Methods for Evaluating Software Architecture: A Survey

Banani Roy and T.C. Nicholas Graham

Technical Report No. 2008-545
School of Computing
Queen's University at Kingston
Ontario, Canada

April 14, 2008

Abstract

Software architectural evaluation becomes a familiar practice in software engineering community for developing quality software. Architectural evaluation reduces software development effort and costs, and enhances the quality of the software by verifying the addressability of quality requirements and identifying potential risks. There have been several methods and techniques to evaluate software architectures with respect to the desired quality attributes such as maintainability, usability and performance. This paper presents a discussion on different software architectural evaluation methods and techniques using a taxonomy. The taxonomy is used to distinguish architectural evaluation methods based on the artifacts on which the methods are applied and two phases (early and late) of software life cycle. The artifacts include specification of a whole software architecture and its building blocks: software architectural styles or design patterns. The role of this paper discussion is to review different existing well known architectural evaluation methods in order to view the state of the art in software architectural evaluation. This paper also concentrates on summarizing the importance of the different

evaluation methods, similarities and difference between them, their applicability, strengths and weaknesses.

Contents

List of Figures

List of Tables

1 Introduction

Maintaining an appropriate level of quality is one of the challenging issues in developing a software system. Over the past three decades, research has been going on to predict quality of a software system from a high level design description. In 1972, Parnas [96] proposed the idea of modularization and information hiding as a means of high level system decomposition for improving the flexibility and understandability of a software system. In 1974, Stevens et al. [126] introduced the notions of module coupling and cohesion to evaluate alternatives for program decomposition. Currently, software architecture has emerged as an appropriate design document for maintaining quality of software systems.

Software architecture embodies the early design decisions covering several perspectives. Generally, these perspectives are decomposition of a system's functionalities in the domain of interest, determination of the structure of the system in terms of components and their interaction, and allocation of functionality to that architecture [72]. This decomposition and the interaction between components determine non-functional properties of the application to a large degree. Moreover, they also have considerable impact on the progress of the software project by influencing both the structure of the developing organization and system properties like testability and maintainability.

Software architecture of a large system can be guided by using architectural styles and design patterns[1] [93, 15]. Architectural styles encapsulate important decisions about the architectural elements and emphasize important constraints on their elements and their relationships. An architectural style (e.g., pipes and filters, blackboard, and repository) may work as a basis for the architect to develop software architectures in a specific domain. Design patterns (e.g., observer, command and proxy) are the common idioms that have been found repeatedly in software designs. Architectural styles and design patterns are important because they encourage efficient software development, reusability and information hiding.

Since architectural decisions are among the first to be taken during system development, and since they virtually affect every later stages of the development process, the impact of architectural mistakes, and thus the resulting economical risk, is high. One of the solutions to cope with architectural mistakes is to evaluate the software architecture of a software system against the problem statements and the requirement specifications. Software architecture evaluation is a technique or method which determines the properties, strengths and weaknesses of a software architecture or a software

[1] In this paper architectural styles and design patterns are considered to be similar as both represent codified solutions to problems which repeatedly arise in software designs[15]

architectural style or a design pattern.

Software architectural evaluation provides assurance to developers that their chosen architecture will meet both functional and non-functional quality requirements. An architectural evaluation should provide more benefits than the cost of conducting the evaluation itself; e.g., Abowd et al. [1] write that *as a result of architectural evaluation, a large company has avoided a multimillion dollar purchase when the architecture of the global information system they were purchasing was, upon evaluation, not capable of providing desired system attribute to support a product line.* Software architectural evaluation ensures increased understanding and documentation of the system, detection of problems with existing architecture, and enhanced organizational learning. While there exist numerous evaluation techniques for evaluating architectures, all techniques require the participation of stakeholders, the generation of requirement lists, and a description of the software architecture with other relevant artifacts.

One of the major challenges in evaluating software architectures is that the description of software architectures and stakeholder requirements can rarely be defined precisely (e.g., it is hard to define precisely what level of maintainability is required for a software system). As a result, critical risks and problems in a software architecture might not be discovered during the evaluation procedure. Architectural evaluation does identify the incomplete architectural documentation and requirement specifications.

A number of evaluation methods have been developed which are applicable in different phases of the software development cycle. The main two opportunities for evaluation are before and after implementation [32]. As architectural styles and design patterns are important in designing successful software systems, different evaluation methods have also been developed explicitly to evaluate them. Evaluation of architectural styles and design patterns employs qualitative reasoning to motivate when and under what circumstances they should be used. This category of evaluation also requires experimental evidence to verify the usage of architectural styles or design patterns in general cases.

Considering different stages of the software development, the goal of the evaluation and different variety of quality attributes, several methods and techniques have been proposed for software architectural evaluation. Among those *scenario-based approaches* are considered quite mature [38, 7]. There are also *attribute model-based methods* [77] and *quantitative models* [128] for software architecture evaluation. However, these methods are still being validated and are considered complementary techniques to scenario-based methods. There are also empirically-based approaches [85] that define some relevant metrics for software architecture evaluation. The metrics are defined

based on the goal of the evaluation. Other approaches have been developed to systematically justify the properties of architectural styles [77] and design patterns [104, 58].

Quality goals can primarily be achieved if software architecture is evaluated with respect to its specific quality requirements before its implementation [32]. The goal of evaluating software architecture before its implementation is to discover problems of the architecture in advance for reducing the cost of fixing errors at a later stage.

In order to fix problems and adapt to new requirements, software systems continuously get changed. As a result, the implemented architecture deviates from the planned architecture. The time spent on the planned design to create an architecture to satisfy certain properties is lost, and the systems may not satisfy those properties anymore. To systematically detect and fix the deviations, software architecture is evaluated after its implementation. Different tool-based [46, 94, 112] and metrics-based [133, 85] evaluation approaches are used to evaluate the implemented architecture.

The goal of this paper is to review existing software architectural evaluation methods and to classify the methods in the form of a taxonomy. The taxonomy is used to focus on the distinction and similarity between the methods for software architecture evaluation and for architectural styles or design patterns evaluation. Evaluation of architectural styles (or design patterns) requires a lot of experimental effort and reasoning to determine the use and applicability of an architectural style for general cases. In contrast, software architecture evaluation determines the use of a software architecture for specific cases. Therefore, it is comparatively easier to evaluate a software architecture than an architectural style or design pattern.

Our proposed taxonomy also considers two phases of a software life cycle: early and late. Early software architectural evaluation techniques can be used to determine the best planned design for the project, whereas late architectural evaluation processes can be used to ensure that the planned design is carried out in the implementation.

The organization of this paper is as follows: Section 2 defines the terminology used in the context of the architectural evaluation methods. Section 3 explains what software architectural evaluation is and describes its challenges. This section also explains our proposed taxonomy of the software architectural evaluation methods. While Section 4 presents and analyzes early software architecture evaluation methods, Section 5 presents and analyzes late software architecture evaluation methods. Section 6 and 7 discuss early and late software architectural styles or design patterns evaluation methods. Section 8 summarizes and compares the four categories of software architectural evaluation. Section 9 summarizes the open problems and future work in the area of software architectural evaluation, and finally, Section 10 concludes this paper.

2 Software Architecture and Related Terminology

This section first explores the definitions of software architecture and presents different architectural views. Thereafter, software architectural styles and design patterns are described. Then, this section discusses quality in software systems and explains the relationship between software architecture and quality. Finally, this section concludes by briefly explaining the risks in software system.

2.1 Definition of Software Architecture

Typically, three arguments have been used to motivate the process of software architectural design [15]. First, it provides an artifact that allows for discussion by the stakeholders very early stage of the design process. Second, it allows for early assessment or analysis of quality attributes. Finally, the decisions captured in the software architecture can be transferred to other systems. While there are several definitions of software architecture, a commonly used definition of software architecture is as follows (Bass et al.[15]):

The software architecture of a program or computer 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.

While this definition provides the structural aspects of the software architecture, the definition given by the IEEE 2000 standard [63] emphasizes other aspects of software architecture:

Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

This definition stresses that a system's software architecture is not only the model of the system at a certain point in time, but it also includes principles that guide its design and evolution. Another well known definition of software architecture is as follows [119]:

A software architecture is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.

At the heart of software architecture is the principle of abstraction: hiding some of the details of a system through encapsulation in order to better identify and sustain its properties. A complex system will contain many levels of abstraction, each with its own architecture. An architecture represents an abstraction of system behavior at that level, such that architectural elements are delineated by the abstract interfaces they

provide to other elements at that level [15].

2.2 Architectural Views

A number of case studies and theories based on practical experience have been published, suggesting the need for multiple architectural views to capture different aspects of a software architecture [32]. The effectiveness of having multiple architectural views is that the multiple views help developers manage complexity of software systems by separating their different aspects into separate views [7]. Several architectural views share the fact that they address a static structure, a dynamic aspect, a physical layout and the development of the system. Bass et al. [15] introduced the concept of architecture structures as being synonyms to view. While different architectural view models are proposed [73, 15, 10], Krutchen [79] presents a collection of system representations that have been successfully used to depict the architecture information in several large projects. These are as follows:

- **The logical view:** This view is called functional view. It describes the functional characteristics of the software. This view is an abstraction of the system's functions and their relationships.
- **The process view:** This view describes concurrency and synchronization in the software. When a complex system is deployed, it is typically packaged into a set of processes or threads which are deployed onto some computational resources. This process view is a necessary step for reasoning about what processes or threads will be created and how they will communicate and share resources. This view is also sometimes called concurrency view.
- **The physical view:** This view describes how the software components are mapped onto the target environment.
- **The development view:** This view describes how the software is organized into modules or compilation units during the development process.

These four views are combined by using another view called *use case view* that illustrates the four views using use cases, or scenarios. The *use case view* helps developers to understand the other views and provides a means of reasoning about architectural decisions.

2.3 Architectural Styles

An architectural style is the building block of a software architecture. New architectures can be defined as instances of specific architectural styles [95]. Since architectural styles may address different aspects of software architecture, a given architecture may be composed of multiple styles. According to Garlen and Shaw [52], an architectural style defines a family of systems in terms of a pattern of structural organization which includes:

- the vocabulary of components and connectors that can be used in instances of that style,
- a set of constraints on how they can be combined. For example, one might constrain the topology of the descriptions (e.g., no cycles) and execution semantics (e.g., processes execute in parallel), and
- an informal description of the benefits and drawbacks of using that style.

There are different varieties of architectural styles available today [52, 95]. Garlen and Shaw [52] have proposed several architectural styles, such as *Pipes and Filters*, *Client-Server*, *Layered Style*, *Data Abstraction and Object-Oriented Organization*, *Implicit Invocation*, *Repositories*, and *Interpreters*. Each architectural style has defined components, connectors and architectural constraints. For example, for the *Pipes and Filters* architectural style, the components are the Filters, the connectors are Data streams (pipes) and the main architectural constraint is that Filters should not share state and should not know the identity of the upstream and downstream filters, and should use a uniform component interface to exploit reusability.

Each architectural style has advantages and disadvantages. For example, one of the important advantages of the *Pipes and Filters* style is that systems developed with this style can easily be maintained and enhanced, but the disadvantage of this style is that it is not good for handling interactive applications due its constraint. A filter cannot interact with its environment because it is inherently independent and cannot know that any particular output stream shares a controller with any particular input stream. These properties decrease user-perceived performance. In contrast, the *Data Abstraction and Object Oriented* style is good for interactive applications as it allows that for an object to interact with another object, the identity of the object (unlike Pipes and Filters) must be known. But this style is not suitable for system's scalability due to the side-effects of common object sharing. This type of tradeoff indicates the necessity of different architectural styles to satisfy the desired quality requirements and

shows that one architectural style is not good for all situations. tectural style is not good for all situations.

2.4 Design Patterns

In parallel with the software engineering research in architectural styles, the object-oriented programming community has been exploring the use of design patterns and pattern languages to describe recurring abstractions in object-based software development. A design pattern is defined as an important and recurring system construct. The design space of patterns includes implementation concerns specific to the techniques of object-oriented programming, such as class inheritance and interface composition, as well as the higher-level design issues addressed by architectural styles [85]. However, the main difference between an architectural style and a design pattern is that a design pattern does not address the structure of a complete system, but only of a few interacting components [134].

Gamma et al. [50] are famous for their book titled "Design Patterns: Elements of Reusable Object Oriented Software" where 23 object-oriented design patterns have been collected and documented. They classified these design patterns into three categories: *Creational design patterns* which concern about creation of objects (such as *singleton*), *Structural design patterns* which capture classes or object composition (such as *adapter*, *Model View Controller (MVC)* and *proxy*), and *Behavioral design patterns* which deal with the way in which classes and objects distribute responsibilities and interact (such as *observer* and *iterator*). Design patterns do not change functionalities of a system, but only the organization or structure of those functionalities. Applying a design pattern generally affects only a limited number of classes in the architecture. The quintessential example used to illustrate design patterns is the MVC design pattern. The MVC pattern (Buschmann et al. [27]) is a way of breaking an application, or even just a piece of an application's interface, into three parts: the *model*, the *view*, and the *controller*. This pattern decouples changes to how data are manipulated from how they are displayed or stored, while unifying the code in each component. The use of MVC generally leads to greater flexibility and modifiability. Since there is a clearly defined separation between the components of a program, problems in each domain can be solved independently. New views and controllers can be easily added without affecting the rest of the application

2.5 Software Quality and Quality Attributes

In the IEEE Glossary of Software System Engineering Terminology [63], quality is defined as *the degree to which a system, a component, or a process meets customer or user needs or expectations*. The quality of the software is measured primarily against the degree to which user requirements, such as correctness, reliability and usability are met. The use of a well defined model of the software development process and good analysis, design and implementation techniques are prerequisite to ensuring quality.

Measuring quality is challenging since measurement must consider the perspectives of different stakeholders. Garvin [54] described quality from five different perspectives: the *transcendental view*, which sees quality as something that can be recognized, but not defined; the *user view*, which sees quality as fitness for the user's purpose; the *manufacturer's view*, which sees quality as conformance to specification; the *product view*, which sees quality as tied to inherent characteristics of the product; and the *value-based view*, which sees quality as dependent on what a customer is willing to pay for it. Pressman [105] mentions that usually software quality is "conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software".

The factors that affect quality are termed as quality attributes. There are different categorizations of quality attributes. Quality attributes can be categorized into two broad groups: attributes that can be directly measured (e.g. performance) and attributes that can be indirectly measured (e.g., usability or maintainability). In the latter category, attributes are divided into sub-attributes so that they can be measured directly. To better explain the quality view, several quality models such as Bohem's Quality Model (1978) [24], McCall's Quality Model (1997) [89], and ISO 9126 Quality Model [65] have been proposed.

Some commonly used quality attributes in the architectural evaluation process are as follows:

- **Maintainability:** Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, requirements and/or functional specifications.

- **Usability:** Usability is a metric to measure how comfortably and effectively an end user can perform the tasks at hand. It involves both architectural and non-architectural aspects. The non-architectural aspects include making user interface

clear and easy to use and the architectural aspect include for example, support for cancelation and undo commands.

- **Performance:** Performance measures execution time and resource utilization. Events occur and a system must respond to them. According to Smith and Williams [122], *performance refers to responsiveness: either the time required to respond to specific events or the number of events processed in a given interval of time*. There are different architectural aspects of performance, such as how much communication is necessary among components, what functionality has been allocated to each component and how shared resources are allocated to each component.

- **Availability:** Availability is concerned with system failure and its associated consequences. It is defined as the relative amount of time the system functionality is accessible. System breakdowns and malfunctions have a negative effect on availability.

- **Reliability:** The mean amount of time that the software is available for use as indicated by several sub-attributes, such as maturity, fault tolerance, and recoverability.

- **Reusability:** The extent to which a program or parts of the program can be reused in other applications.

- **Security:** This attribute ensures the integrity and confidentiality of exchanged information or information of a server by prohibiting intruders accessing unauthorized information.

2.6 Relationship between Software Quality and Software Architecture

Software architecture is a key determinant of whether system quality requirements can be met. In software intensive systems, software architecture provides a powerful means to achieve the system qualities over the software life cycle [15]. Figure 1 shows how software architecture can help predict the quality of a software system. First, the relevant quality attributes are identified from the system's requirement specification. In the next step, these quality attributes are used to drive the software architecture which is subsequently used as a blueprint to assess the system's capabilities and qualities.

Figure 1: Relationship between software architecture and software quality (from Bergey et al. [21])

The software architecture acts as a bridge to attain software quality specified in the system's requirements.

2.7 Risks of Software Systems

Risk is measured as the probability and severity of adverse effects that may arise in the development, maintenance and operations of software [31]. Architectural risk management is the process of identifying and addressing software risks. Architectural risk analysis includes identification and evaluation of risks and risk impacts, and recommendation of risk-reducing measures. Architectural risks can be identified by investigating whether a software architecture satisfies different quality attributes.

3 Software Architectural Evaluation

Software architectural evaluation is a technique which determines properties of software architectures, or software architectural styles, or design patterns by analyzing them. The architectural evaluation verifies architectural decisions against problem statements and requirement specifications. It determines the degree to which a software architecture or an architectural style satisfies its quality requirements.

Software qualities are not completely independent and may interact each other positively or negatively. E.g., modifiability has negative influence on performance and positive affect on reliability and security. Architectural evaluation helps identify and analyze these tradeoffs between quality attributes.

The architectural evaluation process takes software architecture documentation, problem statements, and requirement specifications as inputs. Then, it generates corrected, improved and organized software architecture documentation as output of the

evaluation process. Software architecture documentation contains, for example, different architectural views, specification of architectural elements, connectors and other important information about how architectural elements relate, and specifications of constraints that impact architectural decisions. The team who conducts an architectural evaluation comprises various kinds of stakeholders, such as project managers, system administrators, architects, developers and users. The role of each stakeholder is determined according to his/her domain knowledge.

3.1 Challenges in Software Architectural Evaluation

Software architecture evaluation is a flexible way to uncover problems in a software architecture before the architecture has been committed to code. There are some challenges in architectural evaluation, including, for example:

- Software architectural evaluation requires an expert evaluation team to deal with unpredictable risks along with the known risks. For example, analyzing architectural decisions for satisfying security requirement is challenging. No one can accurately predict how often an attack will occur and how effectively security mechanisms will mitigate the damage. However, an experienced security manager in the evaluation team can estimate the risk and the effectiveness of proposed risk mitigation strategies [28].

- There can be a lack of common understanding of the high level design [104]. Software architects often do not document design rationale. When they do, they may not follow a systematic way of expressing the rationale [130]. There is also no standard notion to describe software architectures. However, currently Unified Modeling Language (UML) is widely used as an architectural specification language, but it is still not possible to express various architectural notations (e.g., quality attributes of interest) using UML.

- The quality requirements for a system may not be written properly or may not be finished when the architecture is designed. For example, often a requirement statement is written like " the system shall be modifiable" which does not indicate any particular change for which the system should be modifiable. As a result, quality attributes sometimes become vague for architectural analysis.

3.2 Taxonomy of Software Architectural Evaluation

Software architectural evaluation can be conducted at different phases of software life cycle. In this paper, we distinguish between early and late software architectural evaluations. A software architecture can be evaluated before its implementation (early evaluation), or after its implementation (late evaluation). These evaluation techniques can be categorized according to the type of design artifacts on which the evaluation methods are applied. Some evaluation methods are intended to evaluate the whole architecture, whereas some are intended to evaluate the building blocks of software architecture, such as architectural styles or design patterns. Our taxonomy for software architectural evaluation is shown in Table 1.

Early software architectural evaluation can be conducted on the basis of the specification and description of the software architecture, and other sources of information, such as interviews with architects. Late software architectural evaluation is performed based on metrics, for example, cohesion and coupling of architectural components. For both early and late software architectural evaluations, Abowd et al. [1] have suggested two basic categories: qualitative evaluation and quantitative evaluation.

Qualitative evaluation generates qualitative questions about software architecture to assess any given quality, whereas quantitative evaluation uses quantitative measurements to be taken from a software architecture to address specific software qualities. Techniques for generating qualitative questions include scenarios, questionnaires, and checklists. Scenarios appear to be the most utilized form for acquiring data [6]. Different scenario-based software architecture evaluation methods have been developed so far [72, 73, 83, 92, 20, 127, 132, 142].

One of the important approaches studied in software architecture is the use of architectural styles and design patterns for handling the complexity at architectural level. Software architects are encouraged to use pre-existing architectural styles and design patterns to make the software systems more understandable, maintainable and reusable. It is therefore important to evaluate the architectural styles and design patterns in order to determine their strengths and weaknesses. Attribute Based Architectural Style (ABAS) [77], Golden et al.'s method [58] and Prechelt et al.'s method [104] are some of the existing approaches for evaluating architectural styles and design patterns.

Table 1: **Taxonomy of Software Architectural Evaluation**

Evaluation	Applied to software architecture			Applied to SAS - DP*	
	Category	**Name**		**Category**	**Name**
Early	Scenario-based	SAAM [72]	Scenario-based Software Architecture Analysis Method	Scenario-based	ABAS [77] - Attribute-based Architectural Style
		ATAM [73]	Architecture-based Tradeoff Analysis Method		
		ALPSM [18]	Architecture Level Prediction of Software Maintenance		
		ALMA [20]	Architecture-Level Modifiability Analysis		
		SBAR [17]	Scenario Based Architecture Reengineering		
		SALUTA [49]	Scenario-based Architecture Level UsabiliTy Analysis		
		SAAMCS [83]	SAAM for Complex Scenarios		CBAM [71]- Cost Benefit Analysis Method
		ESAAMI [92]	Extending SAAM by Integration in the Domain		
		ASAAM [132]	Aspectual Software Architecture Analysis Method		
		SACAM [22]	Software Architecture Comparison Analysis Method		
		DoSAM [127]	Domain-Specific Software Architecture Comparison Model		
	Mathematical model-based: reliability	Path-based	Shooman [120]	Controlled Experiment	Golden et al. [58]
			Krishnamurty and Mathur [78]		
			Yacub et al. [143]		
		State-based	Cheung [30]		
			Kubat [80]		Junuzovic and Dewan [68]
			Laprie [82]		
			Gokhale and Trivedi [56]		
	Mathematical model-based: performance	SPE [123]	Software Performance Analysis	Mathematical model-based	Petriu and Wang [99]
		WS [138]	Williams and Smith		
		PASA [137]	Performance Assessment of Software Architecture		
		CM [35]	Cortellessa and Mirandola		
		BIM [9]	Balsamo et al.		Gomaa and Menascé [59]
		ABI [4]	Aquilani et al.		
		AABI [2]	Andolfi et al.		
Late	Metrics-based	TLC [133]	Tvedt et al.	Controlled Experiment	Prechelt et al. [104]
		LTC [85]	Lindvall et al.		
	Tool-based	FA [46]	Fiutem and Antoniol		
		MNS [94]	Murphy et al.		
		SSC [112]	Sefika et al.		

***Applied to software architectural styles or design patterns**

4 Early Evaluation Methods Applied to Software Architecture

Early software architecture evaluation methods are applied to a software architecture before its implementation. Quality goals can primarily be achieved if the software

architecture is evaluated with respect to its specific quality requirements at the early stage of software development. When a software system is in its early development stage, it is easy to change inappropriate architectural decisions, but at later stages, reverting the changes, or noticing that the architecture is not appropriate for satisfying non-functional quality attributes can be costly.

Four approaches have been developed for early software architecture evaluation: scenario-based, mathematical model-based, simulation based and experience-based reasoning. The scenario-based approaches are flexible and simple [6, 7]. Many scenario-based architecture evaluation methods have been presented in the literature. Mathematical model-based evaluation techniques for assessing the operational quality attributes, such as reliability and performance are also well used, particularly in real-time software systems. On the other hand, few techniques have been developed for simulation and experience-based evaluation. In this paper, we cover the scenario and mathematical model-based evaluation techniques.

4.1 Scenario-based Software Architecture Evaluation Methods

Scenario-based evaluation methods evaluate a software architecture's ability with respect to a set of scenarios of interest. Scenario is brief descriptions of a single interaction of a stakeholder with a system [15]. A scenario expresses a particular quality attribute to compensate for the lack of fundamental understanding about how to express that quality attribute.

The scenario-based evaluation methods offer a systematic means to investigate a software architecture using scenarios. These methods determine whether a software architecture can execute a scenario or not. Evaluation team explores/maps the scenario onto the software architecture to find out the desired architectural components and their interactions, which can accomplish the tasks expressed through the scenario. If the software architecture fails to execute the scenario, these methods list the changes to the software architecture required to support the scenario and estimate the cost of performing the changes. Scenario-based evaluation methods require presence of relevant stakeholders to elicit scenarios according to their requirements. Stakeholders use scenarios to exchange their views and opinions, and come to a common agreement about the architectural decisions.

For example, in a library automation system, a scenario might be: *"Separate user interaction tasks from data storage management "*. When the evaluation team executes

the scenario onto the software architecture, the evaluation team looks for architectural components that offer the functionalities related to user interactions and data storage and checks whether they are properly separated or not. If they are not, the evaluation team marks the problem as poor separation of concerns and finds out the solution to fix the problem, and estimates the effort required for the fixation.

Scenario-based methods can ensure discovery of problems in software architectures from different point of views by incorporating multiple stakeholders during the scenario elicitation process; e.g., system administrator can point out the maintenance problems, whereas the end user can indicate the performance issues. In addition to technical benefits, scenario-based evaluation methods produce social benefits since most involve a brainstorming session to a wide group of stakeholders. The architecture works as a communication vehicle for the stakeholders – a shared language that allows them to discuss their concerns in a mutually comprehensible language.

However, there are challenges in scenario-based evaluation methods, e.g., what happens if important scenarios are missed so that the evaluation fails to discover critical risks and problems in software architectures. There is no particular number of scenario execution which can provide assurance that all the possible risks are identified and the evaluation results are effective. Employing experienced stakeholders in the evaluation session can mitigate this problem to some extent. Another challenge is that a scenario-based method cannot give an absolute measurement about the software quality, e.g., a scenario-based method cannot say how many person-days will be required to enact a change in a software system.

While there exist many scenario-based evaluation methods, SAAM is considered as the parent method for the rest of scenario-based evaluation methods. Many scenario-based methods are refinement of SAAM, so in the following, we will first introduce SAAM in detail and then will cite other methods that are developed based on SAAM. We will also show how these methods differ from SAAM. Finally, we will compare these scenario-based methods in two forms. First, we will compare them using different comparison criteria. Then, we will investigate them using various properties of scenario-based evaluation methods.

4.1.1 SAAM

The goal of SAAM (Scenario-based Software Architecture Analysis Method) is to verify basic architectural assumptions and principles against documents that describe the desired properties of an application. This analysis helps assess the risks inherent in an architecture. SAAM guides the inspection of the architecture, focusing on potential

Figure 2: Common activities in scenario-based evaluation methods

trouble spots such as requirement conflicts or incomplete design specification from a particular stakeholder's perspective. Additionally, SAAM helps compare candidate software architectures.

Kazman et al. [72] first proposed SAAM in 1993 to compare competing software architectures. Over the years, the prescribed steps of SAAM have evolved with the increased experience in architectural analysis. In 2002, Clements et al. [32] have presented another version of SAAM in their book titled "Evaluating Software Architectures Methods and Case Studies". SAAM is easy to learn and easy to carry out with relatively small amounts of training and preparation. According to Clements et al. [32], SAAM is a good place to start if one has not done an architecture evaluation before. The main inputs to the SAAM are business drivers, software architecture description, and quality requirements. The outputs of this method include quality sensitive scenarios, mapping between those scenarios and architectural components, and the estimated effort required to realize the scenarios on the software architecture. There are six activities in the SAAM which are shown in Figure 2 and discussed as follows:

- **Specify requirements and design constraints:** In this step, SAAM collects the functional and non-functional requirements, and design constraints (e.g., man power, budget, time and language constraints). In a group meeting, the evaluation team collects these requirements from the stakeholders and specifies the requirements. ATAT software system [32] was created as an experimental tool for aiding architectural evaluation, and for managing and analyzing architecture. A brief sample of the requirement specification for this software system is as follows.

  The ATAT tool must include:

  – Presentation support for both architectural (components and connectors)

and non-architectural (scenarios and stakeholders) elements,

  - Analysis supports for establishing link between architectural decisions and quality attributes, and guiding the users to find the right information,
  - Process support for enacting processes and providing process guidance to the user, and usability supports for interacting with people and exchanging information.

- **Describe software architecture:** In this step, the candidate architectures are described using a syntactic architectural notation that is well-understood by the evaluation team. SAAM uses different architectural views, such as *a functional view*, *a physical view*, *a code view* and *a concurrency view*. Architects also use a component coordination view (like UML communication diagram) and dynamic views (like UML sequence diagram) to understand the dynamic behavior of the system. The architects use these architectural views as per the goal of the evaluation, e.g., a *functional view* can be used for reasoning about work assignments and information hiding, whereas a *process view* can be used for reasoning about performance [15]. As an example, the *functional view* of ATAT is shown in Figure 3. The four components in the functional architectural view are briefly described as follows:

  1. **Process Support Components:** responsible for enacting process, providing guidance to the user, and ensuring that the user is aware of the current process state.
  2. **Architectural Editorial Components:** responsible for viewing different kinds of architectural and non-architectural information.
  3. **Central Data Repository:** responsible for storing information.
  4. **Constraint Checker:** checks for constraint violations.

  For implementing the architectural editor components (each instance of the component is shown in Figure 3), the architects can propose different candidate architectures, such as the Model-View-Controller (MVC) pattern and the Presentation Abstraction Control (PAC) pattern [32]. The architects describe all the candidate architectures to the evaluation team and the other stakeholders. This activity helps the evaluation team to justify which candidate architecture best fits the implementation.

Figure 3: Functional view of an example software architecture (from Clements et al. [32])

- **Elicit scenarios:** Scenarios are elicited with the presence of the relevant stakeholders. The scenarios represent the tasks that are realized by the stakeholders - end user, customer, maintainer, developer and architect. Stakeholders exchange their opinions through a brainstorming session and create a representative set of scenarios that addresses relevant quality attributes. The effectiveness of the SAAM analysis more or less depends on quality of the elicited scenarios. Scenarios should capture many things, such as all the major uses of the system, users of a system, anticipated changes in the system and the qualities that the system must satisfy now and in the foreseeable future. The processes of scenario development and architectural description are related and iterative (see Figure 2). As more architectural information is collected and shared, more meaningful scenarios are surfaced by the stakeholders.

- **Prioritize Scenarios:** The stakeholders prioritize scenarios according to their importance. This allows the most important scenarios to be addressed within the limited amount of time available for evaluation. The importance of the scenarios depends on the opinions and concerns of the stakeholders. A sample outcome of this scenario prioritization activity for the *ATAT* software architecture evaluation is shown in Table 2.

- **Evaluate architectures with respect to scenarios:** In this step, the impact of the scenarios on software architectures is explored. The evaluation team estimates the changes and the effort needed to realize the scenarios. SAAM classifies two types of scenarios: *indirect scenarios*, which cause modifications to the architecture and *direct scenarios*, which do not. In the case of a *direct scenario*, SAAM

Figure 4: An example flow description of a direct scenario [116]

checks how the scenario can be executed by the architecture. Figure 4 shows a sample flow diagram to map a direct scenario onto the architecture. Each column in the diagram represents an architectural component. An arrow between columns represents either a flow of information (labeled D in the diagram for data connection) or a control relationship (labeled C).

In the case of an *indirect scenario*, SAAM predicts the architectural changes that the scenario will require. A change to the architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification. SAAM also estimates the cost of performing such changes. A sample scenario evaluation results and effort estimations for the two scenarios of Table 2 for the MVC-based candidate architecture is shown in Table 3. Similar evaluation results for the PAC-based architecture are shown in Table 4.

Table 3: **SAAM Scenario Evaluation for a MVC- based Architecture**

Scenario no.	Scenario type	Elements requiring change	No. of changed/added components	Effort for changes (estimate)
7	Indirect	Model, interface to data repository	3	1 person- month
8	Indirect	Model, controller	3	1 person week

- **Interpret and present results:** At the end of the architecture evaluation with respect to the scenarios, SAAM interprets the evaluation results according to the

Table 4: **SAAM Scenario Evaluation for a PAC-based Architecture**

Scenario no.	Scenario type	Elements requiring change	No. of changed/added components	Effort for changes (estimate)
7	Indirect	Abstraction, interface to data repository	1	1 person- month
8	Indirect	The ATAT-Entity PAC agent for scenarios and stakeholders	2	1 person week

evaluation goals. For the indirect scenarios, SAAM determines the coupling of the architectural components by means of the scenario interactions. If two or more unrelated scenarios cause changes to the same components, SAAM determines that the components are tightly coupled, i.e. exhibit poor separation of concerns. SAAM also determines the cohesion of the architectural components. If an indirect scenario affects multiple components, SAAM decides that the architectural components are loosely cohesive i.e. the concerns are distributed. If the goal of the evaluation is to compare multiple candidate architectures, SAAM compares the scenario interaction results, and summarizes the comparison decisions in a table for each candidate architecture, e.g. Table 5 shows that for the scenario 8 (shown in Table 2), the PAC-based architecture is better than the MVC-based architecture.

Table 5: **Evaluation Summary Table**

Scenario no.	MVC	PAC
7	0	0
8	-	+

+: better, -: worse, and 0: no significant difference

Finally, the evaluation team creates an organized written document putting together the elicited scenarios, specification of the software architectures, scenario mapping results, and other important findings and their opinions about the candidate software architectures. Finally, the evaluation team may present the evaluation results.

SAAM has been applied to numerous case studies: global information systems, air traffic control, WRCS (a revision control system), user interface development environments, Internet information systems, keyword in context (KWIC) systems, and embedded audio systems [116, 32]. SAAM has identified different design problems in these systems and provided proper guidelines to fix them. For example, while evaluating

architecture of WRCS, which allows project developers the ability to create archives, compare files, check files in and out, create releases, and back up to old versions of files system, SAAM identified several limitations in achieving the desired portability and modifiability. Thus, a major system redesign was recommended. The senior developers/manager found it important and useful, whereas the other developers regarded this as just an academic exercise [116]. Another good example is the evaluation of a Global Information System's (GIS) architecture. SAAM revealed that the GIS's architecture was inappropriate for the context in which the company wanted to use it. Therefore, the company stopped buying the GIS from the supplier, which saved an investment of tens millions of dollars [116]. SAAM evaluation helped greatly to increase the customer's understandings of the supplier's architecture and direct scenarios played a crucial role in helping the customer to determine the suitable level of detail for the architectural representation. However, although SAAM is a well validated and widely used method, it has several limitations. In the following, we discuss the limitations of the SAAM while presenting other methods that have evolved to address these limitations.

4.1.2 ATAM

SAAM does not consider interactions amongst competing quality attributes. SAAM evaluates software architecture without considering that improvement of one quality attribute often comes at the price of worsening one or more of the others i.e. an architectural decision can affect two quality attributes in an opposite manner. For example, the performance and availability of the client-server architectural style are sensitive to the number of servers considered in the architecture. Both performance and availability can be increased if the number of servers is increased. However, this architectural decision can negatively impact the security of the system, because increasing number of servers, increases potential points of attack and failures. Therefore, performance and availability trade off when determining number of servers. ATAM (Architecture-based Tradeoff Analysis Method) has evolved as a structured way for understanding the tradeoffs in software architecture. ATAM provides a principled way to evaluate the fitness of a software architecture with respect to multiple competing quality attributes.

ATAM activities are more structured and complex than those of SAAM. ATAM consists of nine activities, where some of the activities are executed in parallel. The new three activities are: *present ATAM*, *identify the architectural approaches* and *present ATAM evaluation results*. Although the other six activities of ATAM that appear to be similar as SAAM at a coarse grained level, the activities are different at a fine grained level.

Figure 5: An example utility tree in an e-commerce system (adapted from Clements et al. [32])

ATAM uses a utility tree to capture the high priority scenarios from the relevant quality attributes. The utility tree provides a mechanism for directly and efficiently translating the business drivers of a system into concrete scenarios with priorities. The prioritization may be between a 0 to 10 scale or may use relative ranking, such as *High (H)*, *Medium (M)* and *Low (L)*. Participants prioritize the utility tree along two dimensions: (1) by the importance of each scenario to the success of the system and (2) by the degree of difficulty posed by the achievement of the scenario, in the estimation of the architect.

Figure 5 shows an example utility tree for an e-commerce system. In this system, two of the business drivers can be stated as: *Security is central issue to the success of the system since ensuring the privacy of our customers' data is utmost importance* and *Modifiability is the central to the success of the system since we need to be able to respond quickly to an evolving and a competitive market place*. In Figure 5, these two business drivers are refined into two specific and concrete quality attributes: security and modifiability. The attributes are then decomposed into general scenarios. After that concrete scenarios are derived from the generalized scenarios. Finally, the concrete scenarios are prioritized.

Architects describe the software architecture using Kruchten's 4+1 views [79]. The architects may also employ other architectural views, such as *a dynamic view*, showing how systems communicate, *a system view*, showing how software is allocated to hardware, and *a source view*, showing how components and systems are composed of objects. Additionally, the candidate architectures are described in terms of the architectural elements that are relevant to each of the important quality attributes. For example, according to Kazman et al. [73], voting schemes are an important element for reliability; concurrency decomposition and process prioritization are important for performance; firewalls and intruder models are important for security; and encapsulation

is important for modifiability.

ATAM identifies existing architectural approaches and uses ABASs (Attribute Based Architectural Style) to analyze them with resect to quality attribute specific questions. The ABASs offer attribute specific-frameworks to illustrate how each architectural decision embodied by an architectural style affects the achievement of a quality attribute. For example, a modifiability ABAS helps in assessing whether a publisher/subscriber architectural style would be well-suited for a set of anticipated modifications. Using the ABASs, ATAM can support other techniques like mathematical models to evaluate the architectural approaches, because the ABAS for a particular quality attribute offers an analytic model, which can be either quantitative or qualitative, to assess a particular quality attribute.

During the analysis, the ATAM determines sensitivity points, the architectural decisions that are sensitive to achieve quality attributes. It then determines tradeoff points by identifying sensitivity points that affect the same quality attributes. In the client-server architectural style, the architectural decision of the number of servers is treated as a tradeoff point as the decision is sensitive for both the performance and security. The architectural risks are identified when no satisfactory value of the response for an action is obtained from the architectural approaches, e.g. some assignments of process to the server result in unacceptable values of the response which is a risk [73].

At the end of the analysis, the evaluation team lists the architectural approaches relevant to the utility tree, the analysis question(s) associated with each approach, the architects' responses to the questions, and the identified risks, non-risks, sensitivity and tradeoff points.

ATAM is a well studied method. Several ATAM evaluations have been conducted over the past several years. The members of the Software Engineering Institute (SEI) [114], creators of ATAM, have taken various steps to validate ATAM. Recently, Bass et al. [14] have published a technical report which presents 18 ATAM evaluations.

4.1.3 ALPSM and ALMA

SAAM and ATAM offer generalized frameworks for addressing multiple quality attributes. In contrast, ALPSM (Architecture-Level Prediction of Software Maintenance) [18] and ALMA (Architecture-Level Modifiability Analysis) [20] are specifically developed to address the maintainability quality attribute. The goal of ALPSM is to predict the maintenance effort required to address a change scenario. ALMA extends ALPSM to address risk assessment and comparison of candidate architectures.

Both of these methods require access to the maintainers. The maintainers collect

change scenario to be used in the analysis. ALMA and ALPSM measure the cost of enacting a change scenario in terms of its impact on the size of the system's components. ALMA suggests three alternative techniques for estimating the size of components. Total maintenance effort is predicted by summing up the size of components multiplied by the weights of scenarios. Scenarios are assigned weights depending on their priorities. ALMA also suggests for consideration of ripple effects [85] in order to fine tune the estimation result.

ALMA has been applied to several systems software architectures, such as telecommunication systems, information systems, embedded systems, and in the medical domains. It is considered to be a mature method.

4.1.4 SBAR

SAAM uses scenarios to assess the development-time quality attributes. It provides no techniques for assessing operational quality attributes. Bengtsson and Bosch [17] proposed the SBAR (Scenario-Based Architecture Reengineering) method to assess operational quality attributes. SBAR provides four evaluation techniques: scenario-based, mathematical model-based, simulation-based and experience-based reasoning. ATAM also supports multiple evaluation technique. But SBAR differs from ATAM since SBAR offers different techniques for transforming an architecture. However, when SBAR uses scenario-based evaluation technique, it is basically similar to SAAM, but add architecture transformation steps.

The goal of SBAR is to define Domain Specific Software Architecture (DSSA). The DSSA provides a reusable and flexible basis for instantiating measurements. In this method, an initial version of a software architecture is designed. Then it is evaluated with respect to the quality requirements by using the technique suitable for a particular quality attribute. The estimated values of the quality attributes are compared to the values in the specification. If these are satisfactory then the architecture reengineering process is finished, otherwise the architecture transformation is performed. Five architecture transformation techniques: *impose architectural style*, *impose architectural pattern*, *apply design patterns*, *convert quality requirements to functionality*, and *distribute requirements* are proposed to re-engineer the software architecture. The transformation process continues until all non-functional requirements are satisfied as much as possible.

When SBAR uses scenario-based technique to assess development-time quality attributes, it defines scenario for each quality attribute and maps them on software architecture. SBAR proposes two alternative manners for the assessment: complete and

statistical. In the first approach, a set of scenarios is defined that altogether cover the concrete instances of the software quality. For example, for reusability, all the scenarios that are relevant to reuse the architecture or parts of it are captured. If the architecture can satisfy all the scenarios without problems, the reusability of the architecture is optimal. The second approach is to define a set of scenarios that is a representative sample without covering all possible cases. The ratio between scenarios that the architecture can handle and scenarios not handled well by the architecture provides an indication of how well the architecture fulfils the software quality requirements. Both approaches, obviously, have their disadvantages. A disadvantage of the first approach is that it is generally impossible to define a complete set of scenarios. The definition of a representative set of scenarios is the weak point in the second approach since it is unclear how does one decide that a scenario set is representative.

SBAR has evolved through its application in three projects: fire-alarm systems [25], measurement systems [72] and dialysis systems [17].

4.1.5 SALUTA

SAAM and ATAM provide generalized frameworks for assessing quality attributes. Folmer et al. proposed SALUTA (Scenario-based Architecture Level UsabiliTy Analysis) [49] as a specialized framework directed towards the assessment of usability quality attributes. SALUTA is the first method to assess usability before the implementation of a software architecture [47]. However, there are several earlier works [16, 13, 69, 49] attempted to link usability with software architecture.

SALUTA does not use any specific architectural view to describe a software architecture. It extracts two types of information from the software architecture: (1) usability patterns, the design patterns used to solve a particular scenario and (2) usability properties, the architectural decisions that affect the usability attribute. Usability patterns and properties are identified by analyzing the software architecture, using functional design documentation, and interviewing software architect(s).

SALUTA divides usability into four sub-attributes: satisfaction, learnability, efficiency and reliability. SALUTA elicits usage scenarios from usability requirement specifications. These scenarios represent different use of a system for different types of users and context of use. Using these usage scenarios, SALUTA creates usage profile that represents the required usability of the system. To create a usage profile, users, their tasks and context of use are identified for each usage scenario. The usability sub-attributes are quantified to express the required usability of the system for each usage scenario. For example, consider a usage scenario described by Folmer et al. [49]:

Figure 6: Subset of relationship between usability patterns, properties and attributes (adapted from Folmer et al. [48])

"every page should feature a quick search which searches the whole portal and comes up with accurate search results". The underlying requirement of this scenario is that the searching task should be done quickly and accurately. Thus, for this scenario, SALUTA assigns higher values to the efficiency and reliability sub-attributes than those of learnability and satisfaction. After that, SALUTA extracts usability patterns and usability properties from the candidate architectures. It determines the hierarchical relationship among usability attributes, usability properties and usability patterns shown in Figure 6. In this way, SALUTA relates the problem domain with the solution domain.

These hierarchical relationships work as a framework to find a solution for satisfying a particular usability sub-attribute. For example, the *Undo* and *Cancel* usability patterns can be used to implement the error management usability property, which in turn satisfies the efficiency and reliability sub-attributes. Each usage scenario is analyzed in terms of the usability patterns and properties that it affects. For example, if *undo* affects a given scenario then the relationships of the undo pattern with usability are analyzed to determine how well that particular scenario is supported. For each scenario, the results of the support analysis are expressed using quantitative measures. For example, support may be expressed on a five level scale (++, +, +/-,-,–).

SALUTA is a new method. Folmer et et al. have applied SALUTA to three case studies: *eSuit*, a web based enterprise resource planning (ERP) system; *Compressor*, a web based e-commerce system; and *Web platform*, a web based content management system (CMS).

4.1.6 SAAMCS

SAAM and ALMA do not provide any special framework to handle complex scenarios that are hard to implement. Thus, these methods may not expose the limits or

boundary conditions of a software architecture by handling possibly implicit assumptions. ATAM defines exploratory scenarios to address complex scenarios. But ATAM does not provide specific factors that can make scenarios complex to implement. Lassing et al. [83] proposed the SAAMCS (SAAM for Complex Scenarios) to expose the boundary conditions of an administrative system's architecture with respect to flexibility. SAAMCS defines a class of scenarios that are possibly complex to realize. These scenarios are defined according to three factors that influence the complexity of scenarios. These factors are: *level of impact of the scenario on the software architecture*; *need for coordination between different owners*, and *presence of version conflicts*. SAAMCS analyses the software architecture by measuring at what level the software architecture affects these factors to address the complex scenarios.

The first factor, *level of impact of the scenario on the software architecture*, is used to estimate the effects of a scenario at the micro and macro-architecture levels. Micro-architecture refers to present the internal structure of the system, whereas the macro-architecture defines the role of the system in its environment. Macro-architecture is considered for the system which is integrated with other systems. A system's environment becomes increasingly complex due to the interaction with other systems. For both types of architectures, four impact levels are possible: 1) the scenario has no impact; 2) the scenario affects one component; 3) the scenario affects multiple components; and 4) the scenario affects the whole software architecture, i.e. both micro and macro-architecture.

The second factor, *need for coordination between different owners*, is based on the premise that scenarios affecting components that belong to different owners are inherently more complex to realize than those whose impact is limited to the components of a single owner. Having multiple owners for a component requires additional coordination between the various parties.

The third factor, *presence of version conflicts*, is important, because a scenario that leads to different versions of architectural components is hard to realize. Such scenarios may cause changes to an architectural element that might be unaffected in earlier versions. Four levels of difficulties related to versions are defined: 1) No problem with different versions; 2) The presence of different versions is undesirable, but not prohibitive; 3) The presence of different versions creates complications related to configuration management; and 4) The presence of different versions creates conflicts.

After analyzing a software architecture, SAAMCS uses a measurement instrument to score the response of the software architecture for each factor. A sample of the measurement instrument is shown in Figure 7.

Figure 7: Evaluation framework of the SAAMCS (from Lassing et al. [83])

4.1.7 ESAAMI

Software architecture evaluation is a human and knowledge-intensive activity that can be expensive practice if each evaluation starts from scratch. SAAM does not put any emphasis on knowledge management for reusability, e.g., SAAM does not provide templates for scenarios allowing their future reuse. Molter et al. [92] proposed ESAAMI (Extending SAAM by Integration in the Domain) to integrate SAAM in a domain-centric and reuse-based development process. ESAAMI offers packages of analysis templates which represent the essential features of the domain. The analysis templates collect reusable products, e.g., *proto-scenarios* that can be deployed in the various steps of the method. Proto-scenarios are generic descriptions of reuse situations or interactions with the system.

4.1.8 ASAAM

SAAM and its successor methods focus architectural concerns that can be localized current architectural abstractions. However, there are some concerns at the architectural design level that inherently cut across multiple architectural components and therefore, cannot be localized. SAAM and other methods do not address these "crosscut" concerns (which are also called aspects). As a result, the potentially important aspects might not be considered during architectural design and hence, might not be solved in the implementation. This may lead to poor separation of concerns in the system, and may increase maintenance cost and effort. Tekinerdogan [132] proposed ASAAM (Aspectual Software Architecture Analysis Method) to identify, specify and evaluate aspects at the architectural design level. ASAAM is based on SAAM, but it enhances and refines SAAM by incorporating architectural aspects during the analysis.

ASAAM works in two phases. In the first phase, it identifies aspects. In the second phase, it classifies different types of components based on how they handle aspects. In these two phases ASAAM is based on five artifacts: architecture, problem description, scenario, aspect, and component. The two phases are described as follows:

Figure 8: The six heuristic rules and their applications to obtain the aspects (from Tekinerdogan [132])

Identify aspects: ASAAM classifies direct and indirect scenarios (based on SAAM's classification of scenarios) into aspectual scenarios. Aspectual scenarios are those for which the required changes to software architecture are scattered over many architectural components, and therefore cannot be captured in a single component. Aspectual scenarios are obtained by applying six heuristic rules on the direct and indirect scenarios. An example, rule is *"IF DIRECT SCENARIO IS SCATTERED AND CANNOT BE LOCALIZED IN ONE COMPONENT THEN DIRECT SCENARIO IS ASPECTUAL SCENARIO"*. Architectural aspects are derived from these aspectual scenarios by thoroughly analyzing the software architecture. Figure 8 shows how ASAAM obtains aspects by applying the six rules.

Identify components: Like SAAM, ASAAM performs scenario interactions and identifies the nature of the components. ASAAM defines 12 more rules to identify different types of components, such as direct, indirect, cohesive, composite, tangled, and ill-defined. The component identification process is shown in Figure 9. The rules are defined based on different cases of scenario interactions. For example, for the rule "*R15: IF INDIRECT COMPONENT includes semantically distinct scenarios AND cannot be decomposed THEN COMPONENT becomes TENTATIVE TANGLED COMPONENT*", the tentative tangled component is the intermediate artifact that helps to identify tangled and ill-defined components.

After executing these heuristic rules for each component, related aspects are identified and the components are characterized more specifically. ASAAM is a newly proposed method. It has been only applied in a window management system.

Figure 9: ASAAM's tangled component identification process (from Tekinerdogan [132])

4.1.9 SACAM and DoSAM

None of the above mentioned methods allow standard frameworks for comparing several architectures. Most are focused on evaluating a single architecture at a given point in time. Although, at the end of the evaluation procedure, these methods perform comparison between candidate architectures, the comparison results are often highly dependent on the person performing the evaluation. SACAM (Software Architecture Comparison Analysis Method)[21] and DoSAM (Domain Specific Software Architecture Comparison Model) [22] have evolved for providing the rationale for an architecture selection process by comparing the fitness of candidate architectures.

SACAM, proposed by Bergey et al. [21], first derives criteria (e.g. how modifiable a software architecture is) from requirement specifications. Then, the criteria are expressed as quality attributes that are again refined into quality attribute scenarios. SACAM defines a standard architectural view (or an architectural style) and specifies candidate architectures with respect to that standard view, so that they can be compared at a common level of abstraction. SACAM then extracts metrics (such as, number of modules containing communication protocol dependencies) from candidate architectural views on the basis of quality attribute scenarios. Then, compares the metrics of all the candidate architectures, and scores their fitness. Based on the scores, SACAM provides recommendations about the suitability of candidate architectures. The recommendations help the developers to choose an appropriate product line architecture.

As SACAM compares candidate architectures from different application domains, it might be difficult to view them at the same level of abstraction. Therefore, Bergner

et al. [22] proposed DoSAM, restricting the scope of SACAM in a specific domain.

DoSAM creates a *Domain Architecture Comparison Framework (DACF)* which comprises five components. The first two components are *common architecture blue print* and *architectural services*. While the *common blue print architecture* is treated as a conceptual architectural model of an application domain, *the architectural services* are used to describe the basic functionality and the purpose of the application domain in an abstract manner. An abstract architectural service consists of a certain number of hardware and software components, e.g., the data transfer service of a network-centric system. These two components together form an abstract description schema for all architectures in the application domain. An architectural style or a design pattern can be exploited to meet the purpose of these two components (as with SACAM).

The third and fourth components of this framework are *Relevant Quality Attributes* and the corresponding *Quality Attribute Metrics*. DoSAM elicits scenarios to illustrate the relevant quality attributes. It determines the metric for each quality attribute; e.g., for the availability quality attribute, a relevant metric would be mean time to failure. The last component of the *DACF* is *Quality Computation Weights*. This component relates architectural services and quality attribute metrics. It is provided to express the relative importance of a quality attribute metric on an architectural service for a particular application. For example, the evaluation team might decide that availability of the data transfer service is very important in a certain application domain.

Once the DACF is developed, first, concrete architecture evaluation is performed in the domain of DACF. The concrete architecture is mapped to the *common architecture blue print* for determining a set of components and their connections. In this step, one or more high-level architectural overview diagrams are created and the architectural services of the concrete architecture are identified. In the second step, architectural services are examined and assessed with respect to identified quality attributes. This is done by employing quality attribute metrics. Each of these metrics yields a single evaluation result on a normalized scale from 0 to 100. In the third step, when all architecture services have been assessed with respect to all identified quality attributes, the evaluation results are entered into a weighted quality computation matrix. Again, this leads to a single, normalized evaluation result, characterizing the fitness of a candidate architecture compared to others.

4.1.10 Comparison among the Scenario-based Evaluation Methods

In the previous section, we have provided an overview of several scenario-based evaluation methods. While presenting the methods, we have explained how each method is

different from others, especially from SAAM. In this section, we provide a summarized comparative study. We present two forms of comparison. First, we compare the methods using seven comparison criteria, which we perceive to be important in comparing the methods. Second, we determine questions for investigating the methods. We limit the answer to the questions to yes ('$\surd$') or no ('X'), which will help show the similarities and differences between various methods. The two different ways of comparing scenario-based evaluation methods are discussed as follows.

Comparison criteria: We have determined seven comparison criteria to compare ten scenario based evaluation methods. In the following we provide a brief description of the comparison criteria.

- **Specific Goal(s):** Here, we specify the method's goal. For example, one method can check for the suitability analysis of an architecture while another method might support architectural tradeoff analysis, usability analysis, maintenance analysis, risk assessment and/or comparison of candidate architectures.

- **Method's Activities:** Here, we show how a particular method performs its evaluation from the beginning to the end of the analysis. Although most scenario-based methods are similar at a coarse-grained level, there are significant difference at a finer-grained level.

- **Scenario Classification:** Different methods classify scenarios differently. Here we show the commonalities and differences between the scenario classifications of different scenario-based methods. For example, some methods classify scenarios as direct and indirect, while others employ use-case scenarios or growth scenarios.

- **Scenario impact analysis:** Different methods use different strategies for estimating the impact of considered scenarios on a software architecture. With this criterion we explore strategy that each method follows for estimating the impacts of its scenarios. For example, one method might count the number of components affected by a particular scenario, while another might use metrics for each quality attribute.

- **Approaches used:** Although all the scenario-based methods follow the same approach of using scenarios for evaluation and analysis, they elicit and apply the scenarios in different ways. Moreover, depending on the target quality attribute(s) and the goals of the analysis, some methods use hybrid approaches. For example, one method might use both scenarios and mathematical models in its analysis.

- **Objects analyzed:** All the methods use software architecture documentation in performing their analyses. However, the required level of details and the views of the software architecture vary from one method to another. Some methods require only the component -level architectural specification and the logical view of the architecture for the analysis, while others may need detailed specifications for different architectural styles, design patterns, and multiple architectural views.

- **Target Quality Attributes:** This criterion is used to check what quality attribute(s) is/are targeted by a particular method. For example, some methods target only one quality attribute (e.g., modifiability) while others target multiple quality attributes (e.g., modifiability and performance).

Based on the above criteria we provide a comparative summary of the different scenario-based evaluation methods in Table 6.

Comparison using some distinguishable questions: We use 19 properties to characterize the 10 scenario-based evaluation methods. Each property is associated with a question. If the answer to a particular question is yes for a method, we say that the particular method supports the property, whereas if the answer is no, we say that the method does not support the property. We use the symbol '$\surd$' if the answer is yes, and the symbol 'X' if the answer is no. As some methods partially support tools, we we use the symbol 'P' for those cases. The comparison among different methods is shown in Table 7.

4.2 Mathematical Model-based Software Architecture Evaluation

Most scenario-based software architecture evaluation methods (with the exception of ATAM and SBAR) use qualitative reasoning for assessing development-time quality attributes. However, to measure the fitness of the safety-critical software systems, such as medical, aircraft, and space mission, it is also important to quantitatively assess operational quality attributes. Therefore, a number of mathematical model-based software architecture evaluation methods have been developed. These methods model software architectures using well-known mathematical equations. Then, these methods use the models to obtain architectural statistics, for instance, *mean execution time* of a component. These architectural statistics are used to estimate operational quality attributes. Reliability and performance are two important operational quality attributes. To assess these two quality attributes a wide range of mathematical-models

have been developed. In contrast, very few mathematical models exist for other quality attributes, such as security.

In the following, we first discuss different approaches for assessing reliability of a software architecture. Then, we discuss the approaches for predicting performance at the architectural level.

4.2.1 Software Architecture-based Reliability Analysis

According to ANSI [3], a software system's reliability is defined as the probability of the software operating without failure for a specified period of time in a specified environment. Reliability is defined in terms of the mean time between failures or its reciprocal, the failure rate. Software failures may occur for several reasons: errors and ambiguities in architectural design, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems [75, 124]. To reduce the probability of software failures, different reliability

models have been developed over the past two decades.

Early reliability models are based on reliability engineering, particularly hardware reliability. Such approaches make use of extensive experience and provide advanced mathematical formalism for building software reliability models. These models complement testing by providing an estimate of a program's ability to operate without failure. *Software reliability growth models* (SRGMs) [42, 139] fall into this category. SGRMs characterize the behavior of a software system as a black-box. Only the system's interactions with the outside world are modeled, without taking into account its internal structure. However, these models are applicable to the very late stages of a software life-cycle, ignore information about reliability of the components from which a software is composed, and do not take into consideration the architecture of the software [57].

With the widespread use of object-oriented systems design and web-based development, the use of component-based software development is increasing. Software components can be COTS (commercial-off-the-shelf), developed in-house, or developed contractually. The whole application is developed in a heterogeneous fashion (multiple teams in different environments), and hence it may be inappropriate to model the overall failure process of such applications using existing SRGMs. These heterogeneous systems, where components having different workloads and failure behaviors interact, are now commonly used [101]. Thus, it is essential to predict the reliability of an application by taking into account information about its architecture. The advantage of these architecture based reliability models is that they help understand how a system's reliability depends on the reliability of its components and the reliability of the components' interactions.

Goseva-Popstojanova and Trivedi [103] classify approaches for architecture-based software reliability assessment into three categories: state-based, path-based and additive. Earlier research efforts in this area concentrated on the state-based approaches [30, 82, 30, 80, 57], whereas more recently path-based [120, 78, 143] and additive approaches [40, 141] have been proposed. Yacoub et al. [143] proposed a reliability estimation model named Scenario-Based Reliability Analysis (SBRA) that exploits scenarios to construct a probabilistic mathematical model called Component Dependency Graph (CDG). These architecture-based reliability models [30, 82, 30, 80, 57, 120, 78, 143] are white-box approaches that estimate software reliability by taking into account the component structure of software architecture. These models calculate a system's reliability as a function of components reliability and components' interactions reliability, assuming that components and their interactions reliability are known. Therefore, these models are applicable for a software architecture whose components' implementations

Figure 10: CFG of an example application (adapted from Gokhale [57])

are available.

Roshandel et al. [107, 108] show that architecture-based reliability analysis should not assume that components and their interactions reliability are known. They argue that at the architectural level, the operational profile of a component may not be available. They proposed a new technique to estimate reliability at architectural level where implementations of architectural components are not available.

While there have been numerous software architecture-based reliability approaches, all the approaches model the software architecture and its failure behavior. In the following we discuss the general approach to model a software architecture and its failure behavior.

Modeling Software Architecture: All the approaches model behavior of a software architecture in terms of the interactions of its constituent components. The components interact by transferring execution control from one to the other. During the early design phase, each component is investigated to find its interactions with other components. When the control flows between two components, it is described by non-zero transition probability. Often a probabilistic Control Flow Graph (CFG) is used to model the interactions of all components.

Figure 10 shows the probabilistic CFG of an example application. In the figure, each node represents a component and edge represents flow of controls between components. P_{ij} is the probability that the control is transferred to component j upon the completion of component i.

Modeling Failure Behavior: Architecture-based reliability analysis approaches establish relationships between failure mechanisms of a software system and the system's underlying software architecture. These approaches define failure behavior of a software architecture with respect to failure behavior of its components and their interfaces. A component failure may occur during its execution. A Component's interfaces failure may occur during the transfer of control between two components. Components and their interfaces failures can be specified in terms of their reliability or failure ratings.

Although many techniques have been proposed for estimating component reliability, there is little information available about interface failures, apart from the general agreement that interface failures are different from component failures [136]. Most architecture-based reliability models assume that components' interfaces are perfectly reliable. Few approaches, such as approaches of Cukic [36] and Littlewoods [86] have considered interface failures. In the following we survey different approaches for estimating a component's reliability.

Based on failure data obtained during testing, SGRM can be applied to each component. However, due to the scarcity of failure data, it is not always possible to apply SGRM. Another technique is to estimate a component's reliability from explicit consideration of non-failed execution, possibly together with failure. Here, testing is not an activity for discovering fault but an independent validation activity [102]. The problem with these models is that they require a large amount of execution data to establish a reasonable confidence in the reliability estimate. The fault injection technique [135] is another well-known technique to estimate a component's reliability. However, the effectiveness of fault-injection depends on the range of classes that can be simulated.

All the above mentioned techniques require the implementation of a component to estimate its reliability. This might be a hindrance towards estimating the overall system's reliability at architectural level. To overcome this problem, Roshandel et al. [107] leverage architectural specification to estimate a component's reliability without having its implementation. In this work, the component's structure and intended behavior of a component are modeled from the architectural specification to estimate the component's reliability. A component can be modeled from four perspectives in order to analyze its structure, behavior and non-functional properties [108]. The four perspectives are interface, static behavior, dynamic behavior and interaction protocol.

In the following, we discuss the path-based and the state-based approaches while ignoring the additive approach as it is not directly related to software architecture.

Path-based Analysis: Path-based approaches model a software architecture using the probabilistic CFG, a sample of which is shown in Figure 10. These models compute

software reliability considering the possible execution paths of a program. A sequence of components along different paths is obtained either experimentally or algorithmically. Reliability of each path is determined as the product of the reliability of the components along that path. For the example application shown in Figure 10, $1- > 3- > 5- > 8- > 10$ is a possible execution path, and its reliability is given by *R1R3R5R8R10*. Then, the system reliability is estimated by averaging the reliability of all the paths. One of the major problems with the path-based approaches is that they provide only an approximate estimate of application reliability when the application architecture has infinite paths due to the presence of loops. For example, in the path $1- > 4- > 6- > 8- > 4^{1...*}- > 10$, the sub-path $4- > 6- > 8- > 4$ can occur an infinite number of times.

There are different kinds of path-based models: Shooman's model [120], Krishnamurthy and Mathur's model [78] and Yacoub, Cukic and Ammar's model [143]. Some information about these models is provided in Table 8

Table 8: **Overview of Different Path-based Models**

Model name	Approach Used	Model Parameters
Shooman model	Assumption based	Possible execution paths of the program, failure possibility of each run, frequency with each path run.
Krishnamurthy and Mathur model	Experimental based	The component trace of a program for given test case, reliability of each component in a given test case.
Yacoub, Cukic and Ammar	Tree traversal algorithm based	A tree representation of software architecture where reliability of each node and edge (component) are known.

State-based Analysis: In state-based approaches, the probabilistic CFG of an application is mapped to a state space model. These models assume that transfer of control between components has a Markov property, i.e. the execution of a future component only depends on a current component but not on the other previously executed components. These models consider software architectures as a discrete Markov chain (DTMC) or a continuous time Markov chain (CTMC) or a semi-Markov process (SMP). The DTMC represents applications that operate on demand, while the CTMC and SMP are well suited for continuously operating software applications.

Path-based approaches represent the failure behavior of components using their probability of failures or reliabilities. In contrast, state-based approaches allow component failure behavior to be represented using three types of failure modes: probability of failure or reliability, constant failure rate, and time dependent failure intensity. These

failure modes can be viewed to form a hierarchy, as far as the level of detail that can be incorporated and the accuracy of the reliability estimate are produced.

State-based approaches can be further classified into absorbing (if it is impossible to leave a state) and irreducible (it is possible to get to any state from any state). These approaches assess reliability either by solving the composite model that combines software architecture with failure behavior (composite models), or by superimposing failure behavior on the solution of the architectural model (hierarchical models).

There are various kinds of state-based models such as, Cheung's model [30], Kubat's model [80], Laprie's model [82], and Gokhale and Trivedi's model [56]. Some information about the four models are given in Table 5.

Table 9: **Overview of Different State-based Models**

Model name	Model Solution	Model for software architecture	Model Parameters
Cheung's model	Composite	Absorbing DTMC	Transition probabilities, reliability of each component
Kubat's model	Hierarchical	SMP	Constant failure intensity and deterministic execution time of a component
Gokhale's model	Hierarchical	Absorbing DTMC	Time dependent failure intensity, expected number of executions of each component
Laprie's Model	Hierarchical	Irreducible CTMC	Transition probabilities, mean execution time and constant failure intensity of each component

Limitations of architecture-based reliability analysis approaches: There are several limitations of architecture-based reliability approaches. State-based approaches assume that components are executed in a sequential manner. So these approaches do not take into account the concurrent execution behavior of components. However, there are many real-time software systems where components are executed concurrently. Another limitation is that these models use the variants of Markov models to model the software architecture. But there are many applications which cannot be modeled using Markov models, particularly applications which determine which component is the next to be executed based on execution history. State-based approaches also assume that failures of a component occur independent of the failures of other components.

Further study is required to overcome these problems of architecture-based reliability analysis approaches. However, several steps have been taken to address some of these problems. For example, to take into the concurrent execution of software components, a high-level specification mechanism such as Stochastic Reward Net [57] has been introduced.

Figure 11: Software architecture-based performance analysis

4.2.2 Software Architecture-based Performance Analysis

Software architecture plays an important role in meeting a software system's performance. Performance depends largely on the frequency and nature of inter-component communication and the performance characteristics of the components themselves. Different software architecture-based methodologies have been developed to predict performance attributes, such as *throughput*, *utilization of resources*, and *end-to-end latency*.

Architecture-based performance analysis methodologies transform the specification of a software architecture into desirable models. Then, timing information is added to these models. After that, they are analyzed to estimate performance attributes quantitatively and to provide feedback about the software architecture.

These methodologies work based on availability of software artifacts, such as requirement and architecture specifications and design documents. Since performance is a runtime attribute, these methodologies require suitable description of the dynamic behavior of a software system. Often, automatic tools are used to perform performance analysis once the performance models are created. The general framework for analyzing performance at architectural level is shown in Figure 11. Some of the advantages of architecture-based performance analysis methodologies are as follows:

- They can help predict the performance of a system early in the software life cycle.
- They can be used to guarantee that performance goals are met. They can also be used to compare the performance of different architectural choices.
- They can help in finding bottleneck resources and identifying potential timing problems before the system is built.

However, the use of the architecture-based performance analysis methodologies has been low in the software industry. Kauppi [70] has mentioned three possible reasons

for this low use. First, managing performance at the architectural level is time consuming. Second, performance models required by architectural analysis methods are complex and expensive to construct. Third, estimating resource requirements in the early software system development phase is often difficult.

Different research groups have proposed methodologies, often based on previously developed methodologies. In the following, we discuss some of the important methodologies, presenting them according to their evolution hierarchy.

Smith and William's approaches: In 1990, Smith introduced a design methodology called Software Performance Engineering (SPE) [123]. SPE is considered to be the first comprehensive approach for integrating performance analysis into the software development process. The SPE method is based on two models: *the software execution model* and *the system execution model*.

A *software execution model* works on the basis of execution graphs and represents the software execution behavior. An execution graph consists of nodes and arcs where nodes represent the processing step/elements/components and arcs represent the order of processing. The execution graph contains timing information of resource requirements. The architect provides values for the requirements for each processing step in the model. A sample execution graph is shown at the left side of Figure 12, where n represents the number that *getRequest* and *processRequest* steps have processed and t_n denotes the time that the step requires service from the server (such as processor). Analyzing the software execution model means reducing the model into a quantitative value and comparing the value t (shown at the right side of Figure 12) with the maximum allowed value specified in performance objectives.

The *system execution model* represents the model that utilizes both software and hardware information. This is formed by combining the obtained results of the software execution model with the information about hardware devices. This model is developed based on the well-known performance model *Queuing Networks Model* (QNM) [122]. A QNM is represented as a network of queues. This model is evaluated through analytical methods or simulation for obtaining the quantitative estimation of performance attributes.

In 1993, Smith and Williams [122] extended SPE with specific techniques for evaluating the performance of object-oriented systems. This work focused on early life-cycle issues and introduced use-cases as the bridge between object-oriented systems and SPE. However, it did not specifically address issues related to larger architectural concerns. Therefore, in 1998 the authors proposed a more generalized tool-based approach [138] by extending SPE and their earlier approach [122].

Figure 12: An example execution graph (from Kauppi [70])

Williams and Smith's [138] approach addresses the use of SPE for making architectural tradeoff decisions. This approach demonstrates that software execution models are sufficient for providing quantitative performance data for making architectural tradeoff decisions, and other SPE models are appropriate for evaluating additional facets of architectures. In this approach, Kruchten's 4+1 views are used to collect information, which are required to document a software architecture from different perspectives. This approach has also used the $SPE \cdot ED^{TM}$ performance engineering tool to automate the architectural analysis.

The $SPE \cdot ED^{TM}$ tool draws a software execution model from a use-case scenario depicted using a Message Sequence Chart (MSC) [66]. MSC describes the dynamic behavior of a software architecture in response to a scenario. MSC is similar to a UML sequence diagram. $SPE \cdot ED^{TM}$ collects a specification of computer resource requirements for each software resource, in the form of an overhead matrix, and stores the matrix information in a database for later reuse. Then, the tool produces solutions for both the software execution model and system execution model from the provided resource requirement specifications.

In 2002, Williams and Smith introduced a method for performance assessment of software architectures (PASA) [137]. PASA extends the above mentioned approaches and includes architectural styles and performance anti-patterns (which negatively impact the performance of a software architecture) as analysis tools. The basic difference between SPE and PASA is that SPE aims to construct and design software systems to meet performance objectives, whereas PASA aims to determine whether a software system will meet its performance objectives. PASA also formalizes the architecture assessment process based on the general software performance engineering process. It integrates the concept of SAAM and ATAM in SPE. Like SAAM and ATAM, PASA uses scenarios (particularly performance scenarios) to provide insight into how the soft-

ware architecture satisfies quality goals. PASA expresses scenarios formally using an architecture description language such as UML sequence diagram. In contrast, SAAM and ATAM express scenarios as informal narratives. Like ATAM, PASA also concentrates on extracting and evaluating architectural approaches, but does not make use of the performance ABAS as an analysis tool. PASA uses the $SPE \cdot ED^{TM}$ tool to automatically generate a QNM and to perform quantitative analysis.

The PASA method is the only performance-based software architecture analysis method that provides a framework for the whole assessment process. The available tools help speed-up the assessment process. Kauppi [70] conducted a case study using PASA where he selected it as the most suitable method for analyzing mobile communication software systems. In this case study, a *Layered QNM* (LQNM) was used instead of QNM in order to support concurrent scenarios and layered system.

Cortellessa and Mirandola's approach:[35] Cortellessa and Mirandola propose a methodology combining information from different UML diagrams to generate a performance model of a software architecture. This method follows the methodologies of SPE for performance modeling. It specifies a software architecture by using deployment, sequence, and use-case diagrams. This approach is a more formal extension of the William and Smith's approach [138]. The key contribution of this methodology over the William and Smith's approach [138] is that it adds performance evaluation related information to the considered UML diagram and obtains an Extended QNM (EQNM).

Balsamo et al., Aquilani et al., and Andolfi et al.'s Approaches: The approaches of William and Smith [122, 138] (except PASA [137]) explicitly model a software architecture that becomes part of the software life cycle. The software architecture contributes to the construction of the QNM and its workload. However, these approaches do not measure the performance of the software architecture itself; rather they use the software architecture description to derive the performance model of the final software system. Moreover, none of the approaches of William and Smith consider the concurrent or non-deterministic behaviors of the components while modeling the QNM. To address these problems of William and Smith approaches [122, 138], Balsamo et al. [9] and Aquilani et al. [4, 5] proposed new methods. The two methods are described as follows:

In 1998, Balsamo et al. [9] proposed a method that automatically derives a QNM from a software architecture specification. The software architecture is described using the CHemical Abstract Machine (CHAM) formalism.

An algorithm has been devised to derive a QNM from the CHAM specification of a software architecture. The QNM is constructed by analyzing the interactions among

the system components and among the customers and system which are represented using a Labeled Transition System (LTS) graph. The LTS represents the dynamic behavior of the CHAM architecture, and can be automatically derived from the CHAM specification. In the LTS, nodes are states, arcs are transitions between states, and labels are transition rules that permits state transitions. The algorithm is organized into two sequential phases. In the first phase, all the LTS paths are examined to single out all the pairs of components that are involved in an interaction. In the second phase, the obtained interactions pairs are used to derive the QNM. Finally, the solution of the QNM is obtained by analytical methods or by symbolic evaluation [9].

However, the problem with Balsamo et al.'s approach is that it can only analyze a restricted set of possible interaction patterns among the architectural components. For supporting more complex interaction patterns (concurrent and non-deterministic behaviors of the components) Aquilani et al.'s [4, 5] approach has evolved. For incorporating the complex patterns, these approaches are first formulated independent of a specific architecture description language (ADL), but relying on a finite state model representation. Then, the QNM is extended to deal with the complex interaction patterns. However, this approach was turned out to be inefficient in its computational complexity due to possible state space explosion of the finite state model of the architecture description. To overcome this drawback, Andolfi et al. [2] proposed a methodology based on the approaches of Balsamo et al. [9] and Aquilani et al. [4].

Andolfi et al.'s approach automatically derives a QNM from MSCs. This approach uses an algorithm that automatically transforms MSCs to QNM. This algorithm encodes MSCs by means of regular expressions. It then analyzes the regular expressions to find out their common prefix and to identify interaction pairs. Interaction pairs can give information on the real concurrency between components. These interactions pairs are then analyzed to obtain QNM. Finally, this approach performs model evaluation and provides feedbacks about the software architecture.

A summary of some relevant features of the above discussed approaches is shown in Table 8. In the table, we mark the EG and LTS for the SPE and Balsamo et al., and Aquilani et al. [4] approaches in order to represent that the EG and LTS are not the formal architecture specification languages. We also use SA as the abbreviation of software architecture.

Limitations of architecture based performance analysis approaches: Various tools have been proposed or used to implement some steps of the proposed approaches. However, none of them have yet been implemented into a complete environment for specification, performance analysis and providing feedback to the software

Table 10: **Overview of Different Architecture-based Performance Analysis Approaches**

Methodology	SA design or evaluation methodology?	SA specification language	Tool support?	Support concurrent scenarios?	Performance model
SPE [123]	design	No particular ADL and EG*	No	No	QNM
William and Smith [138]	design	MSC- UML: Deployment, Sequence and Class diagrams	Yes	No	QNM
PASA [137]	evaluation	MSC- UML: UCD, Sequence and Class diagrams	Yes	No	QNM
Cortellessa and Mirandola [35]	design	UML: Deployment, Sequence and Use Case diagrams	No	No	EQNM
Balsamo et al. [9]	evaluation	CHAM and LTS*	Yes	No	QNM
Aquilani et al. [4]	evaluation	No specific ADL and LTS*	Yes	Yes	QNM
Andolfi et al. [2]	evaluation	MSC - UML: Sequence diagram	Yes	Yes	QNM

designer. An open problem and challenge is to completely automate the process of deriving performance models from software specification and to integrate the supporting tools in a comprehensive environment.

4.3 Analysis of Early Architecture Evaluation Methods

In this section, we have presented several forms of early evaluation methods for software architecture. We now contrast the strengths and weaknesses of these approaches.

Scenario-based software architecture evaluation methods appear to be the most broadly applied methods for assessing development-time quality attributes at the early stage of software development. Some scenario-based methods, particularly SAAM, ATAM and ALMA, have been successfully applied in different industrial settings.

Scenario-based evaluation methods basically use change scenarios and scenario interactions to expose potential problem areas in the architecture. These methods measure the risks of a software system by estimating the degree of changes that a software architecture requires to implement a scenario. This risk assessment is influenced by different factors, such as coverage and complexity of scenarios, domain knowledge and objectives of stakeholder's, and consideration of quality attributes' interactions.

In scenario-based methods, it is hard to assess scenario coverage. Scenario coverage means to what extent scenarios can cover quality of a software system. Future

research should focus on developing a framework or methodologies that will help developers determine the scenario coverage. SAAMCS is a good step towards having such framework.

Most scenario-based methods express scenarios as informal narratives that make it difficult to automate the activities of scenario-based evaluation methods. Very few scenario-based methods are tool-supported; e.g., only the activities of SAAM and ATAM are even partially supported by tools.

Mathematical model-based evaluation methods transform the specification of a software architecture into well-known mathematical models. These methods also use scenarios to identify the execution paths of a software system and then examine the paths in detail using the mathematical models. Many mathematical models exist, particulary for assessing reliability and performance. Performance-based evaluation approaches appear to be more matured than those for reliability. Performance-based approaches are mostly tool-supported though none of the tools can support the complete architectural analysis process. Performance-based evaluation approaches also consider concurrent and non-deterministic behavior of software components. In contrast, reliability-based evaluation approaches do not have that much tool-support and these methods are still evolving to support concurrent and non-deterministic behaviors of software components.

The basic difference between the scenario-based and mathematical model-based evaluations is that scenario-based evaluation does not require any implementation-oriented data, whereas the mathematical model-based evaluation requires the data from the past execution history of the architectural components. Moreover, mathematical model-based evaluation is well suited for component-based software system. In contrast, scenario-based evaluation methods can be applied to any type of software systems. Due to the difficulty of converting a software architecture into a mathematical model, the use of mathematical model-based architectural evaluation methods is comparatively lower than scenario-based evaluation methods.

5 Late Evaluation Methods Applied to Software Architecture

In order to fix problems and adapt to new requirements, software systems are continuously modified. The developers who work under intense time pressure and heavy work load cannot always follow the best way to implement changes. As a result the actual architecture may deviate from the planned one. Another reason of this deviation is the

change in the workforce. Many different developers usually change a typical software system. Often a core group of developers design and implement the initial version of the system and as time goes by, members of that core group leave and new developers join the group. New developers who were not part of the group that originally designed the architecture might not easily understand it and will therefore implement changes that might not follow the planned architecture. As a result, when new people work on the system, often the design of the initial system structure is not followed, leading to a further system degeneration. In order to prohibit the actual software architecture from degeneration, late software architecture evaluation method is introduced.

Late software architecture evaluation methods identify the difference between the actual and planned architectures. These methods provide useful guidelines of how to reconstruct the actual architecture, so that it conforms to the planned architecture. During the testing phase, late software architecture evaluation methods are also applied to check the compliance of the source code to the planned design. According to Fiutem and Antoniol [46], the economics of the design-code compliance verification process not only saves time in updating designs, but also improves design artifacts as well as software development and maintenance processes.

Late software architecture evaluation can use data measured on the implementation of software architecture. Metrics can be used to reconstruct the actual software architecture, allowing it to be compared to the planned architecture. Tvedt et al. [133] and Lindvall et al. [85] propose such metrics-based approaches. Some tool-based approaches [46, 94, 112] have also been developed for measuring the conformance of source code to the planned architecture. In the following, we describe these approaches.

5.1 Tvedt et al.'s Approach

Following the Goal Question Metric approach [12], Tvedt et al. propose a late software architecture evaluation method using metrics. The aim of this method is to avoid system degeneration by actively and systematically detecting and correcting deviations of the actual software architecture from the planned architecture. The activities involved in this evaluation process are shown in Figure 13, and discussed as follows:

- A perspective of the evaluation is selected, because a system can be evaluated with different goals and from different perspectives. For example, a system can be evaluated to check whether it conforms to the functional requirements, or it can be evaluated to check at what level it satisfies its non-functional requirements.

- In order to quantify the evaluation results, guidelines and metrics are defined

based on the selected perspective. For example, if the evaluation is to measure the maintenance cost of the software system, a design guideline might be: coupling between architectural components should be low. The corresponding metrics might be *"coupling between modules(CBM)"*.

- A planned architecture is defined in order to determine the deviation of the actual architecture from it. The planned architecture is obtained by analyzing the architectural requirements, the implicit and explicit architectural guidelines and design rules, and implications stemming from the use of architectural styles and design patterns. In reality, the planned architecture is more of a goal for what the architecture should look like rather than how it is actually implemented.

- Next, the actual architecture is identified. A software architecture recovery process is applied to extract software architecture from the existing source code. Different methodologies and tools have been developed to extract the software architecture. According to Pinzer et al. [100], the general approach of recovering software architecture consists of the three steps: first, the low level system representation is obtained applying recovery tool(s) such as SWAG Kit [129] and Shrimp [125]. Second, the architectural elements/components are identified by combining the domain knowledge. Third, the relationships between the architectural elements are identified to get a high level architectural representation of the system.

- After obtaining the actual architecture, deviations between the actual and the planned architectures are identified. The deviations can be violations of design rules and guidelines or values of metrics that exceed a certain threshold. The analysis team takes note of each identified deviation, the circumstances under which it is detected, and the reason the team suspects it to be a violation. If necessary, the team conducts a more detailed analysis of the deviation in order to determine its possible cause and degree of severity.

- Based on the results from the previous step, the analysis team formulates high-level change recommendations that can remove the deviations from the system. The change recommendations may contain two types change requests– change requests for source code and change requests for the planned architecture. The analysis team members do not design or implement change requests; rather they provide valuable feedback to the development team for the constructive improvement of the system.

Figure 13: Activities of a late software architecture evaluation method (adapted from Tvedt et al. [133])

- The identified changes that are implemented must be verified to ensure that the actual architecture complies with the planned one. This step repeats the process steps of identifying the actual architecture and any architectural deviations. This verification is done to make sure that the changes have been done correctly and no new violations have been introduced into the system.

The approach has been applied to the VQI (Visual Query Interface) software system. Two analyses of the architecture were performed to ensure that one developer's work conforms to the original design rules. In this case study, three categories of violations such as design pattern violation, misplaced classes and minor violations are identified. The evaluation team identified the violations as potential threats to the maintainability of the system. However, Tvedt et al. did not mention specific suggestions or means to overcome the violations.

5.2 Lindvall et al.'s Approach

Following the similar structure of the Tvedt et al. approach, Lindvall et al. [85] conducted a case study on an experience management system (EMS) written in Java. After identifying the maintainability problems in the software system, the system has been restructured as component-based system using a new design pattern. After having the restructured system, Lindvall et al. conducted the case study to verify whether the new actual software architecture fulfills the planned software architecture and whether it better fulfills the defined goals and evaluation criteria than does the previous actual software architecture. In the case study two types of comparison (e.g. new actual architecture vs. previous actual architecture and new actual architecture vs. planned architecture) are examined to better understand the software system. Three metrics are used in this case study. Two metrics are based on inter-module couplings: *CBM(m)* and coupling-between-module-classes *CBMC(m)*, and the third metric is to measure intra-module coupling (*CIM(m)*).

5.3 Tool-based Approaches

Different tool-based approaches have been developed to check the compliance of source code to the planned software architecture or design. These approaches involve algorithms that are based on source code. In the following, we briefly describe some of these approaches.

5.3.1 Fiutem and Antoniol's Approach

Fiutem and Antoniol [46] propose a tool-supported approach that investigates the compliance between design and source code in the context of object-oriented development. This approach recovers an "as is" design from the code, compares recovered design with the planned software architecture (design) and helps the user to deal with inconsistency. This approach determines the inconsistency by pointing out the regions of code which do not match with the planned software architecture. This approach has been applied to the design and code of an industrial software system (about 200 KLOC) for telecommunications.

5.3.2 Murphy et al.'s Approach

Murphy et al. [94] software reflexion models are also well known to check the compliance of source code to the planned architecture. In this approach, an architect provides a high-level model of a system that he/she expects to find in the source code. Then he extracts a source model (such as a call graph or an inheritance hierarchy) from the source code, and defines a declarative mapping between the two models. A tool then computes a "*reflexion*" model that shows where the high-level model agrees or disagrees with the source code. This approach has been applied to several cases. For example, a software engineer at Microsoft Corporation applied reflexion models to assess the structure of the Excel spreadsheet product (over 1 million line of C code) prior to a reengineering activity. Murphy et al. used a sequence of reflexion models to compare the layered architectural design of Griswold's program restructuring tool [61] with a source model consisting of calls between modules.

5.3.3 Sefika et al.'s Approach

Another well known tool-based approach is proposed by Sefika et al. [112]. It is a hybrid approach that integrates logic based static and dynamic visualizations. It helps determine design-implementation congruence at various levels of abstraction, from coding guidelines to architectural models such as design patterns and connectors, to

design principles like low coupling and high cohesion. The utility of this approach has been demonstrated in the development of μChoices [29], a multimedia operating system.

5.4 Analysis of Late Architecture Evaluation Methods

In this section, we have presented several approaches for evaluating a software architecture after its implementation. We now contrast the strengths and weaknesses of these approaches.

To achieve successful software system's evolution, late software architecture evaluation is important. The early software architecture evaluation techniques can be used to determine the best planned design for the project, while late architectural evaluation process can be used to ensure that the planned design is carried out in the implementation. Late software architecture evaluation aims to provide an inexpensive and quick means for detecting violations to the software architecture with the evolution of software systems. As a result, this evaluation methods are mostly tool-supported.

Future work is needed to see how late software architecture evaluation methods fit for a wider set of industrial cases. Moreover, the metrics-based approaches have been only used to evaluate the software architecture with respect to maintainability perspective. It would be interesting to use these approaches from other perspectives like security.

Analyzing design-code consistency is one of the important parts in late software architecture evaluation. Although much work has done in this area, no formal framework and taxonomy exists to analyze design-code inconsistencies and prioritize the interventions to make design up-to-date. Therefore, future work should focus on devising such a formal framework and taxonomy.

6 Early Evaluation Methods Applied to Software Architectural Styles or Design Patterns

Modern software architecture is often composed from architectural styles and design patterns for handling different quality attributes, such as maintainability, reusability and evolvability. Software architecture evaluation methods evaluate a software architecture for specific cases, e.g. for specific quality requirements, such as modifiability. They often assume that strengths and weaknesses of its building blocks (architectural styles and design patterns) are known. Therefore, these methods do not incorporate

any experimental study to determine strengths and weaknesses of architectural styles or design patterns. For example, ATAM maps the extracted architectural styles to corresponding ABASs for evaluating them. To address this problem, techniques have been proposed for evaluating architectural styles or design patterns.

Evaluation of architectural styles or design patterns employ both quantitative and qualitative reasoning to motivate when and under what circumstances architectural styles or design patterns should be used. In order to differentiate the classes of designs, these analyses require experimental evidence of how each class has been used.

As with early software architecture evaluation, evaluation of architectural styles or design patterns also involves scenarios and mathematical models. Moreover, controlled experiments have been used to determine the properties of some design patterns. Klein et al. [77] propose a framework called attribute based architectural style (ABAS) to reason about architectural decisions with respect to a specific quality attribute. Kazman et al. [71] propose a scenario-based method named cost benefit analysis method (CBAM) as an extension of ATAM to analyze the costs and benefits of architectural decisions. Mathematical models have been developed to evaluate particular architectural styles or design patterns. For example, Gomaa and Menascé [59] have investigated the client-server design pattern, whereas Petriu and Wang [99] have evaluated the performance of a significant set of architectural patterns (pipe and filters, client-server, broker, layers, critical section and master-slave). Wang et al. [136] have developed an analytical model to estimate the reliability of a heterogeneous architecture consisting of batch-sequential/pipeline, call-and-return, parallel/pipe-filters and fault tolerance styles. Junuzovic and Dewan [68] conducted an experiment to evaluate three well known architectural styles, client-server, peer-to-peer and hybrid, comparing their response times in multi-user collaborations. This work also resulted in a formal performance model focussed on response time. Golden et al. [58] have introduced USAP (Usability Supported Architectural Pattern) to evaluate cancelation usability patterns.

In the following, we briefly discuss some of the above-mentioned methods and controlled experiments.

6.1 ABAS Approach

In 1999, Klein et al. [77] introduced ABAS (Attribute-Based Architectural Style) offering a reasoning framework along with an architectural style. This framework has been developed based on quality attribute specific analytic models, e.g., performance, reliability and modifiability models. There have been many mature quality attributes specific analytic models (e.g., Markov model for reliability and QNM for performance)

that provide a way to establish better understanding of quality attributes. Analytical models guide the designer to comprehensively experiment with, and plan, for architectural quality requirements.

ABAS' frameworks are based on the foundational work of Garlen and Shaw [52](proposed a catalogue of architectural styles) as well as the similar work of the design patterns community (Gamma et al. [50]). However, both Garlen and Shaw [52] and Gamma et al. [50] offer heuristic reasoning to define an architectural style or design pattern. For example, in describing the layered style, Shaw and Garlan write *"if a system can logically be structured in layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations"*. According to Klein et al. [77], this is important information for the designer who is considering the use of this style. But at the same time, it does not provide a principled way of understanding when a specific number and organization of layers will cause a performance problem. The incorporated quality attribute-specific analytic models in an ABAS can answer to this dilemma. An ABAS comprises four sections: problem description, stimulus/response attribute measures, architectural styles, and analysis, which are described as follows:

- **Problem description**: describes the real world problem that an ABAS helps to reason about and solve. The problem description includes a description of the criteria for choosing an ABAS, including when an ABAS is appropriate and the assumptions underlying the reasoning framework. For example, a performance ABAS might only be appropriate for calculating worst-case latency but not average-case latency [77].

- **Stimulus/Response attribute measures**: these characterize the stimuli to which an ABAS is to respond and the quality attribute measures of the response.

- **Architectural style**: describes the architectural style in terms of its components, connectors, properties of those components and connectors, and patterns of data and control interactions (their topology), and any constraints on the style.

- **Analysis**: describes how the quality attribute models are formally related to the architectural styles. An architectural style can have different analysis sections to address different quality attributes and hence, can form multiple ABASs, e.g. a modifiability client-server ABAS and a performance client-server ABAS. This section includes a set of analyses, design heuristics, and a set of extrapolations - typical ways in which an ABAS is extended relating to the choice of architectural

parameters. ABASs characterize quality attribute dividing its information into three categories: external stimuli, architectural decisions, and responses. For example, for performance, the external stimuli are events arriving at the system such as messages, missiles, or user keystrokes. The architectural decisions include processor and network arbitration mechanisms, concurrency structures including processes, threads, and processors, and properties including process priorities and execution times. Responses are characterized by measurable quantities such as latency and throughput.

6.2 Petriu and Wang's Approach

This approach proposes a systematic methodology to derive LQN performance models for a heterogeneous software architecture that consists of a significant set of architectural styles such as pipes and filters, client-server, broker, layers, critical section and master-slave. This work specifies architectural styles using UML collaboration diagrams that combines UML class and sequence diagrams. This approach follows SPE methodology and generates software and system execution models by applying graph transformation techniques. Software architectures are specified using UML collaborations, deployment and use-case diagrams. The sequence diagram is used to obtain the software execution model (which is represented as a UML activity diagram).The class diagram is used to obtain the system execution model (which is represented as a LQN model). Use-case diagrams provide information on the workloads, and deployment diagrams allow for the allocation of software components to hardware sites.

6.3 Golden et al.'s Approach

Golden et al. [58] have created a Usability Supported Architectural Pattern (USAP) that can enhance software architects' responsibility and accuracy for developing large-scale software systems. Each USAP consists of three things: (i) an architectural sensitive usability scenario, (ii) a list of general responsibilities which are generated considering different factors, such as concerned usability related tasks, environment, human capabilities and desires, and software state, and (iii) a sample solution implemented in a larger separation-based design pattern. In this approach, Golden et al. have performed a controlled experiment using a single USAP that supports the cancelation usability pattern [16]. The experiment measured whether architectural solutions produced applying USAP are better than those produced by certain subsets of the USAP components, e.g., involving only the scenarios or scenarios and the relevant respon-

sibilities. However, Golden et al. commented that as USAPs are quite detailed and complex, software architects might find USAPs difficult to apply to their own design problems. The controlled experiment is briefly described as follows:

To create the USAP for cancelation commands Golden et al. provided nineteen cancelation related responsibilities and a sample solution which uses the J2EE MVC design pattern. As an example, two of the responsibilities are as follows:

- CR1: A button, menu item, keyboard shortcut and/or other means must be provided, by which the user may cancel the active command.
- CR2: The system must always listen for the cancel command or changes in the system environment.

For conducting the experiment, 18 graduate students with different level of programming experience were selected. The participants were divided into three groups, each randomly assigned one of the three components of the USAP. Participants in the first condition were provided the first component of the USAP, i.e., the usability scenario for the cancelation command. Participants in the second condition were provided the first two components of the USAP, the scenario and a list of general responsibilities. Finally, participants in the third condition received all the three components of the USAP. Participants in each condition received a different version of a "Training Document". All participants received the same architecture redesign task and proper training. After assigning tasks properly to each of the three groups, the experiment results were obtained. The experimental results showed that the group which was provided with USAP took maximum time to perform the task. But this group obtained better solution than the other two groups, because this group considered most of the cancelation responsibilities. This indicates that USAP provided significant help to the participant in remembering the responsibilities they needed to consider when modifying the software.

6.4 Analysis of Early Architectural Styles or Design Patterns Evaluation Methods

In this section, we have presented several approaches for evaluating architectural styles or design patterns before their implementation. We now contrast the strengths and weaknesses of these approaches.

ABAS is a good attempt to make architectural design more of an engineering discipline, where design decisions are made upon the basis of known properties and well-

understood analyses. Future work should be concentrated on making a handbook with many ABASs. The handbook can serve as pre-packaged design and/or analysis wisdom.

USAP is a valuable tool for evaluating design patterns with respect to usability quality attributes. However, more work needs to be done to increase the consistency with which software architects apply the USAPs, perhaps in the format of the USAP itself or in training provided with it. Additionally, Golden et al.'s study only used USAP for the cancelation usability scenario. But there are also other usability scenarios which require USAPs. Therefore, future investigations are required to determine how Golden et al.'s experimental results can be replicated and extended across additional USAPs.

7 Late Evaluation Methods Applied to Software Architectural Styles or Design Patterns

Many architectural styles and design patterns have been applied in different categories of software applications, but there is little evidence about their strengths and weaknesses. Books and research papers address the usefulness of the architectural styles or design patterns, but they did not provide any empirically validated data to show the effectiveness of architectural styles or design patterns. Evaluation of software architectural styles or design patterns after their implementation has been introduced to verify their usage.

To the best of our knowledge only a single experiment by Prechelt et al.[104], has evaluated the design patterns in the context of maintenance. With this experiment, Prechelt et al. showed that theoretical hypotheses about design patterns can be different in their practical use. In the following, we present Prechelt et al.'s experiment:

7.1 Prechelt et al.'s Approach

Prechelt et al. have performed a controlled experiment in software maintenance to seek empirical evidence whether the application of a design pattern is effective for solving design problems. In this controlled experiment, Prechelt et al. justified the usefulness of design patterns proposed by Gamma et al. in the context of software maintenance for a particular experimental setting. In this setting, the solution of the selected programs do not need to apply all the properties of a design pattern and can be replaced by a simpler solution which uses fewer design patterns or no design patterns. They have found some contradictory results; for example, the use of design patterns can cause more maintenance problems in a software system than the conventional solutions, or

can make the software system simpler to maintain. Having observed this tradeoff, Prechelt et al. come to the conclusion that developers should use a design pattern to solve a particular problem after comparing the solution (with the design pattern) with other alternatives. They also suggest that developers should not be biased by the popularity of the design pattern.

The experiment was conducted using four groups (total 29 participants) of professional software engineers. They were assigned to solve four different programs (*Stock Ticker* ("ST"), *Graphics Library* ("GR"), *Communication Channels* ("CO") and *Boolean Formulas* ("BO")) asking two different versions of solutions. One version of the solutions (named PAT) involves application of different kinds of design patterns, such as *Abstract Factory*, *Composite*, *Decorator*, *Facade*, *Observer* and *Visitors* (as described by Gamma et al. [50]). Another version (ALT) uses simpler design using fewer design patterns than PAT. The experiments carried out pre- and post-tests. Pre-test was conducted without providing any pattern course and post-test was performed after having the pre-test and the pattern course. In both tests each group maintained one PAT program and one ALT program with two or three tasks for each and each group worked on all four programs.

Prechelt et al. have set an expected result for each category of the experiments and then compared the expected results with the actual experimental results. They found remarkable deviations between these two. For example, while applying the *visitor pattern* in the "ST" program, they expected that with the pattern knowledge both versions (ALT and PAT) of the program can be maintained faster than the conventional solutions. But in the actual result, they have found a negative effect from unnecessary application of the design pattern, particularly for the participants with low pattern knowledge (the ALT group). The negative effect came since the ALT group was slightly slower in the post-test than the pre-test which implies that the participants without having knowledge of the *visitor pattern* became confused after getting the design pattern course. The newly learnt design pattern knowledge made the ALT group slow to solve the provided task using the *visitor pattern*. They have also got some results which match with expected results.

7.2 Analysis of Late Architectural Styles or Design Patterns Evaluation Methods

In this section, we have discussed late software architectural styles or design patterns evaluation. Very few studies have been done in this category of evaluation. We have

presented Prechelt et al.'s experiment for evaluating design patterns. We now discuss strengths and weaknesses of their experiment, and provide some guidelines for its future extension.

Prechelt et al. conducted applications using six design patterns. However, many other design patterns exist. Future study is required to investigate whether there are alternative simpler solutions for specialized applications of these design patterns. Future study should also concentrate on determining the tradeoffs involved in the design patterns. Additionally, future work should focus on addressing the following questions. What are the effects of pattern versus non-pattern designs for long term maintenance involving many interacting changes? How does the use or non-use of patterns influence activities other than pure maintenance, e.g., inspections or code reuse?

8 Summary and Discussion

In this paper, we have presented four categories of software architectural evaluation methods. The categorization is done based on the artifacts on which the methods are applied and the two phases of a software life cycle: early and late. Numerous methods or approaches [72, 73, 18, 20, 17, 49, 83, 92, 132, 127, 120, 78, 143, 30, 80, 123, 138, 137, 99, 35, 9, 4, 2] have been developed to evaluate software architectures at the early stage of software development.

At this stage, scenario-based methods [72, 73, 18, 20, 17, 49, 83, 92, 132, 127] are the most widely applied methods for assessing development-time quality attributes. There are many scenario-based early software architecture evaluation methods and they are correlated. In this paper, we have presented ten scenario based methods and compared them using seven comparison criteria and 19 properties. The comparative study can help developers select a method appropriate for their evaluation purposes.

Besides scenario-based evaluation methods, different mathematical model-based evaluation techniques [120, 78, 143, 30, 80, 123, 138, 137, 99, 35, 9, 4, 2] are developed to assess operational quality attributes at the early stage of software development. In this paper, we have discussed mathematical models, particularly for assessing reliability and performance quality attributes.

Compared to early software architecture evaluation, fewer methods [133, 85, 46, 94, 112] have been developed for late software architecture evaluation. Late software architecture evaluation helps prohibit architecture degeneration. This category of evaluation verifies that an implemented software architecture conforms to the planned software architecture. As early software architecture evaluation methods are not intended

for verifying the conformance of the implemented software architecture to the planned software architecture, these methods might not be effective for evaluating a software architecture after its implementation. Moreover, most late software architecture evaluation methods are tool-supported and are executed with respect to concrete quality requirements. In contrast, most scenario-based methods do not have tools support and are often used to deal with unpredictable quality requirements. As a result, the degree of difficulty and uncertainty for late software architecture evaluation is less than that of early software architecture evaluation. However, early mathematical model-based evaluation approaches can be well-suited to late software architecture evaluation, as most of these approaches require implementation-oriented data (except Roshandel et al.'s [106, 107] approaches) and use tools.

In this paper, we have also presented some approaches for evaluating software architectural styles or design patterns. Although there are few attempts in these categories of architectural evaluations, some of them are widely used and useful to evaluate the whole software architecture. For example, ATAM uses ABASs to analyze the extracted architectural approaches and SALUTA can use Golden et al.'s approach to determine the properties of usability patterns such as cancelation. We have presented three categories of early architectural styles or design pattern evaluation techniques: scenario-based [77, 71], mathematical model-based [59, 99] and controlled experiments [68, 58]. Very few approaches are available for evaluating architectural styles or design patterns after their implementation. In this paper, we have presented Prechelt et al.'s [104] controlled experiment for evaluating design patterns.

Evaluation of an architectural style or design pattern requires controlled experiments to determine the applicability of an architectural style or design pattern for wide range of cases. In contrast, evaluation of a whole software architecture determines the applicability of a software architecture for some specific cases. As a result, evaluation of an architectural style or design pattern requires extensive study to determine in general use of architectural styles and design patterns. Hence this category of evaluation is more difficult than evaluating a software architecture.

8.1 Comparison among Four Categories of Software Architectural Evaluation

In this subsection, we provide a higher level comparison between the four categories of methods and techniques reviewed above. For such a higher level comparison, we have determined seven comparison criteria:

- **Implementation-oriented data:** With this criterion we tried to check whether an evaluation category requires implementation-oriented data or not. The basic difference between early and late software architecture evaluation is that early software architecture evaluation does not need implementation of a software architecture, whereas late software architecture evaluation does. However, most mathematical model-based early software architecture evaluation methods need implementation-oriented data of architectural components.

- **Metrics formulation:** Some categories of evaluation are required to formulate metrics for doing quantitative analysis; e.g., metrics-based late software architecture evaluation methods incorporate metrics depending on the perspective of the evaluation. On the other hand, some evaluation categories, such as scenario-based early software architecture evaluation incorporates qualitative evaluation techniques, so they do not always require to formulate metrics. However, some early software architecture evaluation methods such as ATAM, SACAM and some mathematical model-based approaches are required to formulate metrics to assess the operational quality attributes. Early and late software architectural styles or design patterns evaluations may not involve any metrics as they involve experimental study. With the criterion *Metrics formulation*, we tried to check which category of software architectural evaluation always requires to formulate metrics and which category does not.

- **Types of methods:** With this comparison criterion we focus on the different types of evaluation techniques supported by each category of evaluation. For example, early software architecture evaluation supports four different categories of evaluation techniques, whereas late software architecture evaluation supports three types of evaluation techniques. However, we have determined the types of evaluation techniques for each category of evaluation from our observations.

- **Participation of stakeholder:** With this criterion we tried to put emphasis on the necessity of stakeholder presence during the architectural evaluation. For example, early software architecture evaluation, specially scenario-based software architecture evaluation always recommends participation of stakeholder to get a meaningful analysis result. In contrast, late software architecture evaluation may be conducted using tools without having stakeholder presence. Both early and late software architectural styles or design patterns evaluations require participation of the developers to execute controlled experiments.

- **Tool-support:** With this criterion we checked whether an evaluation category supports tools to automate the architectural evaluation activities. For example, most late software architecture evaluation methods have support for tools, whereas the methods in the other three evaluation categories have little support for tools.

- **Difficulty level:** This criterion has been chosen based on our observation in each category of evaluation. We have defined three difficulty levels: '*','**', and '***'. The degree of difficulty increases with the number of '*'. For example, we have used single '*' for late software architecture evaluation, whereas double '*' for early software architecture evaluation to indicate that difficulty level of late software architecture evaluation is less than that of early software architecture evaluation. The reason is that late software architecture evaluation has more tool supports than early software architecture evaluation. Moreover, late software architecture evaluation handles more concrete quality requirements than early software architecture evaluation. We have used triple '*' for early and late software architectural styles or design patterns evaluations. The triple '*' indicates that these categories of evaluation is more difficult compared to early and late software architecture evaluations as they involve controlled experiments which require much time and efforts.

In the following Table 11, we summarize our findings for the four categories of software architectural evaluations.

9 Open Problems

In this paper, we have presented and analyzed four categories of software architectural evaluations. We now summarize open problems that are required to be solved in this area. We also briefly discuss some solutions that partially address these problems.

One of the open problems in software architecture evaluation is that it is hard to assess coverage of scenarios. Scenario coverage means that to what extent the scenarios can address quality of a software system. There is no particular number of scenarios, the execution of which guarantees scenario coverage optimally. No method offers systematic methodologies that help elicit such important scenarios. However, coming up with such methodologies are challenging. These methodologies require to determine a complete set of factors that should include all aspects relevant to the importance of scenarios. Different steps have been taken to address this problem.

Table 11: **Comparison among Four Categories of Software Architectural Evaluations**

Comparison Criteria	Early Software Architecture Evaluation	Late Software Architecture Evaluation	Early Software Architectural Styles or Design Patterns Evaluation	Late Software Architectural Style or Design Patterns Evaluation
Implementation-oriented data	Scenario-based evaluation methods do not need data measured from implementation, but some Mathematical model-based evaluation methods do	Utilize data measured on the actual software implementation	Same as software architecture evaluation	Require empirically validated data
Metrics usage	not always	mostly	not always	not always
Types of methods	Scenario-based evaluation, Mathematical model-based evaluation, Simulation and Experience-based	Metrics based and tool-based	scenario-based, mathematical model-based and controlled experiments	Controlled experiment
Participation of stakeholder?	Required	not always required	required	required
Difficulty level	**	*	***	***
Tool-support	Usually no tool support except SAAM and ATAM and Mathematical model-based evaluation	Most of the methods have tool support	Some methods have tools	No tool support

For example, SAAMCS has introduced a two dimensional framework that offers a measurement instrument. The measurement instrument employs a number of generic classes of possibly complex scenarios and three factors that define the complexity of scenarios. This measurement instrument can determine limitations or boundary conditions of a software architecture by exposing implicit assumptions. Thus, it helps find important scenarios, missing of these scenarios may turn architectural evaluation results meaningless. However, further research should focus on finding more classes of complex scenarios. Future study also requires verification that the factors that have been determined in SAAMCS are the ones that cause the complexity of scenarios. Moreover, stakeholders with enough domain knowledge and expertise are required to elicit such complex scenarios. Therefore, future study should attempt to determine how domain knowledge and degree of expertise affect the coverage of selected scenarios.

Another technique helping to overcome this scenario coverage problem is *Quality Function Deployment* (QFD) [37, 38]. This technique helps elicit the scenarios that

can optimally address software quality. This technique generates a series of matrices to show the relative importance of quality attributes with respect to stakeholder's objectives. Two methods SAEM (Software Architecture Evaluation Model) [39] and SAAMER (Architecture Analysis Method for Evolution and Reusability) [87], use this QFD technique to resolve this scenario coverage problem.

Another problem with architectural evaluation methods is a lack of educated materials. These methods are not also self-explanatory. To understand different aspects of a method properly, practitioners need to go through other resources. Future work is needed to present the architectural evaluation methods in handbooks, where not only the methods activities but also the effects of their various usage should be explained. The book written by Clements et al. [32] is a good step towards documenting architectural evaluation methods. However, this book only covers three scenario-based software architecture evaluation methods. It can be extended to cover more varieties of evaluation methods. A handbook of ABASs can be also a promising solution for understanding aspects of architectural evaluation.

With the exception of SAAM and ATAM, software architectural evaluation methods require more validation. Most are validated by their creators using limited case studies. However, this kind of validation is not enough to convince practitioners about the effectiveness of an evaluation method. Therefore, future work is needed to validate the evaluation methods in different industrial cases.

Another open problem in architectural evaluation methods is that they have a lack of tool-support. Most are executed by human beings. These methods are difficult to apply in a complex and large software systems as they involve significant manual works. Although there are some tools available for evaluating software architecture, none of the tools supports the complete evaluation process. Therefore, a comprehensive research is required to move towards automating software architectural evaluation methods.

10 Conclusion

In this paper, we have surveyed the state of art of software architectural evaluation methods using a taxonomy. The taxonomy shows the architectural evaluation methods and techniques into the four categories: early software architecture evaluation methods, late software architecture evaluation methods, early software architectural styles or design patterns evaluation methods, and late software architectural styles or design patterns evaluation methods.

The results of this study may serve as a roadmap to the software developers and

architects in helping them select the right method or technique for their interests. We hope it may also assist in identifying remaining open research questions, possible avenues for future research, and interesting combinations of existing techniques.

References

[1] G. Abowd, L. Bass, P. Clements, Rick Kazman, L. Northrop, and A. Zaremski. Recommended Best Industrial Practice for Software Architecture Evaluation (CMU/SEI-96-TR-025). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.

[2] F. Andolfi, F. Aquilani, S. Balsamo, and P. Inverardi. Deriving QNM from MSCs for Performance Evaluation of SA. In *the Proceedings on 2nd International Workshop on Software and Performance*, pp. 2000

[3] ANSI/IEEE, "Standard Glossary of Software Engineering Terminology", STD-729-1991, ANSI/IEEE, 1991

[4] F. Aquilani, S. Balsamo, P. Inverardi.*Performance Analysis at the software architecture design level*. Technial Report TRSAL- 32, Technical Report Saladin Project.

[5] F. Aquilani, S. Balsamo, and P. Inveradi. Performance analysis at the software architectural design level. *Performance Evaluation*, vol. 45 , no. 2-3, pp. 147-178, 2001.

[6] M. A. Babar, L. Zhu and R. Jeffery. A Framework for Classifying and Comparing Software Architecture Evaluation Methods. In *the Proceedings on Australian Software engineering*, pp. 309-318, 2004.

[7] M. A. Babar and I. Gorton. Comparison of Scenario-Based Software Architecture Evaluation Methods. In *the Proceedings on Asia-Pacific Software Engineering Conference*, pp. 584-585, 2004.

[8] F. Bachmann, L. Bass, G. Chastek, P. Donohoe and F. Peruzzi. The Architecture Based Design Method. CMU/SEI-200-TR-001 ADA375851. Pittsburg, PA: Software Engieering Institute, Carnegie Mellon University, 2000.

[9] S. Balsamo, P. Inverardi and C. Mangano. An approach to performance evaluation of software architectures. In *the Proceedings on 2nd International Workshop on Software and Performance*, pp. 178-190, 1998

[10] J. Baragr and K. Reed. Why We Need A Different View of Software Architecture. In *the Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pp. 125-134, 2001.

[11] M. R. Barbacci, M. H. Klein and C. B. Weinstock. Principles for Evaluating the Quality Attributes of a Software Architecture (CMU/SEI-96-TR-036 ESC-TR-96-136). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1997.

[12] V. R. Basili, G. Caldiera, and D. H. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, vol. 2, pp. 528-532, 1994.

[13] L. Bass, B.E John, and J. Kates. Achieving Usability Through Software Architecture. Carnegie Mellon University/Software Engineering Institute Technical Report No. CMU/SEI-TR-2001-005, 2001.

[14] L. Bass, R. Nord, W. Wood and D. Zubrow. Risk Themes Discovered Through Architecture Evaluations (CMU/SEI-2006-TR-012 ESC-TR-2006-012). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 2006.

[15] L. Bass, P. Clements and R. K. Kazman. Software Architecture in Practice. SEI Series in Software Engineering. Addison-Wesley, 1998. ISBN 0-201-19930-0.

[16] L. Bass and B. John. Linking Usability to Software Architecture Patterns through General Scenarios. The Journal of Systems and Software, 66 (2003) 187-197.

[17] P. Bengtsson and J. Bosch. Scenario Based Software Architecture Reengineering. In *the Proceedings of International Conference of Software Reuse*, pp. 308-317, 1998.

[18] P. Bengtsson, J. Bosch. Architecture Level Prediction of Software Maintenance. In *the Proceedings on* 3^{rd} *European Conference on Software Maintenance and Reengineering*, pp. 139-147, 1999.

[19] P. Bengtsson. Towards Maintainability Metrics on Software Architecture: An Adaptation of Object-Oriented Metrics. In *the Proceedings on* 1^{st} *Nordic Workshop on Software Architecture*, pp. 638-653, 1998.

[20] P. Bengtsson, N. Lassing, J. Bosch, and H. V. Vliet. Architecture-Level Modifiability Analysis. Journal of Systems and Software, vol. 69, 2004.

[21] J. K. Bergey, M. J. Fisher and L. G. Jones and R. Kazman. Software Architecture Evaluation with ATAMSM in the DoD System Acquisition Context. CMU/SEI-99-TN-012. Pittsburg, PA: Software Engieering Institute, Carnegie Mellon University, 1999.

[22] K. Bergner, A. Rausch, M. Sihling and T. Ternit. DoSAM - Domain-Specific Software Architecture Comparison Model. *In the Proceedings of the International Conference on Quality of Software Architectures*, pp. 4-20, 2005.

[23] B. W. Boehm, J. R Brown, and M. Lipow. Quantitative evaluation of software quality. In *the Proceedings on* 2^{nd} *International Conference on Software Engineering*, pp. 592 - 605, 1976.

[24] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merritt. *Characteristics of Software Quality*, North Holland, 1978.

[25] J. Bosch and P. Molin. Software architecture design: Evaluation and transformation. IEEE Engineering of Computer Based Systems Symposium. IEEE Computer Based Systems, pp. 4-10, 1999.

[26] H.de Bruijn, and H. van Vliet. Scenario-based generation and evaluation of software architectures. Lecture Notes in Computer Science 2186: 128-139, 2001.

[27] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal *Pattern-Oriented Software Architecture: A System of Patterns (POSA)*, Wiley and Sons, 493 p. 1996

[28] S. A. Butler. Security attribute evaluation method: a cost-benefit approach. In *the Proceedings on International Conference on Software Engineering*, pp. 232-240, 2002.

[29] R. H. Campbell and S, Tan. μChoices: An Object-Oriented Multimedia Operating System. In *proceedings of Fifth Workshop on Hot Topics in Operating Systems*, 1995.

[30] R. C. Cheung. *A user-oriented software reliability model*. IEEE Trans. on Software Engineering, vol. 6, pp. 118-125, 1980.

[31] C.G. Chittister and Y.Y. Haimes. Systems integration via software risk management. Systems, Man and Cybernetics, Part A, IEEE Transactions on, vol. 26, pp. 521-532, 1996.

[32] P. Clements and R. K. Kazman, M. Klein. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Professional; 2002. ISBN 0-201-70482X

[33] P. Clements, L. Bass, R. Kazman, and G. Abowd. Predicting Software Quality by Architecture-Level Evaluation. In the proceedings of Fifth International Conference on Software Quality, 1995.

[34] P. Clements. Active Reviews for Intermediate Designs (CMU/SEI-2000-TN-009), Software Engineering Institute, Carnegie Mellon University.

[35] V. Cortellessa and R. Mirandola. Deriving a Queueing Network based Performance Model from UML Diagrams. In *the Proceedings on* 2^{nd} *International Workshop on Software and Performance*, pp. 58-70, 2000.

[36] B. Cukic. The Virtues of Assessing Software Reliability Early. *IEEE Software*, vol. 22, pp. 50-53, 2005.

[37] R. Day. *Quality Function Deployment. Linking a Company with Its Customers.* Milwaukee, Wisc.: ASQC Quality Press, 1993.

[38] L. Dobrica and E. Niemela. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638-653, July 2002.

[39] J.C. Duenas, W.L. de Oliveira, and J.A. de la Puente. A Software Architecture Evaluation Model. In *the Proceedings of Second International ESPRIT ARES Workshop*, pp. 148-157, 1998.

[40] W. W. Everett. Software Component Reliability Analysis. In the Proceedings on IEEE Symposium on Application - Specific Systems and Software Engineering and Technology, pp. 204-211, 1999.

[41] W. Farr. Software Reliability Modeling Survey. Handbook of Software Reliability Eng., M.R. Lyu, ed. pp. 71-117, McGraw-Hill, 1996.

[42] W. Farr. Software Reliability Modeling Survey. *Handbook of Software Reliability Engineering*, M.R. Lyu, ed., pp. 71-117, 1996.

[43] R. Fielding and R. N. Taylor. Principled design of the modern Web architecture. In *the Proceedings of International Conference on Software Engineering*, 2000.

[44] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. Ph.D. Dissertation, 2000.

[45] R. Fiutem , and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *the Proceedings of the International Conference on Software Maintenance*, pp. 94-102, 1998.

[46] R. Fiutem , and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *the Proceedings of the International Conference on Software Maintenance*, pp. 94-102, 1998.

[47] E. Folmer and J. Bosch. Architecting for usability: a survey. *Journal of systems and software*, Elsevier, pp. 61-78, , 2002

[48] E. Folmer, J. v. Gurp, and J. Bosch. Scenario-Based Assessment of Software Architecture Usability. In *the Proceedings of Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction, ICSE*, 2003.

[49] E. Folmer, J. Gurp and J. Bosch. Software Architecture Analysis of Usability. In *the Proceedings on* 9^{th} *IFIP Working Conference on Engineering Human Computer Interaction and Interactive Systems*, pp. 321-339, 2004.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns elements of reusable object oriented software*. Reading, MA: Addison Wesley, 1994.

[51] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. *Foundations of Component-Based Systems*, Leavens, G.T., and Sitaraman, M. (eds). Cambridge University Press, pp. 47-68, 2000.

[52] D. Garlan and M. Shaw. An introduction to software architecture. Ambriola & Tortola (eds.), Advances in Software Engineering & Knowledge Engineering, vol. II, World Scientific Pub Co., pp. 1-39, 1993.

[53] D. Garlan. Software Architecture: A Roadmap. In *the Proceedings on The Future of Software Engineering*, pp. 93-101, 2000.

[54] D. Garvin. What Does "Product Quality" Really Mean?. *Sloan Management Review*, pp. 25-45, 1984.

[55] S. Gokhale, W. E. Wong, K. Trivedi, and J. R. Horgan. An analytical approach to architecture based software reliability prediction. In *the Proceedings of* 3^{rd} *International Computer Performance & Dependability Symp.*, pp. 13-22, 1998.

[56] S. S. Gokhale and K. S. Trivedi. Reliability Prediction and Sensitivity Analysis Based on Software Architecture. In *the Proceedings of the* 13^{th} *International Symposium on Software Reliability Engineering*, pp. 64-75, 2002

[57] S. S. Gokhale. Architecture-Based Software Reliability Analysis: Overview and Limitations. IEEE Transactions on Dependable and Secure Computing, vol. 4, pp. 32-40, 2007

[58] E. Golden, B.E. John and L. Bass. The value of a usability-supporting architectural pattern in software architecture design: a controlled experiment. In *the Proceedings on* 27^{th} *international conference on Software engineering*, pp. 460-469, 2005.

[59] H. Gomaa and D.A. Menascé. Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures. In *the Proceedings on* 2^{nd} *International Workshop on Software and Performance*, pp. 117-126, 2006.

[60] G. Y. Guo, J. M. Atlee and R. Kazman .A Software Architecture Reconstruction Method. *In the Proceedings of IFIP Conference*, pp. 15-34, 1999.

[61] W.G. Griswold and D. Notkin. *Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool*. IEEE Transactions on Soflware Engineering, vol. 21, pp. 275-287, 1995.

[62] J. E. Henry and J.P. Cain. A Quantitative Comparison of Perfective and Corrective Software Maintenance. Journal of Software Maintenance: Research and Practice, John Wiley & Sons, Vol 9, pp. 281-297, 1997.

[63] Recommended practice for architectural description. IEEE Standard P1471, 2000

[64] M.T. Ionita, D. K. Hammer and H. Obbink. Scenario-Based Software Architecture Evaluation Methods: An Overview. *Workshop on Methods and Techniques for Software Architecture Review and Assessment at the International Conference on Software Engineering*, 2002.

[65] ISO, International Organization for Standardization, *ISO 9126-1:2001, Software engineering - Product quality, Part 1: Quality model*, 2001.

[66] ITU. Criteria for the Use and Applicability of Formal Description Techniques, Message Sequence Chart (MSC). International Telecommunication Union, 1996.

[67] K. Jensen. Coloured Petri Nets. *Basic Concepts, Analysis Methods and Practical Use* vol. 1, Springer-Verlag, 2nd corrected printing 1997. ISBN: 3-540-60943-1.

[68] S. Junuzovic and P. Dewan. Response times in N-user replicated, centralized and proximity-based hybrid collaboration architectures. In *the Proceeding on 20th Anniversary Conference on Computer Supported Cooperative Work*, pp. 129-138, 2006.

[69] N. Juristo, M. Lpez, A. M. Moreno, and M. Isabel Snchez. Improving software usability through architectural patterns. In *the Proceedings ICSE 2003 Workshop*, pp. 12-19, 2003.

[70] T. Kauppi. *Performance analysis at the software architectural level*. Technical report, ISSN: 14550849, 2003.

[71] R. Kazman, J. Asundi and M. Klein. Quantifying the Costs and Benefits of Architectural Decisions. In *the Proceedings on* 23^{rd} *International Conference on Software Engineering*, pp. 297-306, 2001.

[72] R. Kazman, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. *In the Proceedings on* 16^{th} *International Conference on Software Engineering*, pp. 81-90, 1994.

[73] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The Architecture Tradeoff Analysis Method. In *the Proceedings on ICECCS*, pp. 68-78, 1998.

[74] R. Kazman. Tool Support for Architecture Analysis and Design. In *the Proceedings of the* 2^{nd} *International Software Architecture Workshop*, pp. 94-97, 1996.

[75] P. A. Keiller, and D. R. Miller. On the Use and the Performance of Software Reliability Growth Models. *Software Reliability and Safety*, Elsvier, pp. 95-117, 1991.

[76] N. L. Kerth and W. Cunningham. Using patterns to improve our architectural vision. IEEE Software, 14(1), pp. 53-59, 1997.

[77] M. H. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci and H. Lipson. Attribute-Based Architectural Styles. In *the Proceedings on First Working IFIP Conference on Software Architecture*, pp. 225-243, 1999.

[78] S. Krishnamurthy and A. P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *the Proceedings of 8th Int'l Symp. Software Reliability Engineering*, pp. 146-155, 1997.

[79] P.B. Krutchen. The 4+1 View Model of Architecture. IEEE Software, pp. 42-50, November 1995.

[80] P. Kubat. Assessing reliability of modular software. Operation Research Letters, 8:35-41, 1989.

[81] Stephan Kurpjuweit. Ph.D. Thesis. A Family of Tools to Integrate Software Architecture Analysis and Design. 2002.

[82] J. C. Laprie. Dependability evaluation of software systems inoperation. IEEE Trans. on Software Engineering, vol. 10(6), pp. 701-714, 1984.

[83] N. Lassing, D. Rijsenbrij, and H. v. Vliet. On Software Architecture Analysis of Flexibility, Complexity of Changes: Size isn't Everything. In *the Proceedings of* 2^{nd} *Nordic Software Architecture Workshop*, 1999.

[84] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. Journal of Systems and Software, vol. 23, no. 2, pp. 111-122, November 1993.

[85] M. Lindvall, R. T. Tvedt and P. Costa. An empirically-based process for software architecture evaluation. Empirical Software Engineering 8(1): 83Y108, 2003

[86] B. Littlewood. A Reliability Model for Markov Structured Software. In *the Proceeding on International Conference Reliable Software*, pp. 204-207, 1975.

[87] C.-H. Lung, S. Bot, k. Kalaichelvan, and R. Kazman. An Approach to Software Architecture Analysis for Evolution and Reusability. In *the Proceedings on CASCON*, 1997.

[88] A. D. Marco and P. Inverardi. Starting from Message Sequence Chart for Software Architecture Early Performance Analysis. In *the Proceedings on the* 2^{nd} *International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, 2003

[89] J. McCall, P. Richards, and G. Walters. *Factors in software quality*. Vol I-III, Rome Aid Defence Centre, Italy, 1997.

[90] N. Medvidovic, D.S. Rosenblum, J.E. Robbins, and D.F. Redmiles. Modeling Software Architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology, vol. 11, pp. 2-57, 2002.

[91] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, 26(1): 70-93, January 2000.

[92] G. Molter. Integrating SAAM in Domain-Centric and Reuse-based Development Processes. In *Proceedings of the* 2^{nd} *Nordic Workshop on Software Architecture*, 1999

[93] R.T. Monroe, A. Kompanek, R. Melton and D. Garlan. Architectural Styles, Design Patterns, and Objects. *IEEE Software*, vol. 15, no. 7, pp. 43-52, January 1997.

[94] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *the Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pp. 18 - 28, 1995.

[95] E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proceedings on International Conference on Software Engineering*, pp. 13-22, 1999.

[96] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), Dec. 1972, pp. 1053-1058.

[97] D.E. Perry and A.L.Wolf. Foundations for the Study of Software Architecture. In *the Proceedings on Software Engineering Notes, ACM SIGSOFT*, pp. 40-52, October 1992.

[98] D. Petriu, C. Shousha, and A. Jalnapurkar. Architecture-Based Performance Analysis Applied to a Telecommunication System. *IEEE Transactions on Software Engineering*, vol. 26, no.11, pp. 1049-1065, 2000.

[99] D. Petriu and X. Wang. From UML descriptions of High-Level Software Architectures to LQN Performance Models. In *the Proceedings on Inernational Workshop on Applications of Graph Transformations with Industrial Relevance*, pp. 47-62, 1999.

[100] M. Pinzger, H.Gall , J. F. Girard, J. Knodel, C. Riva, W. Pasman, C. Broerse and Jan G. Wijnstra. Architecture Recovery for Product Families. Book Chapter: *Software Product-Family Engineering*, LNCS, Springer, vol. 3014, 2004.

[101] K. Goseva-Popstojanova, K. Trivedi, and A. P. Mathur. How different architecture based software reliability models are related? In *the Proceedings of* 11^{th} *International Symposium on Software Reliability Engineering*, pp. 25-26, 2000.

[102] K. Goseva-Popstojanova, A.P. Mathur, K.S. Trivedi. Comparison of architecture-based software reliability models. Proceedings. 1n *the Proceedings on* 12^{th} *International Symposium on Software Reliability Engineering*, pp 22-31, 2001.

[103] K. Goseva-Popstojanova and K. Trivedi. Architecture-based approach to reliability assessment of software systems. *Journal of Performance Evaluation*, vol. 45, no. 2-3, pp. 179-204, 2001.

[104] L. Prechelt, B. Unger, W. F. Tichy, P. Brssler and L. G. Votta. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. IEEE Transactions on Software Engineering, vol. 27, pp. 1134-1144, 2001.

[105] R. S. Pressman. Software Engineering A Practitioner's Approach. McGraw Hill. 2005. ISBN 0-07-283495-1

[106] R. Roshandel, A. van der Hoek, M. Mikic-Rakic, and N. Medvidovic. Mae - A System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 2, pp. 240-276, 2002.

[107] R. Roshandel, S. Banerjee, L. Cheung, N. Medvidovic, and L. Golubchik. Estimating Software Component Reliability by Leveraging Architectural Models. In *the Proceeding on* 28^{th} *International Conference on Software Engineering*, pp. 853 - 856 2006.

[108] R. Roshandel. Calculating Architectural Reliability via Modeling and Analysis.In *the proceedings on* 26^{th} *International Conference on Software Engineering*, pp. 69-71, 2004.

[109] K. Sandkuhl and B. Messer. Towards Reference Architectures for Distributed Groupware Applications. In *the Proceeding on* 8^{th} *Euromicro Workshop on Parallel and Distributed Processing*, pp. 121-135, 2000.

[110] R. W. Schwanke. An intelligent tool for reengineering software modularity. In *the Proceedings of the 13th International Conference on Software Engineering*, pp. 83-92, 1991

[111] J.C. Dueas, W.L. de Oliveira and J.A. de la Puente. A Software Architecture Evaluation Method. In *the Proceedings on Second International ESPRIT ARES Workshop*, pp. 148-157, 1998.

[112] M. Sefika, A.Sane and R. H. Campbell. Monitoring compliance of a software system withits high level design models. In the *Proceedings of the* 18^{th} *International Conference on Software Engineering (ICSE)*, pp. 387-397, 1993.

[113] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *the Proceedings on the International Conference on Software Engineering*, pp. 387396, 1996.

[114] SEI: `http://www.sei.cmu.edu/`

[115] SEI-ATAM: `http://www.sei.cmu.edu/news-at-sei/features/2001/2q01/feature-4-2q01.htm`

[116] SEI-SAAM: `http://www.sei.cmu.edu/architecture/scenario_paper/ieee-sw2.htm`

[117] M. Shaw. Comparing architectural design styles. IEEE Software, 12(6), Nov. 1995, pp. 27-41.

[118] M. Shaw and P. Clements. A field guide to boxology: preliminary classification of architectural styles for software systems. In the Proceeding on 21^{st} COMPSAC, 1997.

[119] M. Shaw. Toward higher-level abstractions for software systems. Data and Knowledge Engineering, 5, 1990, pp. 119-128.

[120] M. Shooman. Structural models for software reliability prediction. In *the Proceedings of 2nd International Conference on Software Engineering*, pp. 268-280, 1976.

[121] C.U. Smith and L.G. Smith. *Performance Solutions: A Practical Guide to Creating Responsive*, Scalable Software. Addison-Wesley, Boston, 510 p., 2002.

[122] C.U. Smith and L.G. Williams. Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives. *IEEE Transaction on Software Engineering*, vol. 19, no. 7, pp. 720-741, 1993.

[123] C. U. Smith. Performance Engineering of Software Systems. Addison- Wesley, Massachusetts, 570 p., 1990.

[124] Software Reliability. `http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/#reference`

[125] The Shrimp. URL: `http://www.thechiselgroup.org/shrimp`.

[126] W.P. Stevens, G.J. Myers, and L.L. Constantine. *Structured Design, IBM Systems J.*, vol. 13, no. 2, pp. 115-139, 1974.

[127] C. Stoermer, F. Bachmann, C. Verhoef, SACAM: The Software Architecture Comparison Analysis Method, Technical Report, CMU/SEI-2003-TR-006, 2003.

[128] M. Svahnberg, C. Wohlin, L. Lundberg, and M. Mattsson. A Method for Understanding Quality Attributes in Software Architecture Structures. In *Proceedings of the* 14^{th} *International Conference on Software Engineering and Knowledge Engineering*, 2002.

[129] SWAG: Software Architecture Group. URL: `http://www.swag.uwaterloo.ca/SWAGKit/`.

[130] A. Tang, M. A. Babar, I. Gorton, and J. Han. A survey of architecture design rationale. *Journal of Systems and Software*, vol. 79, issue 12, pp. 1792-1804, 2006.

[131] R. N. Taylor and N. Medvidovic. A Component- and Message-Based Architectural Style for GUI Software. IEEE Trans. on Software Engineering, 22(6): 390-406, 1996.

[132] B. Tekinerdogan. ASAAM: aspectual software architecture analysis method. In the Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'04), June 2004, pp. 5-14.

[133] R.T. Tvedt, M. Lindvall, and P. Costa. A Process for Software Architecture Evaluation using Metrics. In *the proceedings of* 27^{th} *Annual NASA Goddard/IEEE*, pp. 191-196, 2002

[134] H. v. Vliet. *Software Engineering: Principles and Practice*, Wiley, 22 edition, ISBN-10: 0471975087, 748 p., 2000.

[135] J.M. Voas. *Software fault injection: inoculating programs against errors*. Wiley, New York, 1998.

[136] W. L. Wang, Y.Wu and M.H. Chen. An Architecture-Based Software Reliability Model. In *the Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, pp. 1-7, 1999.

[137] L.G. Williams and C.U. Smith. PASA: A method for the Performance Assessment of Software Architectures. In *the Proceedings of the Third International Workshop on Software and Performance (WOSP '02)*, pp. 179-189, 1990,

[138] L.G. Williams and C.U. Smith. Performance Engineering of Software Architectures. In *the Proceeding on Workshop Software and Performance*, pp. 164 - 177, 1998.

[139] A. Wood. Software Reliability Growth Models: Assumptions vs. Reality. In *the Proceedings on Eighth International Symposium on Software Reliability Engineering*, pp. 136-141, 1997.

[140] M. Woodside. Tutorial: *Introduction to Layered Modeling of Software Performance*, 2003, `URL:http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialf.pdf`.

[141] M. Xie and C. Wohlin.An Additive Reliability Model for the Analysis of Modular Software Failure Data. In *the Proceedings on IEEE* 6^{th} *International Symposium on Software Reliability Engineering*, pp. 188-194, 1995.

[142] S. M. Yacoub, and H. Ammar. A methodology for architectural-level reliability risk analysis. IEEE Transactions on Software Engineering 28: 529-547,2002

[143] S. Yacoub, B. Cukic, and H. Ammar. Scenario-based reliability analysis of component-based software. In *the Proceedings of 10th Int'l Symp. Software Reliability Engineering*, pp. 22-31, 1999.

[144] L. Zhu, M. Ali Babar, and R. Jeffery. Distilling Scenarios from Patterns for Software Architecture Evaluation. In *the Proceedings on* 1^{st} *European Workshop on Software Architecture*, pp. 219-224, 2004.

[145] L. Zhu, M. Ali Babar, and R. Jeffery. Distilling Scenarios from Patterns for Software Architecture Evaluation. In *the Proceedings on* 1^{st} *European Workshop on Software Architecture*, pp. 219-224, 2004.