A Brief Survey of Software Architecture

Rikard Land

Mälardalen University
Department of Computer Engineering
Box 883
721 23 Västerås
+46 (0)21 10 70 35

rikard.land@mdh.se

Abstract: *Software of today is becoming larger and more complex. More powerful ways of structuring complexity are consequently required, whether it is about development methodologies, structural programming, naming conventions, configuration management, or, as is discussed in this report, software architecture.*

A software system's architecture can be described as the "blueprint" of a system at the highest level of abstraction, describing the main components and their most important interactions. We discuss in more detail how architectures can be described and the uses of such descriptions. Much research so far has also been dedicated to methods and case studies, to make the research of practical interest. This report describes how the quality of the software can be ensured to a certain degree through informal approaches – not least because an architectural description provides a common understanding around which different stakeholders can meet and discuss a system. Formal approaches are also emerging, and there are a number of formal languages for description of a system's software architecture.

This report presents a brief survey of the field of Software Architecture; both informal and formal approaches are covered and discussed. The report concludes with presenting the author's planned research, aiming at answering how component-based architectures can be designed to handle change.

Keywords: Software Architecture, Architectural Views, Architectural Description Languages, Architectural Analysis.

1. INTRODUCTION

Complex software needs structure. Structure needs to be implemented and documented. Implemented so that the software can be understood and predictable [3,6,25,26,35]; documented so it can be communicated between people having interests in the software [3,6]. However, until recently, there have only been very informal approaches of software structure in large. The lack of an adequate way of describing and communicating structure is one among several problems leading to budget and time overruns and low quality software [35]. The rest of section 1 presents different aspects of these problems in more detail, and the rise of the research field of software architecture.

1.1 Problem Description

Software design documentation often begins with one or several box-and-lines drawings said to describe the system's architecture, as sketched in Figure 1.

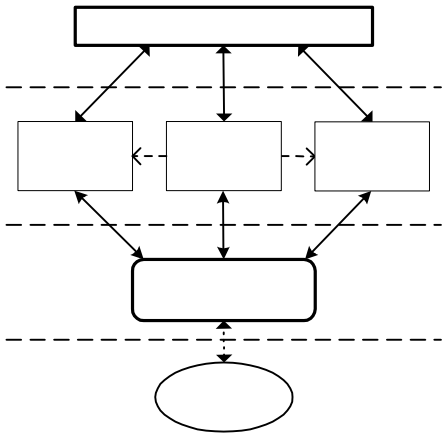

Figure 1. Informal description of a software architecture

Unfortunately, such descriptions are often too informal to be of any real use for others than the author. Do the arrows

represent data flow, control, or some other type of connection? Are the ellipses and rectangles classes, objects, processes, or functions? What do the different shapes, arrowheads, line weights and dashing mean? Why are almost all arrows bi-directional? Are the horizontal dashed lines some sort of border between processes or computers? If the reader of the documentation has the same background and understanding, he will probably understand the description in much the same way as the author of it does. However, the greater the difference between their experience, the higher the probability that the reader will misinterpret the architectural description, or be unable to interpret it at all. And no matter how similar their experience, there will always be room for misunderstanding. If a more formal description were available, there would be less potential sources of misunderstandings; moreover it would be possible to perform many analyses automatically, such as validation and simulation.

In this report, we will explore how the notion of software architecture is formalized. There are also more informal approaches to the art of engineering software architecture, which already has proven successful. Throughout this report we will also stress how consciousness about the notion of software architecture positively affects software development as a whole.

1.2 Definition

It is difficult to capture the term "software architecture" in a definition – what exactly *is* it? From Figure 1, we can however draw some general conclusions about what is usually intended. Software architecture deals with the highest level of a system's design, and a system's architecture can be described as *a set of connected components*. Intuitively, this suits a graphical representation, and accordingly, virtually all formal approaches include box-and-lines representations.

A commonly quoted[1] definition is given in [3]:

> 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.

A more informal description is also given:

> Software architecture concerns the structures of large software systems. The architectural view of a system is an abstract view that distills away details of implementation, algorithm, and data representation and concentrates on the behavior and interaction of 'black-box' components. [3]

[1] Quotations are e.g. found in [6] and [43].

This report will not further elaborate the definitions; we assume that we have enough understanding of the term to be able to continue.

1.3 History

During the history of computers, software has rapidly become more and more complex. A number of approaches have been proposed (and successfully used) to deal with complexity on different levels, such as "structured programming" [11] and Fred Brooks' idea of "conceptual integrity" [8]. The design phase in the software lifecycle has often been split into high-level design and detailed design. Many concepts in the field of ordinary (building) architecture was found to be useful for describing software, thus giving birth to the term "software architecture". The notion of "software architecture" appeared as a useful high-level design solution to part of the complexity problems.

Brooks wrote in the seventies about the importance of architecture, but he intended what we would call the user interface today, however with a touch of today's notion of software architecture [8]. As late as in 1994, Denning and Dargan proposed "software architecture" to be a new software discipline [13], however, their description resembles a development method more than a definition.

Consensus about the term was not achieved until the first half of the nineties. Shaw and Garlan stated in 1996 that "explicit attention to the architecture as a separate level of software design *is relatively recent*" (italics added) and accordingly their book is subtitled "perspectives on an emerging discipline" [35].

1.4 Central Concepts

We will now introduce some central concepts of software architecture and describe these briefly; we will elaborate on more details in subsequent sections.

A software system can be described in many ways – e.g. as a collection of classes or as a collection of processes. Depending on the point of view different characteristics will be discernible. Such different points of views have been identified and named, and are simply called *views*. Many systems are built in a similar way on the architectural level, which makes the introduction of *architectural styles* (or *architectural patterns*) make sense. One important step in formalizing software architecture is to be able to describe architectures in a formal language. A number of *architectural description languages*, ADLs, have been developed. As we have seen, architectures are easily described graphically; many ADLs accordingly have graphical representations and tools.

Two informal methods for evaluating architectures will be described. Both emphasize the evaluation of quality properties. A number of *scenarios* are developed, and the impact of each scenario on the architecture is then

evaluated. These methods can be viewed as means to close the gap between requirements analysis and architectural design. In particular, they intend to give more focus on quality requirements so that the choice of architecture includes conscious choices of expected quality properties.

1.5 The Architectural Community

A number of papers from the Software Engineering Institute (SEI) at Carnegie Mellon University (CMU) were published around 1993, pointing out the direction of the forthcoming research by Gregory Abowd, Robert Allen, Paul Clements, David Garlan, and Mary Shaw [1,19,34]. Shaw and Garlan also wrote a widely referenced book in 1996 [35]. Another influential book is [3], by Len Bass, Clements, and Rick Kazman. Both these books contain thorough surveys and serves as good guides for the novice architect. The two analysis methods we will discuss were described by Kazman and others [25,26].

So far, the people mentioned work at SEI. The architectural research at SEI is organized under the Architecture Based Languages and Environments (ABLE) project, aiming at founding an engineering basis for software architecture. On the project's homepage (see section 7.1) we read: "Components of this research include developing ways to describe and exploit architectural styles, providing tools for practicing software architects, and creating formal foundations for specification and analysis of software architectures and architectural styles."

However there are other research institutions that have contributed to software architectural research. We should note Stanford University with professor David Luckham who developed the architectural language Rapide [31]. Alexander L Wolf at University of Colorado at Boulder and André van der Hoek at University of California, Irvine, has conducted research in relating software architecture to other disciplines, such as versioning, configuration management and product families [37-42]. In Sweden, we can mention Blekinge Institute of Technology (Blekinge tekniska högskola) where Jan Bosch until recently led the research; he is the author of a book on software architecture with focus on product-line approaches [6]. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal of Siemens AG, Germany, made a thorough work in cataloguing architectural patterns in a widely used and referenced book [9].

For thorough surveys of the field, please refer to the two seminal books [3] and [35]. Hofmeister et al presents a "best-practice" approach to the field based on their industrial experience [22]. Wall surveys the field in a technical report, biased at real-time systems [43].

The World-Wide Institute of Software Architects (WWISA) is a nonprofit organization founded to, with their own words from their homepage (see section 7.1), "accelerate the establishment of the profession of software architecture and to provide information and services to software architects and their clients".

1.6 Social Effects

An understanding of software architecture and its importance affects the personal relations within a software development project – hopefully to the better! Quoting Bass et al, architecture "serves as an important communication, reasoning, analysis, and growth tool for systems" [3].

The results of the informal architecture evaluation methods we will describe are explicitly said to be both technical and social [3,25,26]. The analysis "acts as a catalyzing activity on an organization", in the meaning that "participants end up with a better understanding of the architecture" and generates "deeper insights into the trade-offs that are implicit in the architecture", simply because the issue is brought to attention [3].

In this context, it is worth to note that many of the references discuss "stakeholders", and emphasize the importance of letting everybody involved influence the choices made [3,5,6,25,26]. In itself, this is an important step forward to create quality software.

1.7 Outline of the Report

Architectural views will be elaborated in section 2, architectural styles and architectural patterns are presented and discussed in section 3, architectural description languages in section 4, and informal approaches in section 5.

2. ARCHITECTURAL VIEWS

> "As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs, superimposed one upon another." [8]

In other words, you may discover different properties depending on the "angle" from which you view an architecture (see Figure 2). Such a *view* [22,27] "represents a partial aspect of a software architecture that shows specific properties of a software system" [9].

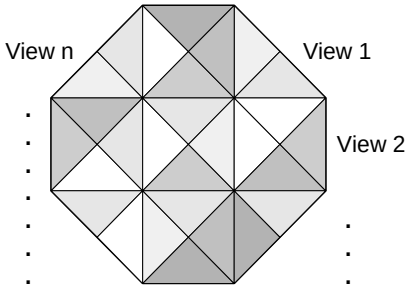

Figure 2. Architectures may be viewed from different positions, and thus bring different properties into light.

In [3], the term "structure", which is what is called "view" in this report, is described similarly:

> Which notion of architecture is the right one, the one whose components are modules or the one whose components are runtime entities such as processes? Obviously they both are. It is an axiom of this book that assuming that the two structures are the same is a fundamental design mistake, since they are optimized to meet completely different criteria. [3]

2.1 The Use of Different Views

There are a number of well-known views, each revealing certain aspects of the architecture being analyzed (on the expense of other characteristics), and an architecture should be described in several relevant architectural views. With "relevant" we mean views that can unveil the properties that are of interest. No more effort should be put into elaborating views than can be ratified by the usage of them, e.g. for analysis and understanding.

Views can be described graphically as a number of components and connections, but the semantics of these artifacts differ between views. For example, in a run-time view of an object-oriented system, we may have the component type "object" and the connector type "message" to our disposal while a design-time view might include "classes" and "inheritance" – if objects and classes are mixed within a description, it would loose all sense. See Figure 3.

Different views are furthermore suited for different analyses. A view describing the run-time objects may be used to estimate the system's performance and find bottlenecks, and a class view to e.g. estimate the maintainability from the number of dependencies between classes. Different software domains often need specific views, such as a temporal view for real-time architectures.

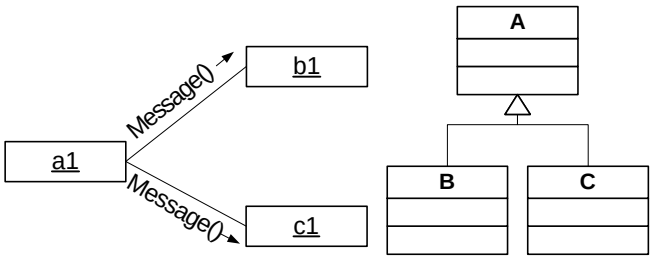

Figure 3. Two views of the same system: a run-time view (to the left) and a design-time view.

2.2 Lists of Views

For the purpose of this report, it is enough to state that it is possible to view a piece of software from different views, and that different views reveal different properties (and are thus suited for different analyses). Three different lists of views are presented below, without further comments.

In the first list of [9], views are called "architectures".

- "Conceptual architecture: components, connectors".
- "Module architecture: subsystems, modules, exports, imports".
- "Code architecture: files, directories, libraries, includes".
- "Execution architecture: tasks, threads, processes".

In the second list of [9], there are four views.

- "Logical view: the design's object model, or a corresponding model such as an entity relationship diagram."
- "Process view: concurrency and synchronization aspects."
- "Physical view: the mapping of the software onto the hardware and its distributed aspects."
- "Development view: the software's static organization in its development environment."

In [3], nine views are listed, called "structures".

- "Module structure. The units are work assignments".
- "Conceptual, or logical, structure. The units are abstractions of the system's functional requirements".
- "Process structure, or coordination, structure. This view […] deals with the dynamic aspects of a running system. The units are processes or threads".
- "Physical structure. This view shows the mapping of software onto hardware".
- "Uses structure. The units are procedures or modules; they are linked by the assumes-the-correct-presence-of relation".
- "Calls structure. The unit are usually (sub)procedures; they are related by the calls or invokes relation".
- "Data flow. Units are programs or modules; the relation is may-send-data-to."
- "Control flow. Units are programs, modules, or system states; the relation is becomes-active-after."
- "Class structure. Units are objects; the relation is inherits-from or is-an-instance-of."

2.3 Graphical Representation

Unfortunately, none of the sources presented any formalization or example of what the views look like graphically. We can just note that the Universal Modeling Language, UML [5], has been widely spread and can be

used for describing architectural views (see section 4.7). In this report, we use UML when we find it appropriate.

3. STYLES AND PATTERNS

3.1 Styles

An *architectural style* is an established pattern of components with a name, such as a *client-server* architecture. A common knowledge of styles is useful when discussing a system, so that a statement like "X is a client-server system" will readily give a common understanding among people involved, and when detailed implementation decisions has to be made, the programmer will choose a solution that conforms to the style.

Often architectural styles are domain-specific (well, more or less), as will be described for some styles below. A style typically addresses specific problems, often quality-related:

> When we have models of quality attributes that we believe in, we can annotate architectural styles with their prototypical behavior with respect to quality attributes. We can then talk about performance styles (such as priority-based preemptive scheduling) or modifiability styles (such as layering) or reliability styles (such as analytic redundancy) and then discuss the ways in which these styles can be composed. [3]

The most commonly known styles are explained briefly below. Please note that some elements in the figures describing the architectures below look similar but have different semantics; arrows may e.g. denote data flow, function calls or some other type of connection.

3.1.1 Pipe-and-Filter

In a *pipe-and-filter* system the data flow in the system is in focus [3,34,35,43]. There are a number of computational components, where output from one component forms the input to the next. A typical example is the use of Unix pipes. See Figure 4, where each box is a processing unit, and an arrow represents data flow[2].

In its purest form, the different components are completely separated (they share no data or state), and may start processing as soon as input starts arriving. A close relative of this architecture is the batch sequencing architecture, where each step finishes before the next start.

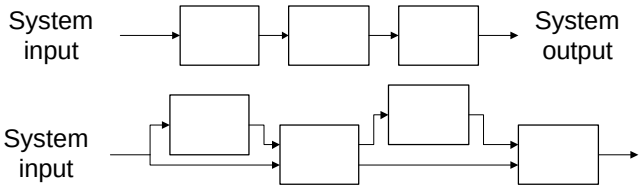

Figure 4. Two pipe-and filter systems, one very simple and the second a little more complicated.

This style fits a program analyzing and formatting text or data, but is not so useful for an interactive system. Because data is copied (at least in the pure pipe-and-filter form) from outputs to inputs, performance is generally decreased.

3.1.2 Object-Oriented Architecture

With an *object-oriented* architecture, the focus is on the different items in the system, modeled as objects, classes etc. Object orientation is one of the most widely spread architectural styles, both in education, industrial practice and science.

It can be discussed whether object-orientation is an architectural style or belongs to lower levels of design. Object-orientation as an architectural style is discussed in [3,6,35].

3.1.3 Layered Architecture

With a *layered* (or onion) architecture, focus is laid on the different abstraction levels in a system, such as the software in a personal computer [3,6,34,35,43]. A stack of boxes or a number of concentric circles is often used to represent a layered architecture graphically (see Figure 5).

In its pure form, communications between the different layers must only occur in the interfaces between two adjacent layers. The style's major drawbacks are that it is not always easy to identify the appropriate abstraction levels. It might also be the case that the system must communicate in a more complex way than is implicated by the layering, due to performance considerations.

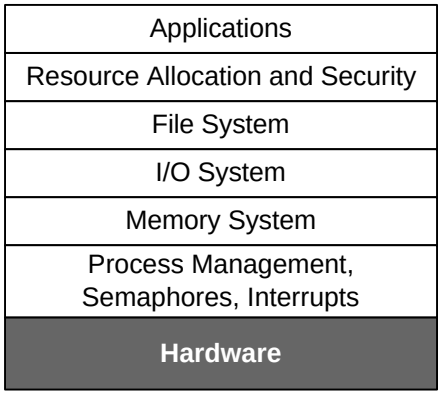

Figure 5. The layered architecture of a personal computer (the layers according to [14]).

[2] We have avoided using UML on purpose, since a pipe-and-filter architecture is only a logical abstraction, while UML symbols would imply how its implementation. It could be implemented e.g. as Unix processes and pipes, threads with shared buffers, or function calls with data structures as parameters.

3.1.4 Blackboard Architecture

A *blackboard* (or *repository*) architecture draws the attention to the data in the system [3,34,35,43]. There is a central data store, the *blackboard*, and *agents* writing and reading data. The agents may be implicitly invoked when data changes, or explicitly by some sort of external action such as a user command. A blackboard architecture is described in Figure 6, where the central data store is represented by the rectangle, agents by the ellipses, and the arrows denote requests to read and write data.

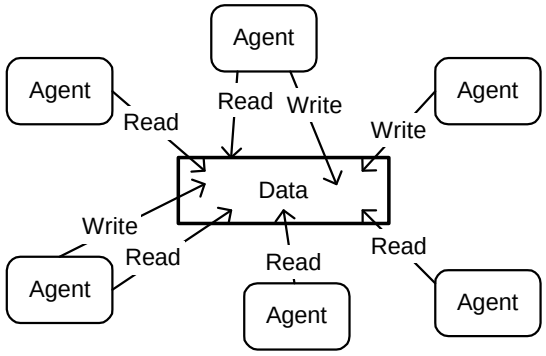

Figure 6. A blackboard (repository) architecture.

A database can easily be described by the blackboard architectural style, where the blackboard itself of course is the data in the database. Examples of agents are client applications, database triggers (small pieces of program code that are executed automatically when data changes), and administration tools.

3.1.5 Client-Server Architecture

A *client-server* architecture focuses on services different clients want to perform [3,6,34,35,43]. This architecture is especially fit when the hardware is organized as a number of local computers (e.g. personal workstations) and one central resource such as a file tree, database, or a cluster of powerful central calculation computers. See Figure 7.

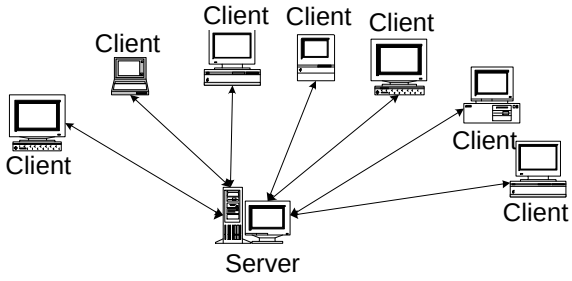

Figure 7. A view of a (hardware) client-server system.

In a software client-server system, there may be several clients in one computer, and even the server can be running on the same computer.

This is a way of describing a multi-user database, on a different abstraction level than that of the blackboard architecture.

3.1.6 Process Control

Real-world systems often control a physical reality, such as control systems in a power plant. There are a number of software paradigms for *process control* [35,43]. The significant properties are that the software takes its input from sensors (such as a flow sensor), and perform control actions (such as closing a valve). The control loop may be of feedback or feed-forward type.

3.1.7 State Machine

When designing a *state machine* architecture, the states of the program can be in are identified, together with legal transitions between them [35].

State machines are well known to mathematicians, and can be thoroughly investigated and validated regarding loops, illegal states etc, which makes this style common in safety-critical systems. State machines are particularly well suited for graphical description (see Figure 8).

Here it is appropriate to ask whether this is actually a style or a view of an architecture. Maybe it is appropriate to talk about the state machine as a style when the clearest description of a system is as a set of states and transitions, and as a view when trying to discern states and transitions in an existing system – the border between styles and views is not as sharp as one might believe! In the case of using a state machine as a style, it should be possible to also describe the system with another style – describing how the state machine is implemented. Such *heterogeneous styles* are discussed in the following section.

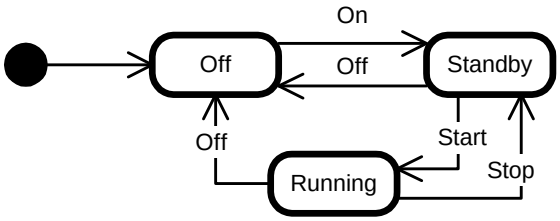

Figure 8. A state machine.

3.2 Heterogeneous Architectural Styles

Reality is more complicated than what might have been implied above. Many systems can be described with several styles simultaneously. Shaw and Clements make this property explicit:

> We describe the styles in their pure forms, although they seldom occur that way. Real systems hybridize and amalgamate the pure styles, with the architect choosing useful aspects from several in order to accomplish the task at hand. Our classification does not impede this heterogeneity, but rather enhances the selection and

> blending process by making stylistic properties explicit. [34]

Some systems are just modifications of an architectural style; others combine them, possibly on different abstraction levels. Consider a system implementing a pipe-and-filter system, where each filter abstracts the operating system via a portability layer, to facilitate portability. The lowest levels of abstraction could maybe be regarded as technique rather than architecture (as was discussed, object orientation can e.g. be thought of as an implementation technique).

This issue is also discussed by Bass et al who state that systems "are seldom built from a single style, and we say that such systems are heterogeneous" [3]. Three kinds of heterogeneity are identified:

- "Locationally heterogeneous" – different runtime parts use different styles.
- "Hierarchically heterogeneous" – different components in a system of one style may be structured according to another style, as our client-server example above.
- "Simultaneously heterogeneous" – several styles serve as a description of the same system (as we saw, a multi-user database can be viewed as both a blackboard and a client-server architecture). This heterogeneity "recognizes that styles do not partition software architectures into nonoverlapping, clean categories" [3].

3.3 Architectural Patterns

When creating an architecture, you should of course use your knowledge and experience. However, creating a "good" architecture is difficult and costly if the developers have to gather knowledge through trial-and-error, especially if they are not even conscious about the notion of "software architecture". Moreover, it is difficult to deduce general knowledge from experience.

Buschmann et al collected such *architectural patterns*, analyzed them, and made them available [9]. They adopt a three-part schema underlying patterns consisting of a *problem* within a *context*, and a *solution*. We believe that every software architect should be armed with a set of patterns, and recognize contexts and problems where there exists a proven solution. It should be noted that there are patterns for all levels of abstraction; Buschmann et al divides patterns into architectural patterns, design patterns and idioms [9] (we are in this report interested solely in architectural patterns[3]).

The knowledge of successful architectural patterns does of course not exclude the need of an architectural evaluation. A pattern is merely a pattern, and is only part of a system's architecture. Moreover, there may be circumstances making a certain pattern unsuitable. For example, [9] presents two patterns for interactive applications, each having different pros and cons: the Model-View-Controller and the Presentation-Abstraction-Control patterns.

We can note that Buschmann et al speak about e.g. the "Pipes and Filters" architectural pattern, so their notion of "pattern" somewhat overlaps what was called "style" in section 3.1. It seems as there are two names for the same thing: a system using the pipes-and-filters *pattern* can always be said to conform to the pipe-and-filter *style*; the opposite is equally true.

4. ARCHITECTURE DESCRIPTION LANGUAGES

As we have seen, architectures can be described roughly as a set of *components* connected by *connectors*. Depending on the application domain and the view, the descriptions can contain other entities as well. A number of formal languages have been developed to allow for formal and unambiguous descriptions. Such an *Architecture Description Language* (ADL) often has a graphical representation.

An ADL defines the basic elements to be used in an architectural description. Different ADLs are designed to meet slightly different criteria, and have somewhat different underlying concepts (compare with the abundance of programming languages – they are designed to be used for different types of programming, and it would be naïve to believe that one language is enough for all purposes). An ADL specifies a well-defined syntax and some semantics, making it possible to combine the elements into meaningful structures. The advantages of describing an architecture using a formal ADL are several:

- Some formal analyses can be performed, such as checking whether an architectural description is consistent and complete[4].

[3] Examples of design patterns are "Whole-Part" and "Publisher-Subscriber", while "Counted Pointer" is an example of an idiom [9]. Gamma et al collected a large number of design patterns into *the* book on design patterns [16].

[4] Allen provides a good explanation of these notions: "Informally, consistency means that the description makes sense; that different parts of the description do not contradict each other. Completeness is the property that a description contains enough information to perform an analysis; that the description does not omit details necessary to show a certain fact or to make a guarantee. Thus, completeness is *with respect to* a particular analysis or property." [2]

- The architectural design can be unambiguously understood and communicated between the participants of a software project.
- We may also hope for a means to bridge the gap between architectural design and program code by transformation of a formal architectural description to a programming language.

The rest of this chapter describes the most known ADLs very briefly. The first four are ADLs in their own right, the next, Acme, identifies the least common denominator of other ADLs, ADML builds on Acme; we also discuss the Universal Modeling language (UML) as an ADL. Most of these, and others, have been compared by Medvidovic and Taylor together within their framework for classifying ADLs [32].

4.1 Rapide

The Rapide language [31], developed by David Luckham at Stanford University, has quite a long history. It builds on the notion of partial ordered sets, and thus introduces quite new (but seemingly powerful) programming constructs; it is also very useful in that it is both an architectural description language and an executable programming language. A number of tools have been built, e.g. for performing static analysis and for simulation.

The combination of an architectural description language with formal and informal methods to analyze it, as well as actually being able to compile and execute it seems to be very powerful.

4.2 UniCon

UniCon [35], developed by Mary Shaw at Carnegie Mellon University, is "an architectural-description language intended to aid designers in defining software architectures in terms of abstractions that they find useful".

UniCon is designed to make "a smooth transition to code" [35], through a very generous type mechanism. Components and connectors can be of types that are built-in in a programming language (e.g. function call), or be of more complex types, user-defined as code templates, code generators or informal guidelines.

4.3 Aesop

Aesop [17], developed by David Garlan at Carnegie Mellon University, is addressing the problem of style reuse. With Aesop, it is possible to define styles and use them when constructing an actual system.

Aesop provides a generic toolkit and communication infrastructure that users can customize with architectural style descriptions and a set of tools that they would like to use for architectural analysis. Tools that have been integrated with Aesop styles include: cycle detectors, type consistency verifiers, formal communication protocol analyzers, C-code generators, compilers, structured language editors, and rate-monotonic analysis tools.

4.4 Wright

Wright [2] was developed at Carnegie Mellon University and forms a basis for Robert Allen's research. It is a formal language including the following elements: *components* with *ports*, *connectors* with *roles*, and *glue* to attach roles to ports. Architectural styles can be formalized in the language with predicates, thus allowing for static checks to determine the consistency and completeness of an architecture.

4.5 Acme

Acme [18], developed by a team at Carnegie Mellon University, can be seen as a second-generation ADL, in that its intention is to identify a kind of least common denominator for ADLs. It is thus not designed to be a new or competing language, but rather to be an interchange format between other languages and tools, and also allow for use of general tools. One could devise one tool searching for illegal cycles, and use it for descriptions in any ADLs, as long as there exist translation functionality between that ADL and Acme.

Acme defines 7 basic element types: components, connectors, systems, ports, roles, representations, and rep-maps (representation maps). See Figure 9 for a description of the five most important (figure slightly modified version from [18]). Acme's textual representation of a small architecture is found in Figure 10 (after [18]).

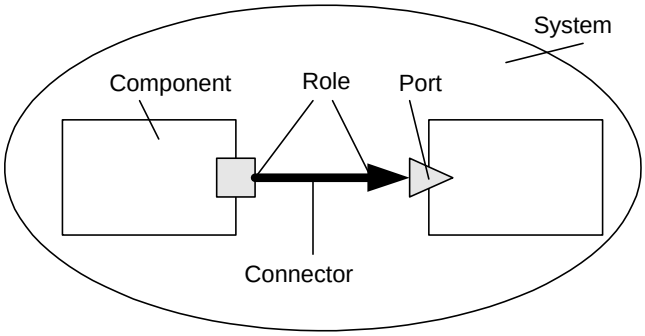

Figure 9. Elements of an Acme description.

```
System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc = { Roles {caller, callee} }
  Attachments : {
    client.sendRequest to rpc.caller ;
    server.receiveRequest to rpc.callee
  }
}
```

Figure 10. An Acme description of a small architecture.

As was implied above, the success of Acme is highly dependent on the existence of tools and translators. The research team at SEI behind Acme has constructed the

graphical architectural editor AcmeStudio. A screen snapshot is presented in Figure 11 – the architectural description is in the top right quadrant; to the left and beneath it are different types of browsers for e.g. components, connectors, properties, and rep-maps. Translators between UniCon, Aesop, Wright, and Rapide have also been constructed [18].

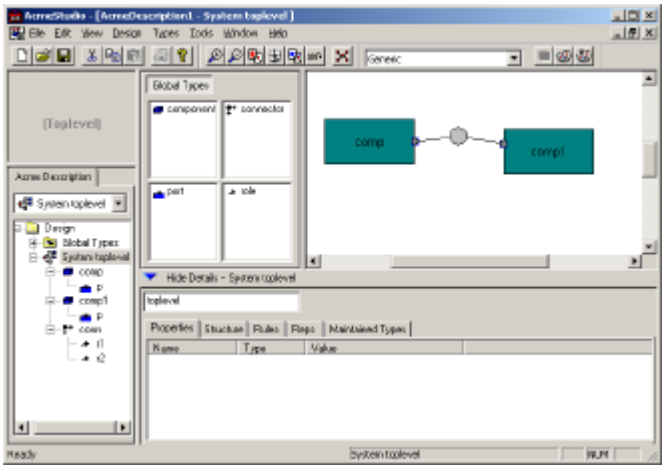

Figure 11. A snapshot of AcmeStudio.

However, voices doubting Acmes universality can also be heard, stating that "its growth into an all-encompassing mediating service never has taken place […] Acme should probably be considered as a separate architecture description language altogether" [12].

4.6 ADML

The Open Group found room for improvement of Acme and have defined the Architecture Description Markup Language (ADML). At ADML's homepage (see section 7.1) we find the following description of ADML: "ADML adds to ACME a standardized representation (parsable by ordinary XML parsers), the ability to define links to objects outside the architecture […], straightforward ability to interface with commercial repositories, and transparent extensibility".

4.7 UML

In this context, it is appropriate to mention the Universal Modeling Language (UML) [5]. It has been a de facto-standard for the design and description of object-oriented systems, and includes many of the artifacts needed for architectural descriptions – processes, nodes, views etc. As a consequence, software architecture and UML are often mentioned together [5,22].

For informal descriptions, UML is very suited just because it is a widely understood standard. It however lacks the full strength needed for an adequate architectural description. As an example, in UML the connectors are language entities such as function calls, while an ADL would contain such things as client-server connections and protocols of interaction. UML is intended to be an object-oriented modeling language, while an architecture could very well be implemented in other types of programming languages – but once again UML do include language-independent artifacts such as processes. Hofmeister et al [22] use UML to describe software architectures, and say in the introduction that "some of our architecture concepts are not directly supported by existing UML elements […] All in all, we think the benefits to be gained by using a standardized, well-understood notation outweigh the drawbacks" [22]. It is possible to within UML define new elements as meta-models, and it might be possible to extend the common UML language with such new concepts to build an ADL.

As was said, UML can today be used to give a good informal description of an architecture by "bending" the notation, but this is not the intention of a fully-fledged ADL, especially when it comes to formal analyses of an architecture.

5. INFORMAL ANALYSIS METHODS

We will continue by describing informal approaches to the field. We shortly present a number of case studies and two methods of performing informal architectural analysis.

5.1 Analysis Methods

Researchers at the Software Engineering Institute have developed two informal architecture analysis methods. The Software Architecture Analysis Method (SAAM) is a method used for analyzing an architecture using scenarios [25]; the Architecture Tradeoff Analysis Method (ATAM) is a development of SAAM, introducing the notion of "tradeoff points" [26].

Common for both is the methodology they implement – quality requirements are evaluated and agreed upon evolutionary as architectures are designed and evaluated. They therefore resemble the "spiral" software lifecycle model where requirements, design and implementation are refined in a cyclic approach [4]. They are not designed for any specific quality attributes or software metrics, but rather to serve as a framework leading the analyst to focus on the right questions at the right time. *Any* quality attribute can be analyzed with these methods; examples are modifiability [25,28], cost [26,28], availability [26], and performance [24,28].

5.1.1 SAAM

The Software Architecture Analysis Method (SAAM) uses scenarios to evaluate quality properties of an architecture [25]. SAAM is applied early in the development cycle, and gives the architect the possibility to choose an architecture with an acceptable tradeoff between quality attributes. With this method, architectures are informally compared through

the use of *scenarios*, such as the use case "the user starts a simulation" or the change scenario "the system is extended with functionality to compare binary output files". Of course, to be able to compare architectures, they must be described in a consistent and understandable way – thus some sort of ADL must form the basis of the analysis.

Of course SAAM cannot give any absolute measurements on quality properties, but should rather be used to compare candidate architectures. The results are of the sort "system *X* is more maintainable than system *Y* with respect to change scenarios *A*, *B*, and *C*, but less maintainable with respect to scenarios *D* and *E*". These results thus form a basis for project decisions where priorities as short-term and long-term costs, time-to-market, future reusability are weighed against each other. For the outcome of the analysis to be reliable, it is crucial that the selected scenarios are indeed representative for actual future scenarios. SAAM emphasizes the participation of all stakeholders of the system, i.e. project managers, users, developers etc.

SAAM was used in the M.Sc. thesis made by the author [28], as part of a commercial development project, and our experience is that it fulfils its expectations. A tool prototype for aiding in SAAM analysis (as well as aiding in documenting architecture at all), SAAMtool, has been built [23].

5.1.2 ATAM

The Architecture Tradeoff Analysis Method (ATAM) is a relative of SAAM, in which the importance of tradeoffs has been noticed [26]. A tradeoff is the decision needed to choose between alternative architectures, to arrive at a set of properties that are acceptable; it is naïve to believe that architectural design aims at finding *the* architecture, meaning the cheapest to build *and* the most resource-effective *and* the most portable *and* the most reusable.

> It is obvious that one cannot maximize all quality attributes. This is the case in any engineering discipline. […] The strongest bridge is not the lightest, quickest to erect, or cheapest. The fastest, best-handling car doesn't carry large amounts of cargo and is not fuel efficient. The best-tasting dessert is never the lowest in calories. [3]

Many such quality attributes are orthogonal, meaning that improving one deteriorates another. The engineering approach is thus to try and find an *acceptable* tradeoff, considering not only the technical aspects of the software, but include all related concerns such as management and financial issues. ATAM supports projects when discussing the system and agreeing upon an acceptable tradeoff by introducing the notion of *tradeoff points*:

> Once the architectural sensitivity points have been determined, finding tradeoff points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The availability of that architecture might also vary directly with the number of servers. However, the security of the system might vary inversely with the number of servers (because the system contains more potential points of attack). The number of servers, then, is a tradeoff point with respect to this architecture. It is an element, potentially one of many, where architectural tradeoffs will be made, consciously or unconsciously. [26]

5.2 Case Studies

Since the field of software architecture is to a large extent informal, much of the work done so far has been to support the theories with case studies, which is also reflected in the titles of much of the work. In "Software Architecture in Practice", Bass et al describe aircraft navigation computers, the World Wide Web (WWW), the design of CORBA, air traffic control, flight simulation, and product line development [3]. Garlan and Shaw investigate mobile robotics and digital oscilloscopes [35], while Bosch uses product lines, fire alarm, measurement systems, and a dialysis system as examples [6]. Hermansson et al report on how the architecture of a telecommunication system was redesigned [21]. Bowman et al describe how they arrived at an architectural description of Linux by reengineering the code [7]. Hofmeister et al build their book "Applied Software Architecture" heavily on industrial practice gathered from throughout the Siemens corporation [22].

Bass et al describes how SAAM was used to evaluate two competing financial management systems, and a revision control system [3]. Kazman et al used ATAM in a case study of a battlefield control system [24].

6. FUTURE RESEARCH

We are still in the middle of the development of this exciting field. As the field matures, we can expect many of the different research branches to converge, and more formal definitions and methods to emerge. Informal aspects of software architecture should not be neglected, however. Many of the practical problems in the industry would be at least partly solved merely by having a common vocabulary and understanding of these issues. Therefore it is important that research results are made accessible for, and widely spread in, industrial practice.

Although work has been conducted in the following areas, much remains to be done:

- For ADLs and views, major challenges are:
  - How can styles be defined in an ADL – what makes a style a style [1]?

  - How can architectures be automatically verified and validated [2]?
  - Can more patterns and styles be formalized in an ADL, as a context-problem-solution triplet [9] that can be found during analysis through some sort of knowledge base or expert system?
  - How can the relation between different views be described? For example, exactly how does the objects in a run-time view relate to the classes in a design-time view?
  - How can UML be extended to allow for a standardized way of describing software architecture?
  - The field of Aspect-Oriented Programming (AOP) focuses on separation of concerns [15], and shows similarities with the concept of architectural views. What do AOP and Software Architecture have in common? How can these fields and techniques be combined?
- In connection to development processes, major challenges are:
  - How is it possible to maintain a connection between an architectural description and the program code implementing it? This would include several views[5].
  - How is it possible to maintain the connection between a system's requirements and its architectural design, to make it possible to see how the architecture fulfills the requirements?
  - How can ATAM be extended to make it possible to find "tradeoff points" formally or through simulation rather than scenarios?
  - How can SAAM and ATAM be extended with a formal description of scenarios, making it possible to automatically analyze a larger number of architectures and scenarios?
  - More patterns should be catalogued (such as has been done by Buschmann et al [9]).
- Software architecture and Component Based Software Engineering are two sides of the same coin:
  - How can these fields be combined?
  - Can the properties of an architecture be calculated from the properties of its components? Which properties? How?
- What happens with a system architecture when its requirements, environment etc. changes?
  - What types of changes can a particular architecture tolerate? To how high degree?
  - Put another way – how can a system's architecture be designed to tolerate change?
  - What happens with the architecture when a system ages [33]? How can the architecture be made to survive longer than the components implementing it?
  - How can a system's architecture be changed in runtime (preferably automatically)?

Now we have arrived at the topics I will focus on in my research. The rest of section 6 is devoted to describing how I intend to reach the combined goals of contributing to the research community in the area of software architecture and achieving a Ph.D. degree.

6.1 Research Plan

My research will focus on the cross-section of three areas: *Software Architecture*, *Component Based Software Engineering*, and *change* (see Figure 12). In my Ph.D. thesis, I hope to be able to answer *how component-based architectures can be designed to handle change*.

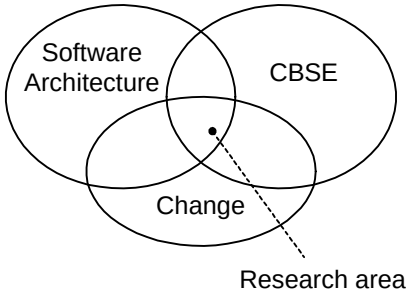

Figure 12. The cross-section of three research fields.

I will study existing research as described below, and try to combine findings and techniques from these fields. My licentiate thesis (half-way to the Ph.D. thesis) will contain a thorough overview of the three areas, a thesis (yet to be formulated), and a validation of it through argumentation and case studies. Case studies can be done in cooperation with Compfab and Westinghouse (see section 6.4); our department, IDt, also has contact with other large software-intensive companies like ABB, Bombardier Transportation, Ericsson, and Volvo.

[5] In a real system with many thousands of lines of code it is hard to see how the architecture is implemented in code – which lines implement the architecture and which lines implement low-level algorithms in a real system? Probably it is impossible to make such a division altogether, making this question even harder to answer.

6.2 Component Based Software Engineering

The research area of *Component Based Software Engineering* (CBSE) [36] is closely related to the field of software architecture – architecture is the way components are composed to a structure, while in CBSE the focus in on the components (and how they can be integrated). In this context, a component is basically a binary executable. The governing idea of CBSE is that when requirements for a system has been formulated, instead of designing, implementing, and testing the components, pre-built and already tested commercial components are found and used. There are in CBSE basically two ways of building software – either building reusable components or combining existing components into systems. See Figure 13.

This will of course affect the way in which the system's architecture is designed – somehow the architecture must take into account that the components cannot be assumed to conform exactly to the architect's wishes (as would be the case if the entire system was developed in-house). The ultimate goal of CBSE is to found a conceptual base to make it possible to find pre-built binary components, assemble them with a minimum of effort, and predict the properties of component assemblies. To make this possible, there will be standards and certification mechanisms; certified components will have a basic set of functionality and be of known quality. The components in question are commercially available, and the competing edge for a component-developing company will be to include more functionality than the standards require, or excel in quality. This touches other research areas, such as how the semantics of a component can be described and assessed, and how to measure quality attributes in an unambiguous way.

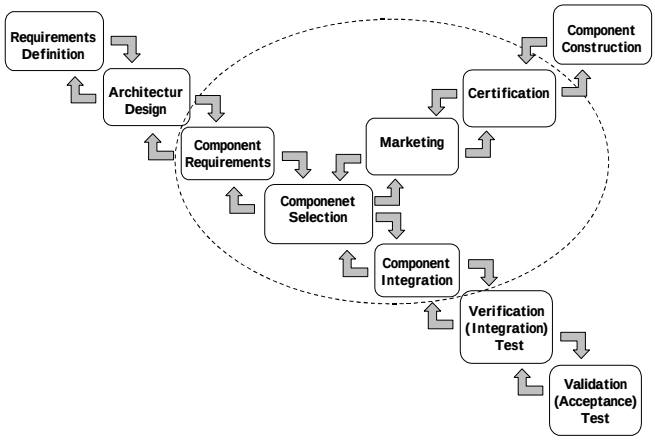

Figure 13. A waterfall sequential representation of the main tasks in CBSE.

Research on CBSE has been made at our department, Department of Computer Engineering (Institutionen för Datateknik, IDt) [10,30], and I will contribute to this research through focusing on the architectural aspects of CBSE.

6.3 Change

Change is a very broad term when associated with software – it can mean e.g. source code modifications, administrative changes to an installed system, or changes in runtime. There are also a number of different types of causes of changes: requirements are modified or added, errors need correction, the quality needs to be improved, technology changes, other systems in the environment change, parts of it is to be reused etc. The question of change can be viewed either on the architectural level [37,39-42] or from a component-based point of view [30]. Part of my research will be to combine these points of view.

There is a trend that more and more of a software system should be variable, so that it is a system administrator who should carry out needed changes rather than a programmer. It is further complicated by the fact that the changes should preferably be carried out in runtime (the system should never be "down"), and preferably by the system itself (without need for a human administrator). The program should therefore be prepared to many types of changes. We can discern a "scale of triviality": names of files, servers, etc. should never be hard-coded but changeable via e.g. initialization files; the number of services could be programmed to response to e.g. new load balancing strategy; most difficult to handle is of course questions about unforeseen changes (e.g. of requirements) and to how high degree a system can be designed for dynamic changes.

To handle different types of change, there are today research, practice, and tools for e.g. revision control, configuration management, testing, and software development processes. Some work has been done to integrate these ideas with the fields of software architecture and CBSE [30,37,39-42]. However, there is still much more left to do, and I hope to be able to develop this area. In particular, I will investigate which changes during runtime an architecture can be designed for and whether there are any configuration management techniques that can be applied to runtime changes.

6.4 Concrete Plans

To arrive at this goal, I have to study some specific areas more in detail. I learnt much from the field of software architecture during my M.Sc. thesis project work, but will study the related areas I will find more thoroughly and keep up with the state-of-the-art; I will study the field of CBSE, through self-studies and two courses: "Component-Based Software Engineering" and "Component Technologies"; I will also study techniques to handle change through literature search – I will e.g. study the ideas of configuration management, and will in particular study the work of André van der Hoek [37,39-42].

I intend to write and publish some papers. I have written a paper about the work done in my M.Sc. thesis work, performed at Westinghouse) and what can be learnt from that [29]; I will also elaborate on some ideas I got then and hopefully be able to draw some general conclusions on how architectural views can be related. I have written one paper with Erik Gyllenswärd and Mladen Kap about "Information Organizer", a product implementing a general architecture for information systems [20] (see the Compfab URL, section 7.1).

An important part of research education is to learn to cooperate. I will collaborate with my colleagues at the Software Engineering group here at IDt to connect my architectural research with CBSE. I have already mentioned that I have collaborated with Erik Gyllenswärd and Mladen Kap at Compfab around their product "Information Organizer" [20], and I plan to keep contact throughout my research and validate my findings using their framework and product. I am employed by Westinghouse, where I performed my M.Sc. thesis work [28] (but I am currently on leave of absence for my Ph.D. studies); they have announced interest in my research and I will hopefully be able to perform some case study in collaboration with them [29]. Anders Wall and Joakim Fröberg at IDt are studying software architecture in relation to real-time systems and are potential collaborators [43].

7. CONCLUSION

We have studied the notion of software architecture, and discussed how to describe and analyze it. We gave a historical background and described where and by whom major research has been performed, and gave references to important literature. *Architectural views* are basically a set of component types and connection types, with which certain aspects of an architecture can be described. The architecture of a piece of software can be described formally in an *Architectural Description Language* (*ADL*), and we presented some ADLs shortly. We also said that many system architectures conform to well-known *architectural styles* or *patterns* such as the *pipe-and-filter* and *client-server* styles, and described some of these. We presented two methods for informal analysis of architectures: the Software Architecture Analysis Method, SAAM, and the Architecture Tradeoff Analysis Method, ATAM. As a recurring theme, we have also argued for the opinion that a deeper understanding of software architecture will increase the quality of both software development and the software itself. We have thus showed that through the notion of software architecture, we have presented a powerful means of defining, formalizing, describing, and enforcing *structure* to software systems.

Current research challenges and the focus of my research were presented, together with a description of how I intend to reach the joint goals of contributing to the field and achieve a Ph.D. degree by describing *how component-based architectures can be designed to handle change*.

7.1 Useful URLs

Below is a list of URLs to useful web pages, as of the date of the release of this report (12-Feb-02).

ABLE: http://www.cs.cmu.edu/~able/

ACME: http://www.cs.cmu.edu/~acme/

ADML: http://www.opengroup.org/onlinepubs/009009899/

Aesop: http://www.cs.cmu.edu/~able/aesop/

Compfab: http://www.compfab.se/

Rapide: http://poset.stanford.edu/rapide/

Wright: http://www.cs.cmu.edu/~able/wright/

WWISA: http://www.wwisa.org/

UML: http://www.uml.org/

7.2 Credits

All research results presented in this report are due to other people's work as indicated. Christina Wallin made Figure 13. Anders Wall read an earlier draft of this report and gave useful suggestions of improvements.

REFERENCES

[1] Abowd G., Allen R., and Garlan D., "Using Style to Understand Descriptions of Software Architecture", In *Proceedings of The First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993.

[2] Allen R. "A Formal Approach to Software Architecture" Ph.D. Thesis Carnegie Mellon University, Technical Report Number: CMU-CS-97-144 1997

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

[4] Boehm, B., Spiral Development: Experience, Principles and Refinements, report Special Report CMU/SEI-2000-SR-008, Carnegie Mellon Software Engineering Institute, 2000.

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

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

[7] Bowman I. T., Holt R. C., and Brewster N. V., "Linux as a Case Study: Its Extracted Software Architecture", In *Proceedings of Proceedings 21st International Conference on Software Engineering (ICSE)*, 1999.

[8] Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition*, Addison-Wesley Longman, 1995.

[9] Bushmann F., Meunier R., Rohnert H., Sommerlad P., and Stal M., *Pattern-Oriented Software Architecture - A System of Patterns*, John Wiley & Sons, 1996.

[10] Crnkovic Ivica and Larsson M., Challenges of Component-based Development, *Journal of Software Systems*, volume 2001, issue December, 2001.

[11] Dahl O.-J., Dijkstra E. W., and Hoare C. A., *Structured Programming*, Academic Press, 1972.

[12] Dashofy E. M. and van der Hoek A., "Representing Product Family Architectures in an Extensible Architecture Description Language", In *Proceedings of The International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain*, 2001.

[13] Denning P.J. and Dargan P. A., A discipline of software architecture, *ACM Interactions*, volume 1, issue 1, 1994.

[14] Eklund S., *Programkonstruktion och projekthantering*, Studentlitteratur, 1993.

[15] Elrad T., Filman R. E., and Bader A., Aspect-oriented programming: Introduction, *Communications of the ACM*, volume 44, issue 10, 2001.

[16] Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[17] Garlan D., Allen R., and Ockerbloom J., "Exploiting Style in Architectural Design Environments", In *Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineerng*, 1994.

[18] Garlan D., Monroe R.T., and Wile D., Acme: Architectural Description of Component-Based Systems, in Foundations of Component-Based Systems, editors, Leavens G.T. and Sitarman M., Cambridge University Press, 2000.

[19] Garlan D. and Shaw M., An Introduction to Software Architecture, *Advances in Software Engineering and Knowledge Engineering*, volume I, 1993.

[20] Gyllenswärd E., Kap M., and Land R., "Information Organizer - A Comprehensive View on Reuse", In *Proceedings of Proceedings of International Conference on Enterprise Information Systems (to appear)*, 2002.

[21] Hermansson H., Johansson M., and Lundberg L., "A Distributed Component Architecture for a Large Telecommunication Application", In *Proceedings of The Asia-Pacific Software Engineering Conference (APSEC), Singapore*, IEEE, 2000.

[22] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, Addison-Wesley, 2000.

[23] Kazman R., "Tool support for architecture analysis and design", In *Proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops (jointly)*, 1996.

[24] Kazman R., Barbacci M., Klein M., and Carriere J., "Experience with Performing Architecture Tradeoff Analysis Method", In *Proceedings of The International Conference on Software Engineering, New York*, 1999.

[25] Kazman R., Bass L., Abowd G., and Webb M., "SAAM: A Method for Analyzing the Properties of Software Architectures", In *Proceedings of The 16th International Conference on Software Engineering*, 1994.

[26] Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H., and Carriere J., "The Architecture Tradeoff Analysis Method", In *Proceedings of The Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), (Monterey, CA)*, 1998.

[27] Kruchten P., The 4+1 View Model of Architecture, *IEEE Software*, volume 12, issue 6, 1995.

[28] Land R. "Architectural Solutions in PAM" M.Sc. Thesis 2001

[29] Land R., "Improving Quality Attributes of a Complex System Through Architectural Analysis - A Case Study", In *Proceedings of 9th IEEE Conference on Engineering of Computer-Based Systems (to appear)*, 2002.

[30] Larsson M. "Applying Configuration Management Techniques to Component-Based Systems" Licentiate Thesis Dissertation 2000-007, Deparment of Information Technology Uppsala University. 2000

[31] Luckham D.C., Kenney J. J., Augustin L. M., Vera J., Bryan D., and Mann W., Specification and Analysis of System Architecture Using Rapide, *IEEE Transactions on Software Engineering*, issue Special Issue on Software Architecture, 1995.

[32] Medvidovic N. and Taylor R. N., "A Framework for Classifying and Comparing Architecture Description Languages", In *Proceedings of Sixth European Software Engineering Conference*, ACM, 1997.

[33] Parnas D. L., "Software Aging", In *Proceedings of The 16th International Conference on Software Engineering*, IEEE Press, 1994.

[34] Shaw M. and Clements P., "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", In *Proceedings of The 21st Computer Software and Applications Conference*, 1994.

[35] Shaw M. and Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

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

[37] van der Hoek A., "Configurable Software Architecture in Support of Configuration Management and Software Deployment", In *Proceedings of ICSE99 Doctoral Workshop*, 1999.

[38] van der Hoek A., "Capturing Product Line Architectures", In *Proceedings of 4th International Software Architecture Workshop*, ACM Press, 2000.

[39] van der Hoek, A., Heimbigner, D., and Wolf, A. L., Investigating the Applicability of Architecture Description in Configuration Management and Software Deployment, report Technical Report CU-CS-862-98, Department of Computer Science, University of Colorado, 1998.

[40] van der Hoek, A., Heimbigner, D., and Wolf, A. L., Versioned Software Architecture, 1998.

[41] van der Hoek, A., Heimbigner, D., and Wolf, A. L., Capturing Architectural Configurability: Variants, Options, and Evolution, report Technical Report CU-CS-895-99, 1999.

[42] van der Hoek A., Mikic-Rakic M., Roshandel R., and Medvidovic N., "Taming Architectural Evolution", In *Proceedings of The Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[43] Wall, A., Software Architectures - An Overview, Department of Computer Engineering, Mälardalen University, 1998.