Towards a Generic Framework for Evaluating Component-Based Software Architectures

Steffen Becker[1], Viktoria Firus[1], Simon Giesecke[1],
Willi Hasselbring[1], Sven Overhage[2], and Ralf H. Reussner[1]

[1]Software Engineering Group, University of Oldenburg
OFFIS, Escherweg 2, 26121 Oldenburg, Germany
{becker, firus, giesecke, hasselbring, reussner}@informatik.uni-oldenburg.de

[2]Dept. of Systems Engineering and Business Information Systems,
Augsburg University, Universitätsstraße 16, 86135 Augsburg, Germany
sven.overhage@wiwi.uni-augsburg.de

Abstract: The evaluation of software architectures is crucial to ensure that the design of software systems meets the requirements. We present a generic methodical framework that enables the evaluation of component-based software architectures. It allows to determine system characteristics on the basis of the characteristics of its constituent components. Basic prerequisites are discussed and an overview of different architectural views is given, which can be utilised for the evaluation process. On this basis, we outline the general process of evaluating software architectures and provide a taxonomy of existing evaluation methods. To illustrate the evaluation of software architectures in practice, we present some of the methods in detail.

1 Introduction

An important goal of the software engineering discipline, which has been pursued since the first software engineering conferences in the 1960s [NR69], is to support the development of software systems that *reliably* (i.e. by construction) conform to the functional and non-functional requirements. The introduction of *software architecture* as a special software engineering concept is a major step towards ensuring the consistency between requirements and detailed software design. This allows the evaluation of software designs with respect to their functional and non-functional characteristics [SG96].

Nowadays, by using components [SGM02] as the building blocks of software systems, it is conceivable to improve the evaluation of software architectures. Then, characteristics of a system (i.e. an assembly) may be algorithmically *deduced* from the characteristics of its constituent parts [WSHK01]. Note that — conforming to the ISO 9126-1 standard [ISO03a] — the term *property* is used for more abstract concepts like performance, while the term *characteristic* is used to denote a metric (which is directly assessable) for a specific property. Thus, the properties of component-based software systems can be predicted precisely (instead of using some heuristic), provided the characteristics of its components are sufficiently known. Moreover, system characteristics can be deduced based solely on the architecture and thus *before* the constituent components have been implemented or acquired.

The contributions of this paper are a generic methodical framework for the evaluation of component-based software architectures, and a discussion on how system characteristics can be deduced from the properties of its constituent components. The framework consists of the definition of a set of architectural views, the description of the general process of evaluating component-based architectures, and a taxonomy of evaluation methods.

The *value for industrial practice* of such a framework is to support the selection of an appropriate evaluation method. A software architect, who has to decide which evaluation methods to use for the next projects, can find comprehensive literature on various methods for architecture evaluation. However, while these methods provide scientific advancements for specific evaluation tasks, most of them have not been integrated in an approach applicable in industrial practice. Although this shows scientific activity in this area, it especially hinders the selection and hence the application of evaluation methods in industrial projects.

To provide an overview of applicable methods, we classify a variety of approaches. Several classification dimensions are proposed to better support the selection process. In particular, the introduced taxonomy is based on the set of architectural views that each evaluation method uses. Once we have refined the taxonomy to reflect necessary inputs of the methods in more detail, such a classification will be particularly helpful for practitioners. The taxonomy allows them either to decide which evaluation approach fits to the way of architectural specification they are accustomed to, or to adapt their way of specifying software architectures according to the evaluation technique selected. As a first step to support the latter, this paper briefly describes some evaluation approaches and references relevant literature.

This paper contributes to the scientific advancement in the area of component-based architecture evaluation by enabling the comparison of evaluation methods. The ability to compare various evaluation methods is a precondition to a deeper understanding of the problem area, to cross-validating different approaches and to constructing improved evaluation methods based on these findings. The taxonomy presented in this paper focuses on the information needs of evaluation methods as a first step. Classifying methods by the characteristics they evaluate in more detail and pointing out how their predictions differ is subject to future work.

This paper is organised as follows: first, the deduction of system characteristics from those of its constituent parts is discussed as a conceptual basis in the following section. We argue that component-based software architectures are superior to others in satisfying this prerequisite. In section 3, several architectural views that can be used to evaluate system characteristics are introduced. The views are used as the basis of the generic framework for evaluating component-based software architecture, which is elaborated in section 4. Some examples of evaluation methods for system characteristics are elaborated on in section 5. A discussion of related work and an outlook on an integrated method that supports the predictable assembly of components concludes this paper.

2 Prerequisites

First of all, the success of a method that computes a characteristic of a system from the characteristics of its constituent parts depends on the availability of a “divide-and-conquer” strategy to reduce the evaluation complexity [Wal03]. Similar to verification approaches that support *compositional or modular reasoning* [dR98, Wal03], such a strategy allows the architect to shift the effort required for the evaluation from the complex system level to the parts level. System characteristics are then computed by composing independently specified (and certified) characteristics of its constituent parts.

However, the formal prerequisites of applying compositional or modular reasoning techniques — the independence of characteristics among the different parts (their so-called *compositionality*) being the most important — cannot be satisfied when analysing a software architecture in most cases[Wal03]. Component technology ensures compositionality with respect to the *construction* of software systems. It provides some means to connect components in such a way that they can interact with each other [SGM02]. This also ensures the compositionality of the characteristics of architectural parts, which is *required* for the analysis of software architectures.

Component-based software architectures are superior to the class-based architectures (i.e., based on object-oriented technology) with respect to the support for compositional and modular reasoning techniques, because components are *encapsulated* units of software. Class-based architectures, on the contrary, make use of connecting classes to each other by *implementation inheritance*, which actually breaks the principle of encapsulation [Sny87] and creates complex interdependencies that usually affect functional as well as non-functional characteristics. For this reason, classes cannot be regarded as independent, compositional pieces of software and are difficult to analyse as architectural units.

In addition, components are coarse-grained units of software with only a few, clearly defined ports of interaction that can be connected using connectors (see figure 1) [SG96]. Due to the coarse granularity of components, the architectural structure resulting from component composition is simpler than those of class-based software architectures and, thus, easier to analyse.

Finally yet importantly, we consider as components only such entities which provide more detailed interface specifications that not only include provided and required interfaces, but also contain annotations regarding the supported interaction protocol as well as non-functional characteristics [Bro00, Ove04]. Especially this additional information, which is unavailable for classes and objects, is an important basis to analyse the characteristics of software architectures (cf. section 4).

However, even when component-based software architectures are used, system characteristics cannot always be deduced from component characteristics using formal modular reasoning techniques. Such a deduction is impossible [Wal03] because the characteristics of the individual components often influence each other and are thus interdependent.

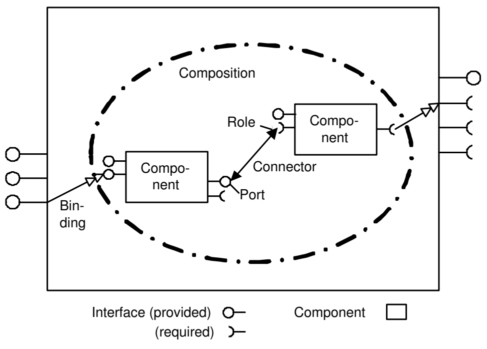

Figure 1: Conceptual Port-Connector Model to describe software architectures (based on [SG96]).

Consequently, formalising a compositional theory to compute system characteristics from those of the individual components may still not be possible.

Ignoring these interdependencies between component characteristics leads to simplified theories for analysis that may not be completely accurate, but produce useful results in many cases. To stress the fact that often simplified theories are used to deduce system characteristics, the term "evaluation" is used instead of "analysis" throughout this paper.

3 Architectural Views

Specifying software architectures is a complex task and usually achieved by describing different aspects of the system design. Existing approaches differ with respect to the aspects that are described as well as the notations that are used. Nevertheless, in general four different types of architectural views can be identified [SG96, CBB$^+$03]:

Structural Views These explicate the constituent parts of an architecture (i.e. the individual components) as well as the connections between components. Structural views are used to illustrate the logical structure of a software system and its decomposition. In the Unified Modelling Language (UML 2.0) [OMG03], the logical structure can be represented using Component Diagrams or Class Diagrams (when not using inheritance).

Resource Mapping Views The components described in a structural view have to be mapped (a) to hardware units (such as processors, network connections, or specific devices, such as printers, etc.) and (b) to organisational entities, such as developers, teams or third-party vendors. In the UML 2.0, resource mappings of the former kind can be represented using Deployment Diagrams [OMG03].

Dynamic Views They describe the dynamic behaviour of a system at run-time. The term "dynamic" does not necessarily refer to dynamic changes of the structural view.

The dynamic views are concerned with the flow of control through the system's structure. This may or may not result in dynamic changes of the structure, e.g., if components are created or destroyed dynamically. In the UML 2.0, dynamic views can be represented using Sequence Diagrams, State Machine Diagrams, or Activity Diagrams [OMG03].

Domain Views Domain views are concerned with information referenced in all of the above-mentioned views. For example, a glossary with definitions or clarified concepts of the application domain may be part of the domain view. Additionally, the domain views may contain information more specific to the implementation domain [JRvdL00] like domain-specific patterns or architectural styles or, most importantly for reliability analysis, information about exceptions and error types (e.g., transient or permanent errors).

Regarding the domain views, one may argue that such information should be established during requirements analysis and thus is not part of an architecture. Although the information is gathered during requirements analysis, it is essential for implementing and evolving the architecture. This information should therefore be made explicit and should be associated with the architecture to ensure its consistency with the gathered requirements [DW99].

This set of view types emphasises the fact that an architecture is not just defined by structural information, but consists of several views. This is different from other approaches that use connectors (as first-class entities) to include behavioural dynamic information in structural diagrams [SG96, MT00].

4 Evaluating Software Architectures

The usefulness of the specification of a software architecture can be considered from different points of view. The architecture is used to support the communication between the stake-holders of a project, to distribute the development work between the organisational units, or to evaluate certain properties of the resulting system. In this paper, we focus on the latter, which is an active field of current research. Several methods use representations of software architectures as input data to evaluate characteristics of the resulting system. Some examples of evaluation methods used today are presented in more detail in the section 5.

This section is further divided into three subsections. Firstly, general considerations with respect to the input data needed for an evaluation are presented. Secondly, the process of applying evaluation methods is discussed, with a special focus on the importance of a feedback mechanism. Finally, a rough overview on the input data needed by some evaluation methods is presented.

4.1 Views for architectural evaluation

Before starting an evaluation of a software architecture, it is important to check whether the necessary specifications are available and if the evaluation is worth the effort. When developing a small application, for instance, a cost-benefit analysis can easily show a negative result. Therefore, the overall information needs and basic considerations are presented in the following. A summary of the important elements of the general evaluation process is depicted in figure 2.

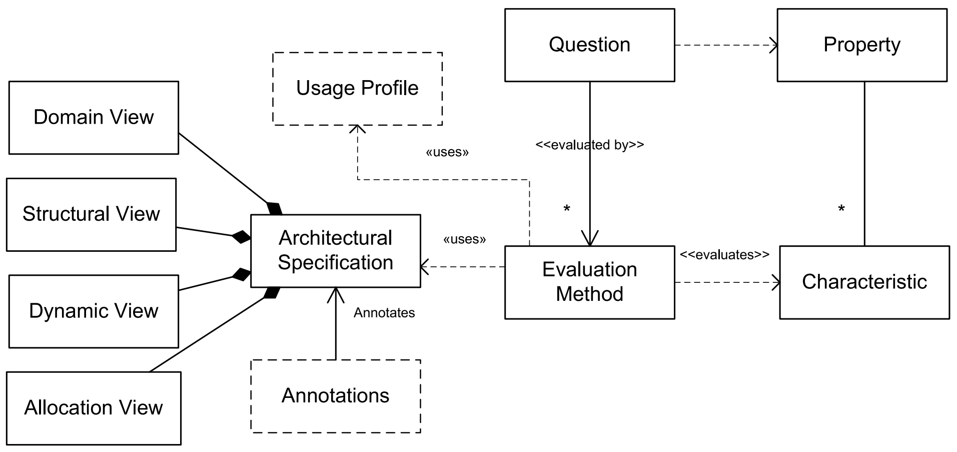

Figure 2: Generic evaluation method (static view)

Evaluating a software system with respect to all of its properties is inconceivable. Before performing an evaluation, the set of properties of interest must be determined, e.g. one might be interested in Quality of Service properties or the development costs of the system. In other words, there has to be an *evaluation goal* first.

Once an evaluation goal has been defined, there usually is a variety of different *evaluation methods* available to perform the actual evaluation. Typically, several characteristics correspond to a given property. Since every characteristic is considered to be associated with at least one evaluation method, several evaluation methods exist to predict each property.

Most evaluation methods use some kind of formal model internally, each of which has certain advantages and disadvantages in a specific context. In an ideal setting each of the methods is supported by tools hiding the internal formalism. Then, the user of the specific method needs no additional knowledge of the application of the underlying formal methods.

One of the deficiencies of existing evaluation methods is that each of them considers only one specific characteristic, and thus only one property. But often there are interdependencies among the properties. For example, a high performance system might not be as reliable as a system which performs worse. On the other hand, one might be more costly than the other. The characteristics of the interdependency often depend on the architecture of the system to be built but also on its usage profile [BR04].

When given a goal and a method, it is possible to determine the extent of *input data* the method requires. As the focus of this paper is on the evaluation of software architectures, an evaluation method is always applied to a specific architecture. Most methods expect the architecture to be annotated with certain method-specific attributes. While software architectures are specified according to the views presented in section 3, the required annotations usually can be obtained from the specifications of the single components [Ove04].

By definition, the architectural specification is created at design time. As some of the properties an analyst might be interested in are runtime properties, e.g. performance, additional input that is not contained in the architectural specification is required, e.g. in the form of a *usage profile*. The influence of the usage profile on the evaluation becomes obvious when considering Quality of Service properties like performance or reliability. For instance, processing the tenfold amount of data requires more time than processing the single amount; a system failure is also more probable in the former case . However, despite the obvious dependency of some properties on the usage profile, there are few evaluation methods incorporating the usage profile in their computation comprehensively, e.g. parameter values are often neglected.

It is important to consider the consistency of the specification, because the quality of an evaluation is restricted by the quality of its input data. There are two different kinds of inconsistency problems: inconsistencies in a single view, and inconsistencies between views.

Inconsistencies *in a single view* often result from specification mistakes, e.g. connecting two components which are incompatible with each other, draft a sequence diagram which is inconsistent with the protocol of at least one component, breaking the preconditions of a component or deploying a component on an incompatible hardware or software platform. Note that these inconsistencies are specifically addressed by certain evaluation methods, e.g., a method of analysing component interoperability problems. Nevertheless, other evaluation methods assume that the specified views of an architecture are consistent. Consider, for example, a cost estimation method. It is very unlikely that such a method will gain a reasonable result if the analysed architecture cannot be consistently implemented.

Inconsistencies *between views* are even worse for most of the evaluation techniques. Inconsistencies between views are caused by mistakes during the specification process, but also by changes to the architecture. This phenomenon is well known to software engineering practitioners from the context of UML specifications of software systems. In order to keep all UML views consistent — especially during the development phase — updating all dependent views requires more time than it takes to perform the primary change. As a result, the consistent update of the UML views is often postponed in practice or the necessary updates to the models are performed only in a single view leaving the other views in an inconsistent state. Specific evaluation methods can be used to discover certain kinds of inconsistencies, but more often, an evaluation method assumes that the views are pairwise consistent with each other.

4.2 Evaluation Process

In figure 3, the process of performing an architectural evaluation is depicted as a UML 1.x activity diagram.

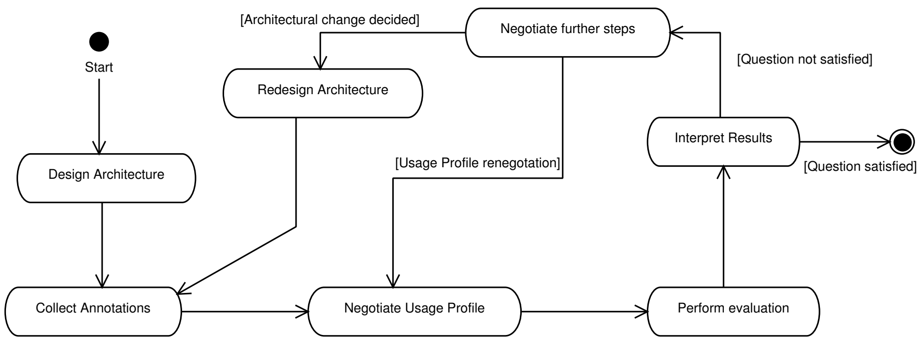

Figure 3: Generic evaluation method (dynamic view)

Before applying an evaluation method, the necessary input data has to be gathered. Again, for the application of a method in an industrial setting it is very important to have tool support during that stage. When all necessary specifications (architectural views and annotations) are available, a negotiation phase takes place, as part of which the architect and the customer (or the users of the software) agree on the usage profile the architecture should be able to support. Once this phase is completed, the actual evaluation is performed.

After the evaluation, mapping the results of the specific evaluation is often necessary to give hints or answers to the original question which the evaluation was supposed to solve. Such a mapping is necessary because the model used by the formal evaluation methods differs from the original architecture. Therefore, it is necessary to *interpret* the results of the methods in the context of the original input data to get useful conclusions.

Based on the interpreted figures a decision is made on whether the results satisfactorily resolve the question or not. If not, there is a choice: either one can change the architecture or re-negotiate the usage profile with the customer.

The whole process is repeated as long as the evaluation question is not satisfied. When the architecture satisfies the requirements of the customer, the process ends.

4.3 A taxonomy of evaluation methods

The structure of the table depicted in figure 4 stresses the inter-relationships of the evaluation properties, the evaluation methods and their input data. Based on the discussion in this section, an overview of well-known evaluation methods is given. They are classified according to the required input data which is represented by the views discussed in section 3.

Property	Method	Reference	Domain	Static	Dynamic	Resource Mapping
Performance	CB-SPE	[BM04]		✔	✔	✔
	UML-PSI	[BGM03]		✔	✔	✔
	LQN	[PW00]	✔	✔	✔	
	Cap. Planning	[MAA01]	✔	✔	✔	
Reliability	Cheung	[Che80]		✔	✔	
	Wang et al.	[WWC99]	✔	✔	✔	
	Ledoux	[Led99]	✔	✔	✔	
	Krishnamurthy et al.	[KM97]		✔	✔	
	Xie et al.	[XW95]			✔	
Interoperability	Zaremski	[ZW97]		✔		
	Yellin	[YS97]		✔	✔	
Cost	COCOTS	[ABC00]		✔		
	COCOMO	[BCH$^+$95]		✔		
	ACSPM	[Pau01]		✔		

Figure 4: Overview of evaluation methods and their input data

Although the overview is not complete, it can be seen as a guide for practitioners and researchers to understand the information needs of various evaluation methods, which affect the costs to obtain the required data. Each view contributes at least some information which is as such not provided by any other view but may heavily influence a property. Thus, neglecting one view necessarily restricts the precision of the evaluation.

5 Architectural Evaluation Methods

In this section some of the more common properties are elaborated by means of an example evaluation method. The specification needs and the evaluation result are characterised respectively.

5.1 Interoperability Checking

Interoperability checking is an important problem at assembly time of a component based software architecture. While it is one aim of Component-based Software Engineering to reuse as many existing components as possible, reusing components as they are is often difficult. This is sometimes considered a non-issue when components are always built to fit in a certain context. But in practice, software architects often have to deal with legacy systems which have to be integrated in a software architecture [HRJ$^+$04]. The business value of these systems disallows rebuilding them from scratch. When analysing a software architecture the architect has to ensure that the components chosen to realise the desired domain functionality are cooperating in the desired way.

Interoperability problems can emerge on different levels of the system's specification (see

[BOR04]). In general, the more information is available, the more problems are detectable already during architectural design.

Static views on the architecture can be used to detect problems related to signature or syntax. A method which can be applied to these is proposed by Zaremski and Wing [ZW97]. It tries to match services and well known requirements, which result from required interfaces of components. Often simple interoperability problems can be overcome by conversion routines using simple transformations. When contra-variant use of parameters or signatures should be allowed, an evaluation method has to deal with sub-typing during matching.

When using the information available in the dynamic view, protocol interoperability problems are detectable and sometimes resolvable. The work of Yellin and Strom [YS97] is often used in this context. Problems in protocol adaptation are often related to the way component protocols are specified. A formalism often used in the context of architectural evaluation is the model of finite state machines (FSMs) because of its algorithmic properties. On the other hand, FSMs have limited expressiveness, so that several extensions and other models are used, including Petri Nets, linear timed logics (LTLs) or process algebras (PAs). The latter are also applied during interoperability checking, for example in [BBC02].

If domain related architectural views are available, it is also possible to detect domain related interoperability problems which arise often by a differing use of domain terminology and concepts. Especially the usage of components developed by different vendors is precarious if all components conform to common domain standards or standardised component frameworks. Often a manual check of the terms used and their definitions is required to decide on domain-related interoperability.

Taking into account the sum of all architectural views, it is also possible to check the interoperability regarding non-functional properties on the levels discussed above. As this topic is subject to current research, a detailed look at methods for such evaluations is beyond the scope of this subsection.

5.2 Performance Evaluation

Performance is a property for which quite a large variety of architectural evaluation methods are available. Performance is always an important issue during the design phase of a software development project. Nowadays the fix-it-later approach is often used, i.e., performance is neglected during the early development phases and performance is evaluated later. If the final performance is insufficient, critical parts of the system are identified using a profiling tool and an attempt is undertaken to speed them up.

Since costs of changes to a software system increase with the progress in development, it is desirable to use architectural specifications to shift the performance evaluation to the early stages of the development process. The CB-SPE approach [BM04] aims to adapt the established SPE method [Smi90] for component based systems. The method is presented here in more detail as an example.

The *input data* needed for the evaluation process is taken from UML diagrams. These diagrams are annotated with specifications from the UML Profile for Schedulability, Performance, and Time [OMG02] which is also used by several other methods for performance evaluation. This method uses several types of UML diagrams [OMG03]: Class Diagrams model the static view, the dynamics are extracted from Sequence Diagrams, and Deployment Diagrams model the mapping of software components to hardware resources. Additionally, the method takes the usage profile into account by weighting use cases taken from UML Use Case Diagrams.

The method is exemplary of evaluation methods of quality properties of software architectures. The above-mentioned input specification is transferred as a XMI data file to a custom transformation tool which transforms the input data into the required formal model. In the case of CB-SPE, the specification is used to build a queueing network model which is solved by a standard queueing network model solver, showing among others response times and average queue loads.

These results are then interpreted and re-transformed into meaningful statements about the original software architecture. For this, CB-SPE adds tagged values to the XMI data exchange file. This file can be reloaded into a UML modelling tool to visualise the results. The results can then be evaluated to identify performance bottle necks.

5.3 Reliability Estimation

Architecture-based approaches to reliability estimation consider white-box models of software, following a long tradition of reliability estimation techniques that considered software systems as monolithic blocks. The classification of Goševa-Popstojanova and Trivedi [GPT03] distinguishes state-based, path-based and additive reliability estimation methods. State-based methods use various kinds of Markov chains and model components as states. Note, however, that the concept of a component is used in a very general sense in the context of reliability estimation. In particular, it is often used synonymously to that of a module. Path-based models are based on execution traces which have components as their elements. Additive models assume knowledge of the failure behaviour of individual components only.

The state-based model by Cheung [Che80] is one of the most thoroughly applied, since the model is used in the Cleanroom software engineering method [PMM93]. It requires knowledge of the probabilities of transitions between components (and thus at least implicit knowledge of the structural dependencies between components) and the individual components' reliabilities. The model is extended by Wang et al. [WWC99] and adapted to specific architectural styles, thus incorporating domain knowledge.

A model that applies to reactive systems is described by Ledoux [Led99]. Based on [Lit75], it requires knowledge of both component and interface failure rates. The model distinguishes primary and secondary failures specific to continuously running, reactive systems.

The path-based model by Krishnamurthy et al. [KM97] requires as input the reliabilities of individual components and a set of component traces. The traces are acquired experimentally by executing test cases in [KM97], but they could in principle be taken from a specification of allowed traces.

The additive model by Xie et al. [XW95] describing individual component reliability and overall system reliability by modelling their failure intensities as non-homogeneous Poisson processes. The component failure intensities have to be known. Further input is not used, in particular structural aspects of the analysed system are neglected.

5.4 Cost Estimation

In the following, some examples of cost estimation methods based on architectural information are presented. A well known method is the COCOMO or COCOMO II model respectively [BCH$^+$95]. Based on this model, the COCOTS method was introduced as an advancement. COCOTS is a method to evaluate the integration costs of commercial of the shelf components. Costs for the search or development of single components are disregarded. COCOTS includes the costs for assessment, tailoring or adaptation, development of glue code and system volatility costs (this type of costs capture how often a component is updated, e.g., by bug-fix releases).

Architecture-centric software project management (ACSPM) by Paulish [Pau01] includes an architecture based approach for software cost estimation. This approach uses bottom-up estimates: for each component the responsible developer is asked to predict the costs of developing this component. The costs of all components are summed up and an additional amount for costs of software integration is added. The approach therefore uses the structural view of an architecture.

Although no additional support for the prediction of component costs or the estimation of integration costs is given, one could apply other estimation techniques for that. Examples are object point analysis [BKK92] or various size estimation techniques, and historical cost data, such as proposed in [Hum94, pp. 97]. The results of bottom-up estimation are used to create the project schedule which itself influences the software development plan. This plan is used to create the schedules for each developer.

6 Related Work

As outlined in this paper, developing methods to evaluate component-based software architectures is still a field of intensive research. Current research projects often focus on developing specialised methods to evaluate a specific characteristic of software architectures, which, for example, help to predict the performance of software systems [BM04, BIM98, BGM03, PW00, MAA01, BMDI04, BMIS04], the reliability of software systems [SM02], the interoperability of individual components [ZW95, ZW97, YS97], or the development costs of software systems [ABC00, BCH$^+$95, Pau01]. A general introduction to evaluating these and other system properties based on software architectures can be found in [Bos00].

The evaluation methods primarily focus on evaluating a single characteristic of a software system. In the long run, however, it is desirable to have a mix of methods which supports the evaluation of multiple system characteristics simultaneously. While this currently remains a vision, a variety of integrated methods for evaluating software architectures is already emerging. Notable among these are SAAM (Software Architecture Analysis Method) and ATAM (Architecture Tradeoff Analysis Method) [CKK01] as well as PACC (Predictable Assembly from Certifiable Components) [Wal03, HMSW02].

The set of architectural views in section 3 is not fully consistent with the three view types defined in [CBB^{+}03]. The module view type *and* the component-connector view type are concerned with structural views, while the allocation view type matches with the resource mapping views. Besides that, it differs from the four views described by Hofmeister et al. [HNS99]. Hofmeister's conceptual and module views can be considered as structural views. Hofmeister's code and execution views partially model information that belongs to the dynamic view (or are at least strongly influenced by that information). Finally, Hofmeister's execution view includes information of the resource mapping view.

7 Conclusions

In this paper, we present a generic methodical framework which illustrates the prerequisites and the overall process of evaluating software architectures in order to deduce functional and non-functional characteristics of software systems. It provides a taxonomy of methods to deduce specific characteristics and explains some of them in detail.

The paper illustrates that a variety of architecture evaluation methods exists, which can be used to deduce specific system characteristics. The above-mentioned methods primarily cover reliability and performance of systems, which can be measured using well-defined metrics [ISO03b]. Research into evaluation of other quality properties like maintainability [BB01], security, or privacy is still only emerging.

As a result, many system properties of interest nowadays cannot be predicted using architecture evaluation methods. The outlined framework thus remains incomplete. However, even by only applying the above-mentioned methods, significant improvements in software development (and architecture management) could be achieved. To this end, the outlined framework may serve as a guideline to introduce architecture evaluation into the process of software development. Additionally, we structure the field of architecture evaluation and thereby also highlight unresolved issues that motivate further research.

References

[ABC00] Chris Abts, Barry W. Boehm, and Elizabeth Bailey Clark. COCOTS: a COTS software integration cost model – model overview and preliminary data findings. In *Proc. European Software Control and Metrics Conf. (Escom 2000)*, pages 325–333, 2000.

[BB01] Jan Bosch and PerOlof Bengtsson. Assessing Optimal Software Architecture Maintainability. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 168. IEEE Computer Society, 2001.

[BBC02] Andrea Bracciali, Antonio Brogi, and Carlos Canal. Dynamically Adapting the Behaviour of Software Components. In Farhad Arbab and Carolyn L. Talcott, editors, *Coordination Models and Languages, 5th International Conference, COORDINATION 2002, York, UK, April 8–11, 2002, Proceedings*, volume 2315 of *Lecture Notes in Computer Science*, pages 88–95. Springer-Verlag, Berlin, Germany, 2002.

[BCH$^+$95] Barry W. Boehm, Bradford Clark, Ellis Horowitz, J. Christopher Westland, Raymond J. Madachy, and Richard W. Selby. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering*, 1:57–94, 1995.

[BGM03] Simonetta Balsamo, Mattia Grosso, and Moreno Marzolla. Towards Simulation-Based Performance Modeling of UML Specifications. Technical Report CS-2003-2, Dep. di Informatica, Universtita Ca' Foscari Venezia, Italy, 2003.

[BIM98] Simonetta Balsamo, Paola Inverardi, and Calogero Mangano. An approach to performance evaluation of software architectures. In *Proc. first international workshop on Software and performance (WASP'98)*, pages 178–190. ACM Press, 1998.

[BKK92] Rajiv D. Banker, Robert J. Kauffman, and Rachna Kumar. An Empirical Test of Object-Based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment. *Journal of Management Information Systems*, 8(3):127–150, 1992.

[BM04] Antonia Bertolino and Raffaela Mirandola. CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Proc. 7th International Symposium on Component-Based Software Engineering (CBSE 2004), Edinburgh, UK*, volume 3054 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2004.

[BMDI04] Simonetta Balsamo, Moreno Marzolla, Antinisca Di Marco, and Paola Inverardi. Experimenting different software architectures performance techniques: a case study. In *Proceedings of the fourth international workshop on Software and performance*, pages 115–119. ACM Press, 2004.

[BMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[BOR04] Steffen Becker, Sven Overhage, and Ralf Reussner. Classifying Software Component Interoperability Errors to Support Component Adaption. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *Component-Based Software Engineering, 7th International Symposium, CBSE 2004, Edinburgh, UK, May 24-25, 2004, Proceedings.*, volume 3054 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2004.

[Bos00] Jan Bosch. *Design and Use of Software Architectures – Adopting and evolving a product-line approach*. Addison-Wesley, Reading, MA, USA, 2000.

[BR04] Steffen Becker and Ralf H. Reussner. The Impact of Software Component Adaptors on Quality of Service Properties. In Carlos Canal, Juan Manuel Murillo, and Pascal Poizat, editors, *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT 04)*, June 2004.

[Bro00] A. W. Brown. *Large-Scale, Component-Based Development*. Prentice Hall, Upper Saddle River, NJ, 2000.

[CBB$^+$03] Paul C. Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures*. SEI Series in Software Engineering. Addison-Wesley, 2003.

[Che80] Roger C. Cheung. A User-Oriented Software Reliability Model. *IEEE Transactions on Software Engineering*, 6(2):118–125, March 1980. Special collection from COMPSAC '78.

[CKK01] Paul C. Clements, Rick Kazman, and Mark Klein. *Evaluating Software Architectures*. SEI Series in Software Engineering. Addison-Wesley, 2001.

[dR98] W. P. de Roever. The Need for Compositional Proof Systems: A Survey. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 1998.

[DW99] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML. The Catalysis Approach.* Addison-Wesley, Upper Saddle River, NJ, 1999.

[GPT03] K. Goševa-Popstojanova and K. S. Trivedi. Architecture-based approaches to software reliability prediction. *Computers & Mathematics with Applications*, 46(7):1023–1036, October 2003. Special Issue on Applied Stochastic System Modelling.

[HMSW02] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging Predictable Assembly. In Judy M. Bishop, editor, *Component Deployment, IFIP/ACM Working Conference, CD 2002, Berlin, Germany, June 20-21, 2002, Proceedings*, volume 2370 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 2002.

[HNS99] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied Software Architecture*. Addison-Wesley, Reading, MA, USA, 1999.

[HRJ$^+$04] W. Hasselbring, R. H. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, and S. Krieghoff. The Dublo Architecture Pattern for Smooth Migration of Business Information Systems. In *Proceedings of the 26rd International Conference on Software Engeneering (ICSE-04)*, Los Alamitos, California, May23–28 2004. IEEE Computer Society.

[Hum94] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Reading, MA, USA, 1994.

[ISO03a] ISO/IEC Standard. Software Engineering – Product Quality – Part 1: Quality Model. ISO Standard 9126-1, ISO/IEC, 2003.

[ISO03b] ISO/IEC Standard. Software Engineering – Product Quality – Part 2: External Metrics. ISO Standard 9126-2, ISO/IEC, 2003.

[JRvdL00] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, Reading, MA, USA, 2000.

[KM97] Saileshwar Krishnamurthy and Aditya P. Mathur. On the Estimation of Reliability of a Software System Using Reliabilities of its Components. In *Proc. 8th Intl. Symp. Software Reliability Eng. (ISSRE'97)*, pages 146–, September 1997.

[Led99] J. Ledoux. Availability modeling of modular software. *IEEE Transactions on Reliability*, 48(2):159–168, 1999.

[Lit75] B. Littlewood. A reliability model for systems with Markov structure. *Applied Statistics*, 24(2):172–177, 1975.

[MAA01] Daniel A. Menascé, Virgilio A. F. Almeida, and Virgilio Almeida. *Capacity Planning for Web Services: Metrics, Models, and Methods*. Prentice Hall, Englewood Cliffs, NJ, USA, 2. edition, 2001.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[NR69] P. Naur and B. Randell, editors. *Software Engineering*, Brussels, 1969. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[OMG02] OMG. UML Profile for Schedulability, Performance, and Time. OMG Specification ptc/2002-03-02, Object Management Group, 2002.

[OMG03] OMG. UML 2.0 Superstructure Specification. Final Adopted Specification ptc/03-08-02, Object Management Group, 2003.

[Ove04] Sven Overhage. UnSCom: A Standardized Framework for the Specification of Software Components. In Mathias Weske and Peter Liggesmeyer, editors, *Object-Oriented and Internet-Based Technologies, 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, NODe 2004, Proceedings*, volume 3263 of *Lecture Notes in Computer Science*, pages 169–184, Berlin, Heidelberg, 2004. Springer.

[Pau01] Daniel J. Paulish. *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesley, Reading, MA, USA, 2001.

[PMM93] J. H. Poore, Harlan D. Mills, and David Mutchler. Planning and Certifying Software System Reliability. *IEEE Software*, 10(1):88–99, 1993.

[PW00] Dorina C. Petriu and Xin Wang. Deriving Software Performance Models from Architectural Patterns by Graph Transformations. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, November 16-20, 1998, Selected Papers*, volume 1764 of *Lecture Notes in Computer Science*, pages 475–488. Springer, 2000.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley, Harlow, 2. edition, 2002.

[SM02] Judith A. Stafford and John D. McGregor. Issues in Predicting the Reliability of Composed Components. In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, editors, *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, Los Alamitos, California, May 2002. IEEE Computer Society.

[Smi90] Connie U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, Reading, MA, USA, 1990.

[Sny87] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In N. Meyrowitz, editor, *Proceedings of the 1st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'86)*, volume 21 (11) of *ACM SIGPLAN Notices*, pages 38–45. ACM, 1987.

[Wal03] Kurt C. Wallnau. A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, 2003.

[WSHK01] Kurt C. Wallnau, Judith Stafford, Scott A. Hissam, and Mark Klein. On the Relationship of Software Architecture to Software Component Technology. In Jan Bosch, Clemens Szyperski, and Wolfgang Weck, editors, *Proceedings of the 6th International Workshop on Component-Oriented Programming (WCOP 01)*, 2001.

[WWC99] Wen-Li Wang, Ye Wu, and Mei-Hwa Chen. An Architecture-Based Software Reliability Model. In *Proc. 1999 Pacific Rim International Symposium on Dependable Computing*, pages 143–150. IEEE Computer Society, 1999.

[XW95] M. Xie and C. Wohlin. An additive reliability model for the analysis of modular software failure data. In *Proc. 6th Intl. Symp. Software Reliability Engineering (ISSRE'95)*, pages 188–194, Toulouse, France, October 1995. Institute of Electrical and Electronics Engineers.

[YS97] D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.

[ZW95] A. M. Zaremski and J. M. Wing. Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

[ZW97] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.