the heterogeneity of components. Different system developers as well as system users often define data in slightly different ways, making it important to understand the intended semantics of the different entities. A situation where these problems are clearly manifested is the integration of databases from different vendors and users.

The *connector semantics* issue is perhaps the most considered issue in the reviewed approaches (e.g. Rapide, Darwin and Wright). In principle, these approaches are appropriate for analyzing connector semantics. However, with current levels of abstraction (programming language levels) much detailed information is required for each component and connector. Moreover, for independently specified components and connectors, an important task is the mapping between port and role processes. It is probable that the communication event *a* in a component port is called *b* in its associated connector role. These symbols, denoting the same event, need to be bound to each other. Furthermore, as with data representation, there is often a layering of communication protocols, and a trade-off exists between the depth of the analyzed interaction and the effort spent on information gathering. In comparison to the data representation issue, however, interaction protocols are more standardized, thereby in principle allowing better reuse of their formalizations (although there is little evidence of such reuse in current practice).

The *component semantics* issue is considered by several approaches. In particular, C2Sadel employs pre- and postcondition matching between requesting and providing components. In addition to the information required for the connector semantics, internal states and side effects also need to be specified and understood by both components. As for connector semantics, the analysis effort grows rapidly with the complexity of the system (in particular system size, heterogeneity, and information unreliability).

The *error control* issue is explicitly considered by two of the reviewed approaches and is implicitly allowed by several others. Developing error models is not significantly different from developing component or connector semantics models. It is possible in principle, but cumbersome in practice.

The *location issue* is not explicitly considered by the reviewed approaches. Firstly, most architecture description languages are static to their nature. For interesting location concerns to be considered, dynamic architectures are necessary. Although several languages support a constrained type of dynamism, only C2Sadel allows unanticipated dynamism. For these languages, no location analyses have been attempted, although technology-specific ones are imaginable. In the context of enterprise software systems, the location issue may be the one least affected by the size of the components, since their data or behavioral complexities are unrelated to the concern.

Extra-functional properties are analyzable to a varying degree. The reviewed literature considers analyses of performance (including schedulability), concurrency issues (e.g. safety and liveliness properties), reliability, security, and atomicity. Firstly, many important properties are not considered, including data and functional consistency, modifiability, isolation, and durability. Secondly, several of the analyzed properties are considered in a restricted context: the security analysis is only applicable to Bell-LaPadula-compliant systems, the atomicity analysis is only applicable to X/Open-compliant systems, the schedulability analyses are only applicable to real-time, fixed priority, pre-emptive platforms. Nearly all of the considered property analyses require models of a detail that may be hard to attain in the context of enterprise software systems. Finally, many of the analyses rely on assumptions that hardly are fulfilled by the enterprise software system, e.g. real-time, preemptive scheduling.

As an endnote, one of the most difficult problems when applying deduction-based architectural analysis methods to enterprise systems is to find a reasonable level of abstraction. Even for traditional systems, this is a trade-off between viability of the assumptions on which the abstractions are based and effort spent in information gathering (as discussed elsewhere in this thesis, few assumptions are unquestionable in an artificial world). In the case of enterprise software systems, these difficulties increase. Many assumptions of homogeneity, reasonable in the context of traditional systems, are not credible in enterprise software systems, due to the independence of component and connector development.

Furthermore, components are comparatively large. If the level of detail even comes close to those considered in the reviewed approaches, the amount of information and work necessary for a reasonable analysis will undoubtedly require collaboration between developers and users, agreements on specification standards, and completely new processes for software management.

Chapter 7
Summaries of Included Papers

PAPER A:

IT Infrastructure Architectures for Electric Utilities: A Comparative Analysis of Description Techniques

Jonas Andersson and Pontus Johnson

In Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, 2000.

This paper was our first explicit attempt to apply architectural concepts to enterprise software system concerns. The paper proposes explicit modeling of enterprise software architectures employing established modeling notations as an approach for the management of the enterprise software system. In an exploratory case study of four companies in the Swedish electricity market, three problematic characteristics resulting from the integration of software systems in electric utilities are identified: overlapping data, overlapping functionality, and interfaces and connectors.

The suitability of a number of common modeling notations, including class diagrams, entity diagrams, entity-relationship diagrams, Jackson system development and deployment diagrams are evaluated for modeling the three problem areas. The results of the evaluation indicate that several notations may be employed for modeling enterprise software systems, but they lack explicit consideration of certain aspects highlighted in the article. For instance, it is noted that explicit modeling of connectors is desirable but poorly supported by the notations. Additionally, the paper stresses the need for an explicit software design process in software-owning organizations such as electric utilities. A section on the enterprise software architecture management process is also included in order to set the context.

Summarizing, the paper empirically identifies three architectural problem areas, and proposes and evaluates modeling notations as a means for managing the problems.

PAPER B:

Extending Attribute-based Architectural Analysis to Enterprise Software Systems

Jonas Andersson and Pontus Johnson

In Proceedings of the Third Australasian Workshop on Software and Systems Architecture, 2000.

This paper explores the applicability of traditional architectural analysis processes on enterprise software systems. The central part of the paper elaborates on the issues that distinguish analysis of enterprise software systems from traditional software systems and describes a proposed modified analysis process in generic terms. The modified process is subsequently presented in the context of an acquisition project in a mid-sized Swedish electric utility. Enterprise software systems are distinguished from traditional systems in several areas. These differences are also considered in Paper D.

The enterprise software architecture adaptations of the traditional analysis process are summarized in seven points: 1) harsher constraints on the design space caused by legacy systems, usage of COTS components, and limited availability and modifiability of components necessitates a higher focus on the *constraints and context description,* 2) heterogeneity of components and middleware makes integration a prime design issue during *architectural representation,* 3) limited availability and modifiability of COTS components results in discrete design alternatives during *architectural representation,* 4) higher architectural abstractions containing less details may result in higher uncertainty of *analysis* results, 5) system properties not encountered in traditional software systems may require new attribute *analyses*, such as identification of overlapping data and functionality, 6) dependence on software vendor organizations shifts some of the focus from technological to organizational and contractual issues, that to some extent also may be subject to architectural *analysis*, considering issues such as upgradeability, modifiability of components, etc., 7) complex components with an abundance of interfaces results in high abstractions and high selectivity in the information contained in architectural descriptions throughout the process.

PAPER C:

Exploring Architectural Analysis Credibility from a Developer Perspective

Mathias Ekstedt and Pontus Johnson

In Proceedings of the Fourth Australasian Workshop on Software and Systems Architecture, 2002.

This paper explores the credibility of deduction-based analysis methods when assumptions of correct inter-specification transformations are relaxed. Recognizing the limited success of formal methods for program transformation and architectural refinement, the article considers how informally devised transformations between specification languages may invalidate the results of architectural analyses. It is assumed that the use of deduction-based analysis methods may be justified also in cases where the relation between the architectural description, the underlying specifications and the implemented system is not formally proven. However, when this relation is not ensured, transformations between specifications may become subject to distortions. The paper considers a number of transformation distortions, their effects, and explores to what extent existing development tools, such as compilers and analyzers, may be employed to validate transformations.

An included example is based on a simple system specified in UML and is composed of several steps. Firstly, the usefulness of architectural analysis with informally devised transformations is demonstrated by a transformation of the UML specification to the architecture description language Wright and a subsequent analysis. Secondly, the employed informal transformation is demonstrated applicable only to a limited set of specifications. Thirdly, it is argued that the set of specifications for which an informal transformation is applicable is generally unknown. Fourthly, the propagation of transformation distortions is exemplified by a transformation to C++. Fifthly, transformation distortions introduced by black-box components are exemplified.

Consequences of the findings include a need for harmonization of software abstractions used in the same development project, including component technologies, programming languages and modeling languages. Component certification is suggested as a means for increasing the trust in the architectural specification, and thereby the architectural analysis. Another suggested means for increasing the trustworthiness of the architectural analysis is certification or other kinds of validation of automated transformers, such as code generators or compilers. Finally, assessment of the input data sensitivity of architectural analysis methods is suggested as a way to increase the credibility of the analysis results.

PAPER D:

Architectural Integration Styles for Large-Scale Enterprise Software Systems

Jonas Andersson and Pontus Johnson

In Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference, 2001.

This paper considers induction-based analysis and design using architectural integration styles, i.e., architectural styles describing software structures of integration solutions for enterprise software systems. The article proposes an approach for selection of integration solutions based on these integration styles.

Enterprise software systems are distinguished from systems traditionally considered by the architecture community by seven points: 1) components are large-grained, 2) the supply of packaged components is limited, 3) the legacy architecture constitutes the starting point of the system development effort, 4) the enterprise software system may contain both data and functional redundancy, 5) components are not modifiable, 6) components are heterogeneous, 7) connectors are heterogeneous. These differences constitute a base for the use of architectural styles on the enterprise level.

A number of styles are presented, including the mediator styles, the gateway style, the desktop integration style, the message router style, the database federation style, the point-to-point style, and the adapter style. The styles are defined, the quality attributes they are considered to impact are described, and requirements on the involved components are presented.

An example of the use of architectural integration styles as a means for designing integration solution selection is presented, based on data gathered in a participatory case study in a mid-sized electricity retailer.

RELATED PUBLICATIONS NOT INCLUDED IN THE THESIS

ANDERSSON, J., P. JOHNSON, "Procurement of Integrated IT Systems for the Deregulated Electric Utility, " *Proceedings of CIRED'99*, 1999.

JOHNSON, P., "Control and Information System Procurement at Vattenfall." In JOHNSON, P, SUNDSTRÖM, M., *Deregulation of the Electricity Market: Effects on Inter-Firm Relations*, Arbetsnotat 8, Program Energisystem, Linköping University, 1999.

BÄCKLUND, M., M. ERIKSSON, P. JOHNSON, M. SILWER, "New markets, new business opportunities: Alternative scenarios and strategies for providing services based on communication," *Proceedings of Distribution Automation and Demand Side Management (DA/DSM) Europe*, 1998.

ANDERSSON, J., P. JOHNSON, *Fallstudie av projekt Blondie: En funktionell upphandling av ett integrerat system för nät- och balansavräkning*, Internal Report, Department of Industrial Information and Control Systems, Royal Institute of Technology (KTH), 1998.

Chapter 8 Conclusions

The primary raison d'être of software architecture as a discipline is its ambition to address issues of large-scale software system structures. Software architecture is concerned with the modeling of these systems and, of particular importance to the present text, analysis of their properties, such as reliability, security and modifiability. For several types of large-scale systems, the discipline has had considerable academic and industrial success; however, there is a broad category of systems that to a large extent have been neglected, here referred to as enterprise software systems. In recognition of this, the present thesis has addressed the extent to which traditional software architecture analysis is applicable to enterprise software system integration. Integration has been focused upon for its considerable importance in the context of enterprise software systems.

In order to address this problem, four subjects have been considered. Firstly, enterprise software systems have been distinguished from the systems traditionally considered by the software architecture discipline. Secondly and thirdly, deduction-based and induction-based methods employed for analysis of traditional systems have been assessed for their applicability to enterprise software system integration. Fourthly, the process employed for analysis of traditional systems has been assessed in the context of enterprise software system integration. A distinction has thus been made between methods and processes, where method denotes the means by which conclusions are drawn from architectural descriptions, and process denotes the engineering context in which these conclusions are drawn.

ENTERPRISE SOFTWARE SYSTEMS

Enterprise software systems are those systems that are managed by organizations primarily interested in using them, as opposed to those developed by organizations primarily interested in selling them. Elaborating briefly, the purpose of these systems is thus to support an enterprise's overarching operations as efficiently as possible, today including most areas of concern to a company, such as production, planning, billing, distribution, external communication, finance, personnel, and more. In contrast to these systems are the systems of singular vendors, typically considered by the traditional software architecture discipline. One might expect the differences of the two system types to be significant, and in the present context it has been justified to consider these differences in greater detail (as presented in Section 5.7, Paper A, and Paper B).

Architectural descriptions of software systems are typically described by their constituent components and connectors, as well as by their system-level characteristics. Approaching enterprise software systems with the same terminology, the present research has identified a number of distinct characteristics. Considering the components of enterprise software systems, their most important distinguishing features are their black-box nature, their heterogeneity and their large-grainedness. The black-box nature is typically a result of third-party development as well as software legacy, and implies that the components are difficult (if at all possible) to modify, and that information about their structure and characteristics is often incomplete. Heterogeneity is measured by the consistency of designs between components. The enterprise software system suffers from considerable component heterogeneity as a result of uncoordinated development by many actors under a long time period. Few are the standards to which all components comply in an enterprise software system. Additionally, components in enterprise software systems are large-grained, often constituted of complete single-vendor systems.

Considering the connectors of enterprise software systems, their most significant trait in comparison with the connectors of systems traditionally considered by the software architecture discipline is their heterogeneity. Connector types of traditional systems are typically limited to one or a few, such as procedure calls or pipes. In contrast, enterprise software systems in the general case contain a significant number of conceptually different connectors, such as remote procedure calls, messaging technologies, file transfer mechanisms, and more.

Considering the overarching characteristics of the enterprise software system, including process-related aspects, three issues of particular importance have been identified. Firstly, the supply of components is limited, and secondly, there is always a considerable legacy to take into account. Whereas traditional software system development generally assumes the possibility of green-field development of custom-made components, enterprise software system development is normally limited to market procurement of components. Further limitations are introduced by the legacy of components (as well as connectors). An important consequence of this restriction is that the design space of the architect goes from a continuum of potential architectures to a discrete set, limited by the available components. Thirdly, a system composed of prefabricated entities is liable to contain a considerable redundancy, with respect to data as well as functionality.

To summarize, when attempting to describe enterprise software architectures, several important characteristics appear. Firstly, they are characterized by a size that makes the information required to completely describe them overwhelming. Secondly, this information is often both difficult to find and unreliable. Thirdly, the systems are heterogeneous, which makes it difficult to generalize over them. Most generalized assumptions (e.g. about component behavior) are therefore questionable in the context of enterprise software systems. Finally, arguably as a combination of their complexity and short history, they are not well understood; there are few known laws governing the properties of enterprise software systems.

DEDUCTION-BASED ARCHIECTURAL ANALYSIS METHODS

As mentioned, the thesis has considered both methods and processes for software architecture analysis. Two conceptually different kinds of methods are distinguished in the present work, deduction-based and induction-based approaches.

Deduction-based architectural analysis methods are based on a paradigm of reductionistic rationality, where properties of a system are inferred from underlying models of component and connector behavior and structure. For instance, models of inter-component communication may be employed for identifying potential deadlocks in a system. Typically, deduction-based analysis methods are based on formal specifications of the architectures, as well as formal theories for generating predictions of system properties. Although deduction-based architectural analysis methods have been successfully applied to a significant number of systems, this thesis indicates that considerable problems are encountered when addressing enterprise software system integration (cf. Section 6.5 and Paper C).

In the context of enterprise software systems, there are some properties of deduction-based analysis methods that are of particular importance. Firstly, because deduction-based methods are based on (formal) theories, they can only be applied to systems described by abstractions appropriate for the given theory; presently, these abstractions are generally on the conceptual level of programming languages (or below). Secondly, the theories underlying the deduction-based methods make fairly extensive assumptions about the analyzed system (for instance that the components are implemented on real-time platforms). Thirdly, the deduction-based architectural analyses reviewed in the thesis are particularly concerned with certain architectural issues, such as behavioral aspects.

When assessing the applicability of these deduction-based methods on enterprise software system integration, the thesis has identified a number of problems that schematically can be explained with the above characteristics of the methods and enterprise software systems.

Firstly, the programming-language abstractions provided by the deduction-based methods are not well suited for the richness of information present in an enterprise software system. The present deduction-based abstractions are thus insufficiently abstract. Secondly and related, analyses based on programming-language abstractions often yield unreliable results, as low-level information on enterprise software systems may be both difficult to find and (if found) flawed. For instance (as considered in Section 6.5), when attempting to model the behavior of components and connectors, as well as error behavior, these problems becomes visible.

Thirdly, in the cases when deduction-based abstractions are on a satisfactory level, they are, as mentioned, often based on fairly extensive assumptions about the system. These assumptions are typically of a generalized nature, presupposing homogeneity of, for instance, component behavior. Enterprise software systems, however, are heterogeneous to their nature. It is rarely safe to assume, for instance, that all components are implemented on real-time platforms, or that all components comply with a certain rule set. Chapter 6 of the thesis presents several cases where this conflict becomes problematic, including the deduction-based analysis of extra-functional properties, such as security, reliability, atomicity and performance as well as the analysis of data syntax and semantics incompatibilities.

The fourth issue concerns the sensitivity of deduction-based methods to minor inconsistencies between the architectural description and the actual system. If, as considered above, the architectural description is not completely accurate, will the analysis be completely useless? Presently, as discussed in Paper C, the sensitivity of analysis methods is more or less unexplored. It is thus not known whether minor misrepresentations will completely invalidate the analysis or not. Paper C indicates that such analysis incorrectness may in fact occasionally be the result of minor representational inaccuracies.

INDUCTION-BASED ARCHITECTURAL ANALYSIS METHODS

Architectural styles (or patterns) are typically defined as proven solutions to recurring problems. In the present work, architectural styles have been used as representatives of the induction-based approach. Induction-based methods address the task of architectural analysis quite differently from their deduction-based relatives, drawing conclusions by analogy to similar architectures rather than based on formally expressed rule sets. Causation is thus substituted by association, in the sense that the main concern no longer is why an architecture demonstrates certain properties, only that the properties appear when the architecture takes on a particular form (i.e., a particular style).

In the context of enterprise software systems, there are some properties of induction-based analysis methods that are of particular importance. Firstly, they are based on empirical relations rather than theoretical constructs. The only theory relevant for an architectural style is the relation between the style and the system properties that are associated with it, e.g. the relation between the black-board style and the extra-functional property modifiability. Secondly, because the underlying theory is so meager, there are few restrictions on the employed abstractions. Recall that deduction-based approaches in this respect are limited to the abstractions acceptable to the (formal) theories on which they are based. Thirdly and related, induction-based methods do per se not make extensive assumptions about the considered system. The only characteristics of the system that are explicitly assumed are those that define the style description. Fourthly, induction-based methods are suitable for assessing externally measurable properties, e.g. integration cost or throughput (while deduction-based assessments are derived from the internal workings of the considered system).

These characteristics of induction-based methods have several consequences for their applicability to enterprise software system integration. Firstly, the description of styles is facilitated by lenient requirements on acceptable abstractions. Abstractions suitable for the enterprise software system context – rather than for program-level theories – may thus be employed.

Secondly, because the requirements on underlying theory are limited, the proposition of styles and their relations to system properties becomes fairly straight-forward. Paper D exemplifies these two issues by describing a number of styles for enterprise software system integration as well as their expected impact on a number of extra-functional properties of the system. However, since there is little underlying theory, other validation means than deduction are required. Since the concept of induction is closely related to statistics, surveys over the relations between styles and system properties should arguably fill this validating function.

Thirdly, induction-based approaches are not explicitly based on extensive assumptions about the modeled system in the way deduction-based approaches are. If a strong correlation between e.g. reliability and a certain architectural style has been statistically determined, then this relation is valid whether the operating system is implemented on a real-time platform or not. Because generalized assumptions about enterprise software systems often are questionable due to the heterogeneity of the systems and the unreliability of available information, this becomes a major benefit of induction-based methods. However, unintended assumptions may be introduced in the relations between styles and system properties as a result of an unrepresentative statistical selection when validating styles. In other words, all the systems used to (statistically) determine the relationship between a certain style and a certain system property may be similarly constructed, thus introducing unintended assumptions.

ARCHITECTURAL ANALYSIS PROCESS

The analysis process, in contrast to the above considered analysis methods, is the engineering process surrounding the actual analysis of architectural descriptions. The traditional analysis process typically contains the following activities: 1) Objectives, requirements, constraints, and context elicitation; 2) Scenario construction; 3) Architectural representation; 4) Actual architectural analysis; and 5) Architectural modification. The analysis methods considered above are typically employed in the fourth activity (actual architectural analysis). Paper B considers to what extent this process is extensible to enterprise software system integration analysis. Considering the characteristics of the enterprise software system, the paper proposes a number of modifications to this process. Three of the most important consequences of the move to enterprise software systems are considered here.

Firstly, the legacy of enterprise software systems is normally significant and there is a market from which to select predefined components. In comparison with the traditional analysis scenario, these two enterprise software system characteristics require a considerable effort for context description, gathering information about the present system and any considered new components to be procured. Secondly and related, the legacy and the limited availability of components introduce substantial constraints on the design space. Whereas the traditional process presupposes a considerable freedom in modifications to the architecture, the design alternatives for the enterprise software system are typically few.

Thirdly, the heterogeneity of both components and connectors makes integration a prime issue during architectural representation. Common assumptions, such as standards compliance, cannot be assumed for enterprise software systems, while any new functionality is typically already present either in the legacy or in components available on the market. Analysis of the integration aspects of enterprise software systems thus become a key focus.

SUMMARY

The application of software architecture analysis in the context of enterprise software system integration is far from straightforward. In many respects, the increase in complexity is similar to the move from chemistry to biology, where the low-level theories are insufficient to explain and predict the high-level phenomenon. To understand the workings of the higher-level systems, it is necessary to explore their behavior empirically and devise new models from observations. In most disciplines, however, a synthesis is desired between empirical induction-based and theoretical deduction-based approaches. For instance, a deduction-based approach may be employed for generating hypotheses, which are then tested by induction-based approaches.

In the introduction to this thesis, the research area was partially motivated by a growing realization at our department of the need for software engineering methods for enterprise software system management. At the same time, there seems to be a growing realization in the software engineering community of the "enterprise software" nature of traditional systems. Component-based engineering is more popular now than ever, even simple systems employ unmodifiable commercial-of-the-shelf software, and concepts such as product-line architectures challenge the traditional assumption of software development from scratch. In this sense, the distinction between traditional software systems and enterprise software systems is already blurring. From the perspective of the author, this convergence is a good thing.

Chapter 9
Further Works

The conclusions of this thesis unavoidably lead to suggestions for further works. This section suggests two areas of particular interest, related to induction-based and deduction-based analysis respectively.

Architectural integration styles. The present work concludes by highlighting the possibilities of induction-based architectural analysis. The area has, however, only been subjected to a superficial exploration. Further works within the area include additional documentation of architectural integration styles as well as empirical validation of their impact. As of yet, styles and patterns have primarily been validated in two ways: theoretically and pseudo-empirically. Theoretical validation of styles is based on the deduction-based method of formal specifications combined with extensive assumptions about the underlying system. The pseudo-empirical validation has mainly been limited to individual reports of successful style application. In general, when complexity increases and the credibility of the assumptions become dubious, statistical methods are employed as a complement in the validation. Also for architectural styles, this method appears reasonable for determining whether styles actually have the expected effects.

Deduction-based analysis. Paper C explores some of the fundaments of deduction-based architectural analysis. Further investigation into the reliability of these methods for early prediction of enterprise software system properties is proposed as further work. Deduction-based analysis is based on fairly extensive assumptions that rarely are known to be true, e.g. perfect transformations from architectural specification to implementation, homogeneity of components and connectors, simplifications, etc. The effects of relaxations of these assumptions on the credibility of the analysis results are currently poorly understood. Both theoretical and empirical studies exploring these effects belong to the suggested further works.

References

[Abd96] Abd-Allah, A., *Composing Heterogeneous Software Architectures* (Ph.D. Thesis), University of Southern California, 1996.

[Add02] d'Adderio, L., R. Dewar, A. Lloyd, P. Stevens, "Has the Pattern Emperor any Clothes? A Controversy in Three Acts," *Software Engineering Notes*, 2002.

[Aho86] Aho, A., R.Sethi, J., Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Ala98] Alagar, V., K. Periyasamy, *Specification of Software Systems*, Springer, 1998.

[Ale74] Alexander, C., *Notes on the Synthesis of Form*, Harvard University Press, 1974.

[All94a] Allen, R., D. Garlan, "Beyond Definition/Use: Architectural Interconnection," *Proceedings of the Workshop on Interface Definition Languages*, 1994.

[All94b] Allen, R., D. Garlan, "Formalizing Architectural Connection," *Proceedings of the 16th International Conference on Software Engineering*, 1994.

[All97] Allen, R., *A Formal Approach to Software Architecture* (Ph.D. Thesis), Carnegie Mellon University, 1997.

[All98] Allen, R., R. Douence, D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering*, 1998.

[All99] Allamaraju, S., "Nuts and Bolts of Transaction Processing", White Paper, http://www.subrahmanyam.com/articles/transactions/NutsAndBoltsOfTP.html, 1999 [Accessed 11 April, 2002].

[And98] Andersson, J., P. Johnson, *Fallstudie av projekt Blondie: En funktionell upphandling av ett integrerat system för nät- och balansavräkning*, Internal Report, Department of Industrial Information and Control Systems, Royal Institute of Technology (KTH), 1998.

[And99] Andersson, J., P. Johnson, "Procurement of Integrated IT Systems for the Deregulated Electric Utility, " *Proceedings of CIRED'99*, 1999.

[And00a] Andersson, J., P. Johnson, "IT Infrastructure Architectures for Electric Utilities: A Comparative Analysis of Description Techniques," *Proceedings of the 33rd Hawaii International Conference on System Sciences*, 2000.

[And00b] Andersson, J., P. Johnson, "Extending Attribute-based Architectural Analysis to Enterprise Software Systems," *Proceedings of the 3rd Australasian Workshop on Software and System Architectures*, 2000.

[And01a] Andersson, J., P. Johnson, "Architectural Integration Styles for Large-Scale Enterprise Software Systems," *Proceedings of the 5th IEEE International Enterprise Distributed Object Computing Conference*, 2001.

[And01b] Andersson, J., T. Cegrell, K-H. Cheong, M. Haglind, "Strategic Management of Information Technology in Small and Medium Sized Electric Utilities: Bridging the Gap Between Theory and Practice," *Proceedings of the Portland International Conference on Management of Engineering and Technology*, 2001.

[And02] Andersson, J., *Enterprise Information Systems Management: An Engineering Perspective on the Aspects of Time and Modifiability*, Ph.D. Thesis, Royal Institute of Technology (KTH), 2002.

[App00] Appleton, B., "Patterns and Software: Essential Concepts and Terminology," http://www.enteract.com/~bradapp/docs/patterns-intro.html, Modified 2000 [Accessed 2001].

[Aqu01] Aquilani, F., S. Balsamo, P. Inverardi, "Performance analysis at the software architectural design level," *Performance Evaluation*, Elsevier, 2001.

[Arc01] ARC Advisory Group, "Enterprise Integration Market 20% Annual Growth to $11 Billion by '06," ARC Advisory Group, http://www.arcweb.com/arcweb/aboutarc/arcnews/pressitem.asp?ID=212 [Accessed December 17, 2001]. Cited in "Företagintegration för 120 miljarder," *ComputerSweden*, 31 October, 2001.

[Avg00] Avgeoru, C., "Information Systems: What Sort of Science Is It?" *Omega*, 2000.

[Baa02] Baan Website, http://www.baan.com, Baan. [Accessed April 4, 2002].

[Bac87] Bach, M., *Design of the Unix Operating System*, Prentice-Hall, 1987.

[Bac00] Bachmann, F., L. Bass, G. Chastek, P. Donohue, F. and Peruzzi, *The Architecture-Based Design Method*, Technical Report CMU/SEI-2000-TR-01, CMU SEI, 2000.

[Bar98] Baragry, J., K. Reed, "Why is it so hard to define Software Architecture?" *Proceedings of the 1998 Asia Pacific Software Engineering Conference*, 1998.

[Bar95a] Barbacci, M., M. Klein, T. Longstaff, C. Weinstock, *Quality Attributes*, Technical Report CMU/SEI-95-TR-021, 1995.

[Bar95b] Barjaktarovic, M., S-K. Chin, K. Jabbour, "Formal specification and verification of communicating protocols using automated tools," *Proceedings of the First IEEE International Conference Engineering of Complex Computer Systems,* 1995.

[Bar92] Barry, P., "Abstract syntax notation-one (ASN.1)," *IEE Tutorial Colloquium on Formal Methods and Notations Applicable to Telecommunications*, 1992.

[Bas96] Basili, V., "The Role of Experimentation in Software Engineering: Past, Current, and Future," *Proceedings of the 18th International Conference on Software Engineering*, 1996.

[Bas98] Bass, L., P. Clements, R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

[Bas01a] Bass, L., "Software Architecture Design Principles," In *Component-Based Software Engineering: Putting the Pieces Together*, Eds. Heineman, G., Councill, W., 2001.

[Bas01b] Bass, L., B. John, "Supporting Usability Through Software Architecture," *IEEE Computer*, 2001.

[Bec87] Beck, K., W. Cunningham, “Using Pattern Languages for Object-Oriented Programs,” *Proceedings of the OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming*, 1987.

[Ben00] Bengtsson, P., N. Lassing, J. Bosch, H. van Vliet, “Analyzing Software Architectures for Modifiability,” *Technical Report*, University of Karlskrona/Ronneby, 2000.

[Ben02] Bengtsson, P., *Architecture-Level Modifiability Analysis* (Ph.D. Thesis), Blekinge Institute of Technology, 2002.

[Ber00] Bergner, K., A. Rausch, M. Sihling, A. Vilbig, “Adaptation strategies in componentware,” *Proceedings of the Australian Software Engineering Conference*, 2000.

[Bir84] Birrell, A., J. Nelson, “Implementing Remote Procedure Calls,” *ACM Transactions on Computer Systems*, 1984.

[Bjö82] Björner, D., ”Stepwise Transformation of Software Architectures,” In *Formal Specification and Software Development*, Prentice-Hall, 1982.

[Boe81] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[Boe88] Boehm, B., “A Spiral Model of Software Development and Enhancement,” *IEEE Computer*, 1988.

[Boo99] Booch, G., J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[Bos99] Bosch, J., P. Molin, “Software Architecture Design: Evaluation and Transformation,” *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems*, 1999.

[Bos00] Bosch, J., *Design and Use of Software Architectures*, Addison-Wesley, 2000.

[Boy99] Boyle, J., R. Resler, V. Winter, "Do you trust your compiler?" *IEEE Computer*, 1999.

[Bra00] Bray, T., J. Paoli, C. Sperberg-McQueen, E. Maler, *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C, 2000.

[Bri92] Bright, M., A. Hurson, S. Pakzad, “A Taxonomy and Current Issues in Multidatabase Systems,” *IEEE Computer,* 1992.

[Bri97] Briand, L., J.Daly, J. Wust, “A Unified Framework for Cohesion Measurement in Object-Oriented Systems,” *Proceedings of the Fourth International Software Metrics Symposium*, 1997.

[Bri99] Briand, L., Daly, J., Wüst, J., “A Unified Framework for Coupling Measurement in Object-Oriented Systems,” *IEEE Transactions on Software Engineering*, 1999.

[Bri02] Encyclopædia Britannica Online, "Ockham's razor," *Encyclopædia Britannica Online*, http://www.eb.com:180/bol/topic?eu=58133&sctn=1, Accessed April 1 2002.

[Bro75] Brooks, F., *The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering*, First published in 1975, Addison-Wesley, 1995.

[Bro00] Brown, A., *Large-Scale, Component-Based Development*, Prentice-Hall, 2000.

[Bus96] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.

[But99] Butler Group, *Application Integration Management Guide*, Butler Direct Ltd, 1999.

[Bäc98] Bäcklund, M., M. Eriksson, P. Johnson, M. Silwer, “New markets, new business opportunities: Alternative scenarios and strategies for providing services based on communication,” *Proceedings of Distribution Automation and Demand Side Management (DA/DSM) Europe*, 1998.

[Cam86] Campbell, D., "Rationality and Utility from the Standpoint of Evolutionary Biology," *Journal of Business*, 1986.

[Cat95] Cattaneo, F., "An Evaluation of Architecture Description Languages," *Technical Report 96.152*, Politecnico di Milano, 1995.

[Ceg86] Cegrell, T., *Power Systems Control Technology*, Prentice-Hall, 1986.

[Cha96] Chappell, D., *Understanding ActiveX and OLE: A Guide for Developers and Managers*, Microsoft Press, 1996.

[Che94] Cheung, S., J. Kramer, "Tractable dataflow analysis for distributed systems," *IEEE Transactions on Software Engineering*, 1994.

[Che97] Cheong, K-H., *Distribution Automation: Cost-Effective Introduction Strategies* (Licentiate Thesis), KTH Royal Institute of Technology, 1997.

[Che01] Chester, T., "Cross-Platform Integration with XML and SOAP," *IT Professional*, 2001.

[Chi00] Chiang, C., "Wrapping Legacy Systems for Use in Heterogeneous Computing Environments," *Information and Software Technology*, 2000.

[Cim98] Cimitile, A., U. De Carlini, A. De Lucia, "Incremental migration strategies: data flow analysis for wrapping," *Proceedings of the Fifth Working Conference on Reverse Engineering*, 1998.

[Cla96] Clarke, E., J. Wing, et al., "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, 1996.

[Cle96] Clements, P., "A Survey of Architecture Description Languages," *Proceedings of the 8th International Workshop on Software Specification and Design*, 1996.

[Com99] Compare, D., P. Inverardi, A. Wolf, "Uncovering architectural mismatch in component behavior," *Science of Computer Programming*, 1999.

[Cza00] Czarnecki, K., U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[Dag00a] Dagens Industri, " Vattenfall förvärvar polsk eljätte för 4 mdr," *Dagens Industri*, 2000.

[Dag00b] Dagens Industri, " Klart i dag för Vattenfalls östtyska köp," *Dagens Industri*, 2000.

[Dag00c] Dagens Industri, " Vattenfall blir stor aktör i Norge," *Dagens Industri*, 2000.

[Dag00d] Dagens Industri, " Vattenfall går in i avfallsförbränning," *Dagens Industri*, 2000.

[Dag00e] Dagens Industri, " Vattenfall blir stor aktör i Norge," *Dagens Industri*, 2000.

[Dag00f] Dagens Industri, " Sydkraft gör ny miljardaffär," *Dagens Industri*, 2000.

[Dag00g] Dagens Industri, " Sydkraft köper avfallsbolag," *Dagens Industri*, 2000.

[Dag00h] Dagens Industri, " Franskt kraftbolag stärker greppet om Graninge," *Dagens Industri*, 2000.

[Dag01a] Dagens Industri, "Birka Energi till Fortum för 14,5 miljarder kronor," *Dagens Industri*, 2001.

[Dag01b] Dagens Industri, " Vattenfall köper tyska HEW," *Dagens Industri*, 2001.

[Dag01c] Dagens Industri, " Vattenfall köper Bewag," *Dagens Industri*, 2001.

[Dag01d] Dagens Industri, " Klart i dag för Vattenfalls östtyska köp," *Dagens Industri*, 2001.

[Dav93] Davis, A., *Software Requirements: Objects, Functions, and States,* Prentice Hall, 1993.

[Dei84] Deitel, H., *An Introduction to Operating Systems*, Addison-Wesley, 1984.

[Del96] Dellarocas, C., *A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components* (Ph.D. Thesis), MIT Center for Coordination Science, 1996.

[Del97a] Dellarocas, C., "Towards a Design Handbook for Integrating Software Components," *Proceedings of the Fifth International Symposium on Assessment of Software Tools and Technologies*, 1997.

[Del97b] Dellarocas, C., "The Synthesis Environment for Component-Based Software Development," *Proceedings of the Eighth IEEE International Workshop on Software Technology and Engineering Practice*, 1997.

[DeL99] DeLine, R., *Resolving Packaging Mismatch* (Ph.D. Thesis), Carnegie Mellon University, 1999.

[Dia93] Diaz-Herrera, J., "The importance of static structures in software construction," *IEEE Software*, 1993.

[Dij68a] Dijkstra, E., "Letters to the Editor: Go To Statement Considered Harmful," *Communications of the ACM*, 1968.

[Dij68b] Dijkstra, E., "The Structure of the "THE"-Multiprogramming System," *Communications of the ACM*, 1968.

[Dij69] Dijkstra, E., *Notes on Structured Programming*, Technical U. Eindhoven, 1969.

[Dij76] Dijkstra, E., *A Discipline of Programming*, Prentice-Hall, 1976.

[Dun96] Duncan, W., *A Guide to the Project management Body of Knowledge*, PMI Standards Committee, 1996.

[Ede94] Eder, J., G. Kappel, M. Schrefl, "Coupling and Cohesion in Object-Oriented Systems," *Technical Report*, Univ. of Klagenfurt, 1994.

[Emm01] Emmerich, W., E. Ellmer, H.Fieglein, "Tigra – An Architectural Style for Enterprise Application Integration," *Proceedings of the 23rd International Conference Software Engineering*, 2001.

[Ene00] Energimyndigheten, *Utvecklingen på Elmarknaden 2000:I - Schablonberäkning*, Energimyndigheten, 2000.

[Ene01] Energimyndigheten, *Elmarknaden 2001*, Energimyndigheten, 2001.

[Eng99] Engelken, L., A. Gay, H. Tram, "Development of an Information Technology Strategy and Architecture for Energy Delivery Utility Mergers," *1999 IEEE Transmission and Distribution Conference*, 1999.

[Eri93] Ericsson, G., P. Forsgren, E. Gyllenswärd, T. Rahkonen, "A State of the Art Study of Commercial Industrial Control Systems," *External Report,* Department of Industrial Control Systems Royal Institute of Technology (KTH), 1993.

[Eri95] Ericsson, G., T. Rahkonen, "Openness in Communication for Power System Control: A State-of-the-Practice Study," *Proceedings of Stockholm PowerTech, IEEE International Symposium on Electric Power Engineering*, 1995.

[Eri96] Ericsson, G., J. Schubert, "Evolution of Communication in Power System Control." In Ericsson, G*., On Power System Control*, Ph.D. Thesis, Royal Institute of Technology, 1996.

[Fie97] Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and T. Berners-Lee, *Hypertext Transfer Protocol -- HTTP/1.1*, RFC 2068, January 1997.

[Foi85] Fogiel, M., *Handbook and Guide for Comparing and Selecting Computer Languages*, Research and Education Association, 1985.

[Gac95] Gacek, C., A. Abd-Allah, B. Clark, B. Boehm, "On the Definition of Software System Architecture," *Proceedings of the ICSE 17 Software Architecture Workshop*, 1995.

[Gac98] Gacek, C., *Detecting Architectural Mismatch During System Composition* (Ph.D. Thesis), University of Southern California, 1998.

[Gam98] Gamma, E., R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1998.

[Gan00] Gannod, G., S. Mudiam, T., Lindquist, "An Architecture-Based Approach for Synthesizing and Integrating Adapters for Legacy Software," *Proceedings of the Seventh Working Conference on Reverse Engineering*, 2000.

[Gar94a] Garlan, D., R. Allen, J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard," *IEEE Software*, 1994.

[Gar94b] Garlan, D., R. Allen, J. Ockerbloom, "Exploiting Style in Architectural Design Environments," *Proceedings of SIGSOFT '94*, 1994.

[Gar95a] Garlan, D. "What Is Style?" *Proceedings of the Dagshtul Workshop on Software Architecture*, 1995.

[Gar95b] Garlan, D., "An Introduction to the Aesop System," 1995. http://www.cs.cmu.edu/afs/cs/project/able/ www/aesop/html/aesop-overview.ps

[Gar97] Garlan, D., R. Monroe, D. Wile, "ACME: An Architecture Description Interchange Language," *Proceedings of CASCON'97*, 1997.

[Gar98] Garlan, D., J. Ockerbloom, D. Wile, "Towards an ADL Toolkit," *EDCS Architecture and Generation Cluster*, 1998. http://www-2.cs.cmu.edu/~acme/adltk/index.html.

[Gar00] Garlan, D., "Software Architecture: A Roadmap", *The Future of Software Engineering* (volume published in conjunction with *the 22nd International Conference on Software Engineering*), ACM Press, 2000.

[Gla98] Glancer, D., *Technical Overview: Foundation Fieldbus, FD-043 Revision 2.0*, Fieldbus Foundation, 1998.

[Gol99] Gold-Bernstein, B., "EAI market Segmentation," *EAI Journal*, 1999.

[Hag02] Haglind, M., *On Information Systems Planning in Small and Medium-sized Electric Utilities* (Ph.D. Thesis), KTH Royal Institute of Technology, 2002.

[Har94] Hartley, J., "Case Studies in Organizational Research," In Cassel, C., G. Symon (ed's), *Qualitative Methods in Organizational Research: A Practical Guide*, Sage Publications, 1994.

[Hei01a] Heineman, G., W. Councill (Eds), *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.

[Hei01b] Heineman, G., W. Councill, "Definition of a Software Component and Its Elements". In Heineman, G., W. Councill (Eds), *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.

[Hel00] Helander, J., *Supply Chain Evolution in the Manufacturing Industry* (Licentiate Thesis), Royal Institute of Technology (KTH), 2000.

[Hen92] Henshall, J., *Opening up OSI: An Illustrated Introduction*, Ellis Horwood, 1992.

[Hit95] Hitz, M., B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems," *Proceedings of the International Symposium on Applied Corporate Computing*, 1995.

[Hoa85] Hoare, C., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hof99] Hofmeister, C., R. Nord, D. Soni, *Applied Software Architecture*, Addison-Wesley, 1999.

[Hon01] Hong, K-K., Y-G. Kim, "The critical success factors for ERP implementation: an organizational fit perspective," *Information & Management*, In Press, Uncorrected Proof, Available online 28 November 2001.

[Hua98] Huang, Y., Ravishanker, C., "Constructive Protocol Specification Using Cicero," *IEEE Transactions on Software Engineering*, 1998.

[Huß97] Hußmann, H., *Formal Foundations for Software Engineering Methods*, Lecture notes in Computer Science, Springer-Verlag, 1997.

[Hut00] Michael Huth, Mark Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*, Cambridge University Press, 2000

[Här99] Härder, T., Sauter, G., Thomas, J., "The Intrinsic Problems of Structural Heterogeneity and an Approach to Their Solution," *The VLDB Journal*, 1999.

[IEE00a] IEEE, 802.3: *Carrier Sense Multiple Access with Collision Detection*, IEEE, 2000.

[IEE00b] IEEE 1076-2000: *Standard VHDL Language Reference Manual*, IEEE, 2000.

[ILo00] I-Logix, Rhapsody: *Reference Guide*, I-Logix Inc., 2000.

[Ima00] Imamura, T., H. Maruyama, "Mapping Between ASN.1 and XML," *IBM Research Report RT0362*, 2000.

[Jac99] Jacobson, I., G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[Jaz00] Jazayeri, M., A. Ran, A., F. van der Linden, *Software Architecture for Product Families*, Addison-Wesley, 2000.

[Jep01] Jepsen, T., "SOAP Cleans Up Interoperability Problems on the Web," *IT Professional*, vol. 3, no. 1, 2001.

[Jma00] Jmaiel, M., "A Unified Algebraic Framework for Specifying Communication Protocols," *Proceedings of the Third IEEE International Conference on Formal Engineering Methods*, 2000.

[Joh99] Johnson, P., "Control and Information System Procurement at Vattenfall." In Johnson, P, Sundström, M., *Deregulation of the Electricity Market: Effects on Inter-Firm Relations*, Arbetsnotat 8, Program Energisystem, Linköping University, 1999.

[Kaz94a] Kazman, R., L. Bass, G. Abowd, M. Webb, "SAAM: A method for analyzing properties of software architectures," *Proceedings of the 16th International Conference on Software Engineering*, 1994.

[Kaz94b] Kazman R., L. Bass, Toward Deriving Software Architectures from Quality Attributes, *Technical Report CMU/SEI-94-TR-10*, 1994.

[Kaz97] Kazman, R., P. Clements, L. Bass, "Classifying Architectural Elements for Mechanism Matching", *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference*, 1997.

[Kaz98] Kazman, R., M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere, "The Architecture Tradeoff Analysis Method," *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems*, 1998.

[Kaz99] Kazman, R., M. Barbacci, M. Klein, J. Carriere, "Experience with performing architecture tradeoff analysis," *Proceedings of the 1999 International Conference on Software Engineering*, 1999.

[Kha95] Khandker, A., P. Honeyman, T. Teorey, "Performance of DCE RPC," *Second International Workshop on Services in Distributed and Networked Environments*, 1995.

[Kim91] Kim, W., J. Seo, "Classifying Schematic and Data Heterogeneity in Multidatabase Systems," *IEEE Computer*, 1991.

[Kin93] King, K., "Modula-3: a threat to Ada?" *Proceedings of the Tenth Annual Washington Ada Symposium on Ada's Role in Software Engineering*, 1993.

[Kin94] King, G., R. Keohane, S. Verba, *Designing Social Inquiry: Scientific Inference in Qualitative Research*, Princeton University Press, 1994.

[Kle91] M. Klein, J. Goodenough, "Rate Monotonic Analysis for Real-Time Systems," *Technical Report*, 1991.

[Kog95] Kogut, P., P. Clements, "Features of Architecture Description Languages," *Proceedings of Software Technology Conference*, 1995.

[Kom98] Kompanek, A., *AcmeStudio: User's Manual*, Carnegie Mellon University, 1998.

[Kru95] Kruschten, P., "The 4 + 1 View Model of Architecture," *IEEE Software*, 1995.

[Lan90] Lane, T., A Design Space and Design Rules for User Interface Software Architecture, *Technical Report CMU/SEI-90-TR-223*, 1990.

[Las99] Lassing, N., D. Rijsenbrij, H. van Vliet, "Towards a Broader View on Software Architecture Analysis of Flexibility," *Proceedings of the Sixth Asia Pacific Conference on Software Engineering*, 1999.

[Las02] Lassing, N., P-O. Bengtsson, H. van Vliet, J. Bosch, "Experiences with ALMA: Architecture-Level Modifiability Analysis," *Journal of Systems and Software*, 2002.

[Lin00] Linthicum, D., *Enterprise Application Integration*, Addison-Wesley, 2000.

[Lin01a] Linthicum, D, *B2B Application Integration: E-business-Enable Your Enterprise*, Addison-Wesley, 2001.

[Lin01b] Lindgren, L., *Application Servers for E-Business*, Auerbach, 2001.

[Lit01] Litwin, L., "The medium access control sublayer," *IEEE Potentials*, 2001.

[Lon01] Longshaw, A., "Choosing between COM+, EJB, and CCM" In Heineman, G., W. Councill (Eds), *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.

[Luc95a] Luckham, D., L. Augustin, J. Kenny, J. Vera, D Bryan, W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions of Software Engineering*, 1995.

[Luc95b] Luckham, D., J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions of Software Engineering*, 1995.

[Luc97] Lucia, A. De, G. Di Lucca, A. Fasolino, P. Guerra, S. Petruzzelli, "Migrating legacy systems towards object-oriented platforms," *Proceedings of the International Conference on Software Maintenance*, 1997.

[Mag95] Magee, J., N. Dulay, S. Eisenbach, J. Kramer, "Specifying Distributed Software Architectures," *Proceedings of the 5th European Software Engineering Conference*, 1995.

[Mag96] Magee, J., J. Kramer, "Dynamic Structure in Software Architectures," *4th Symposium on the Foundations of Software Engineering*, 1996.

[Mag97a] Magee, J., J. Kramer, D. Giannakopoulou, "Analyzing the Behaviour of Distributed Software Architectures: a Case Study," *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, 1997.

[Mag97b] Magee, J., A. Tseng, J. Kramer, "Composing distributed objects in CORBA," *3rd International Symposium on Autonomous Decentralized Systems*, 1997.

[Mal84] Malone, J., *Comparative Languages*, Chartwell-Bratt, 1984.

[Mal94] Malone, T., K. Crowston, "The Interdisciplinary Study of Coordination," *ACM Computing Surveys*, 1994.

[McC77] McCall J., P. Richards, G. Walters, *Factors in Software Quality, Vols I, II, III,* US Rome Air Development Center Reports, 1977.

[McG00] McGoveran, D., "Architected Simplicity," *EAI Journal*, 2000.

[Med96] Medvidovic, N., P. Oreizy, J. Robbins, R. Taylor, "Using Object-Oriented Typing to Support Architectural Design in the C2 Style," *ACM SIGSOFT Software Engineering Notes, Proceedings of the fourth ACM SIGSOFT symposium on Foundations of software engineering*, 1996.

[Med97] Medvidovic, N., D. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages," *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages*, 1997.

[Med98] Medvidovic, N., D. Rosenblum, R. Taylor, A Type Theory for Software Architectures, *Technical Report UCI-ICS-98-14*, University of California, 1998.

[Med99] Medvidovic, N., D. Rosenblum, R. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution," *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[Med00] Medvidovic, N., R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, 2000.

[MGr00] McGrath, R., "Discovery and Its Discontents: Discovery Protocols for Ubiquitous Computing", *Computer Science Technical Report UIUCDCS-R-99-2132*, National Center for Supercomputing Applications, 2000.

[Mik98] Mikkonen, T., "Formalizing design patterns," *Proceedings of the 1998 International Conference on Software Engineering*, 1998.

[Mod91] Modiri, N., "The OSI reference model entities," *IEEE Network*, 1991.

[Mon97] Monroe, R., A. Kompanek, R. Melton, D. Garlan, 1997, "Architectural Styles, Design Patterns, and Objects," *IEEE Software*, 1997.

[Mon00] Monson-Haefel, R., *Enterprise JavaBeans, 2nd Ed.*, O'Reilly & Associates, 2000.

[Moo95] Moormann Zaremski, A., J. Wing, "Signature Matching: A Tool for Using Software Libraries", *ACM Transactions on Software Engineering and Methodology*, 1995.

[Moo97] Moormann Zaremski, A., J. Wing, "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, 1997.

[Mor01] Morgenthal, JP., *Enterprise Application Integration with XML and Java*, Prentice-Hall, 2001.

[Mor95] Moriconi, M, X. Qian, R., Riemenschneider, L. Gong, "Correct Architecture Refinement," *IEEE Transactions on Software Engineering*, 1995.

[Mor97a] Moriconi, M., X. Qian, R., Riemenschneider, L. Gong, "Secure Software Architectures", *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[Mor97b] Moriconi, M., R. Riemenschneider, "Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies", *Technical Report SRI-CSL-97-01*, SRI Computer Science Laboratory, 1997.

[Mou01] Mougin, P., C. Barriolade, "Web Services, Business Objects and Component Models," *White Paper*, Orchestra Networks, 2001.

[MSD01] Microsoft, *Microsoft Developer Network Online Library*, http://msdn.microsoft.com/library/default.asp, Microsoft Corporation, 2001.

[Mur99] Murphy, G, R. Walker, E. Baniassad, "Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming," *IEEE Transactions on Software Engineering*, 1999.

[Obe98] Oberndorf, P., "COTS and Open Systems," *SEI Monographs on the Use of Commercial Software in Government Systems*, 1998.

[Obj01] The Object Management Group, *The Common Object request Broker: Architecture and Specification, Version 2.4.2*, The Object Management Group, 2001.

[Ola01] Olavsrud, T., "Report: IT/IS Spendings Bottoms Out," *Internetnews.com*, 2001. http://www.internetnews.com/ent-news/article/0,,7_942311,00.html [Accessed March 31, 2002].

[Par72] Parnas, D., "On the Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, 1972.

[Par83] Partsch, H., R. Steinbrüggen, "Program Transformation Systems," *ACM Computing Surveys*, 1983.

[Per92] Perry, D., A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, 1992.

[Pit01] Pittser, T., "Computer Applications in the Electric Utility Industry," *2001 Rural Electric Power Conference*, 2001.

[Plo02] Witthawaskul, W., "Pattern Languages of Programs Conference," http://jerry.cs.uiuc.edu/~plop/, 2002.

[Pol01] Pollock, J., "The Big Issue: Interoperability vs. Integration," *Modulant White Paper*, 2001.

[Pos80] Postel, J., *User Datagram Protocol*, RFC 768, Network Information Center, SRI International, Menlo Park, Calif., August 1980.

[Pos85] Postel, J., and J. Reynolds, *File Transfer Protocol (FTP)*, RFC 959, ISI, October 1985.

[Pre00] Pressman, R., *Software Engineering: A Practitioner's Approach*, 5th ed., McGraw-Hill, 2000.

[Pri99] Pritchard, J., *COM and CORBA Side by Side: Architectures*, Strategies, and Implementations, Addison-Wesley, 1999.

[Rap01] Raptis, K., D. Spinellis, S. Katsikas, "Multi-Technology Distributed Objects and their Integration", *Computer Standards & Interfaces*, Elsevier, 2001.

[Rey98] Reynolds, J., *Theories of Programming Languages*, Cambridge University Press, 1998.

[Ric94] Rice, M., S. Seidman, "A Formal Model for Module Interconnection Languages," *IEEE Transactions on Software Engineering*, 1994.

[Ric00] Richard, G., "Service Advertisement and Discovery: Enabling Universal Device Cooperation", *Internet Computing*, IEEE, 2000.

[Rie99] Riemenschneider, R., V. Stavridou, "The Role of Architecture Description Languages in Component-Based Development: The SRI Perspective," *Proceedings of the 1999 International Workshop on Component-Based Software Engineering*, 1999.

[Rob98] Robbins, J., N. Medvidovic, D. Redmiles, D.Rosenblum, "Integrating Architecture Description Languages with a Standard Design Method," *Proceedings of the 1998 International Conference on Software Engineering*, 1998.

[Roy70] Royce, W., "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of Wescon*, 1970. Also available in *Proceedings of the 9th International Conference on Software Engineering*, 1987.

[Rud96] Ruddock, D, B. Dasarathy, "Multithreading Programs: Gudielines for DCE Applications," *IEEE Software*, 1996.

[Ruh01] Ruh, W., Maginnis, F., Brown, J., *Enterprise Application Integration: A Wiley Tech Brief*, John Wiley & Sons, Inc., 2001.

[San81] Sandewall, E., C. Stromberg, and H. Sorensen, "Software Architecture Based on Communicating Residential Environments," *Proceedings of the Fifth International Conference on Software Engineering*, 1981.

[Sap02] SAP Website, http://www.sap.com, SAP. Accessed April 4 2002.

[Sch86] Schmidt, D., *Denotational Semantics: A Methodology for Language Development*, Wm. C. Brown Publishers, 1986.

[Sch00] Schmidt, D., M., Stal, H., Rohnert, F., Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

[Sea99] Seaman, C., "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Transactions on Software Engineering*, 1999.

[Sha86] Shahdad, M., "An Overview of VHDL Language and Technology," *Proceedings of the 23rd ACM/IEEE Conference on Design Automation*, 1986.

[Sha01] Shaikh, A., R. Tewari, M. Agrawal, "On the Effectiveness of DNS-based Server Selection," *Proceedings of IEEE INFOCOM 2001*, 2001.

[Sha89] Shaw, M., "Larger Scale Systems Require Higher-Level Abstractions," *Proceedings of the Fifth International Workshop on Software Specification and Design*, 1989.

[Sha95a] Shaw, M., "Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging," *Proceedings of the 17th International Conference On Software Engineering*, 1995.

[Sha95b] Shaw, M., R. DeLine, D. Klein, T. Ross, D. Young, G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering*, 1995.

[Sha96a] Shaw, M., Garlan, D., *Software Architectures: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[Sha96b] Shaw, M., R. DeLine, G. Zelesnik, "Abstractions and Implementations for Architectural Connections," *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, 1996.

[Sim69] Simon, H., *The Sciences of he Artificial*, 3rd ed., MIT Press, 1996. 1st ed. 1969.

[Sim47] Simon, H., *Administrative Behavior*, 4th Ed., Free Press, 1997. 1st ed. 1947.

[Smi98] Smith, G., J. Gough, C. Szyperski, C., "A Case for Meta-Interworking: Projecting CORBA Meta-Data into COM," *Proceedings of Technology of Object-Oriented Languages*, 1998.

[Sne96] Sneed, H., "Encapsulating Legacy Software for use in Client/Server Systems," *Proceedings of the Third Working Conference on Reverse Engineering*, 1996.

[Sne97] Sneed, H., "Program Interface Reengineering for Wrapping," *Proceedings of the Fourth Working Conference on Reverse Engineering*, 1997.

[Sne98] Sneed, H., R. Majnar, "A Case Study in Software Wrapping," *Proceedings of the International Conference on Software Maintenance*, 1998.

[Sne01] Snell, J., "Implementing Business Processes with the Web Services Architecture," Tutorial, *The IEEE Enterprise Distributed Object Computing 2001 Conference*, 2001.

[Sou01] Sousa, J., D. Garlan, "Formal modeling of the Enterprise Javabeans™ Component Integration Framework," *Information and Software Technology*, 2001.

[Spi89] Spivey, J., *The Z Notation: A Reference Manual*, Series in Computer Science, Prentice Hall, 1989.

[Spi98] Spitznagel, B., D. Garlan, "Architecture-Based Performance Analysis," *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, 1998.

[Sta01] Stavridou, V., B. Dutertre, R. Riemenschneider, H. Saidi, "Intrusion Tolerant Software Architectures," *Proceedings of the DARPA Information Survivability Conference & Exposition*, 2001.

[Ste74] Stevens, W., G. Myers, L. Constantine, "Structured Design," *IBM Systems Journal*, 1974.

[Ste98] Stevens, R., R. Brook, K. Jackson, and S. Arnold, *Systems Engineering: Coping With Complexity*, Prentice-Hall, 1998.

[SvK02] Svenska Kraftnät, *Svensk Elmarknadshandbok*, Svenska Kraftnät, 2002.

[Szy98] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[Tan81] Tanenbaum, A., "Network Protocols," *ACM Computing Surveys*, 1981.

[Tan87] Tanenbaum, A., *Operating Systems: Design and Implementation*, Prentice-Hall, 1987.

[Tan89] Tanenbaum, A., *Computer Networks*, 2nd Ed., Prentice-Hall, 1989.

[Tan95] Tanenbaum, A., *Distributed Operating Systems*, Prentice-Hall, 1995.

[Tho99] Thompson, M., *Technology Audit: MQSeries Product Family*, Butler Direct Ltd, 1999.

[Tid00a] Tidningarnas Telegrambyrå, " Sydkraft köper Nora kommuns energiverksamhet," *Tidningarnas Telegrambyrå*, 2000.

[Tid00b] Tidningarnas Telegrambyrå, " Birka Energi köper Arvika Energi," *Tidningarnas Telegrambyrå*, 2000.

[Tid01] Tidningarnas Telegrambyrå, " Energiaffär i Norrtälje avbryts," *Tidningarnas Telegrambyrå*, 2000.

[Tok90] Tokuda, H., T. Nakajima and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System," *Proceedings of USENIX Mach Workshop*, 1990.

[Uni02] United Nations, *United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport*, http://www.unece.org/trade/untdid/welcome.htm, United Nations [Accessed 11 April, 2002].

[Ves93] Vestal, S., "A Cursory Overview and Comparison of Four Architecture Description Languages," *Technical Report*, Honeywell Technology Center, 1993.

[Ves98] Vestal, S., *MetaH User's Guide*, Honeywell Technology Center, 1998.

[Vli98] Vlissides, J., *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, 1998.

[Wad94] Waddington, D., "Participant Observation." In Cassel, C., G. Symon (ed's), *Qualitative Methods in Organizational Research: A Practical Guide*, Sage Publications, 1994.

[Wal01] Wallnau, K., S. Hissam, R., Seacord, *Building Systems from Commercial Components*, SEI Series in Software Engineering, 2001.

[Wan98] Wang, Y., G. King, A. Dorling , D. Patel , I. Court , G. Staples , M. Ross, "A Worldwide Survey of Base Process Activities towards Software Engineering Process Excellence," *Proceedings of the 1998 International Conference on Software Engineering*, 1998.

[Wir74] Wirth, N., "On the Composition of Well-Structured Programs," *ACM Computing Surveys*, 1974.

[Yak99a] Yakimovich, D., J. Bieman, V. Basili, "Software Architecture Classification for Estimating the Cost of COTS Integration," *Proceedings of the 21st International Conference on Software Engineering*, 1999

[Yak99b] Yakimovich D., G. Travassos V. Basili, "A Classification of Software Component Incompatibilities for COTS Integration," *Proceedings of the 24th Software Engineering Workshop, NASA/Goddard Space Flight Center*, 1999.

[Yin96] Yin, R., *Case Study Research: Design and Methods*, 2nd ed., Sage Publications, 1996.

[Zel96] Zelesnik, G., *The UniCon Language Reference Manual*, http://www-2.cs.cmu.edu/afs/cs/project/vit/ www/unicon/reference-manual/ Reference_Manual_1.html, Carnegie-Mellon University, 1996.