Evaluating Software Architectures: Development, Stability, and Evolution

Rami Bahsoon and Wolfgang Emmerich

Dept. of Computer Science,
University College London (UCL)
Gower Street, London WC1E 6BT, UK
{r.bahsoon; w.emmerich}@cs.ucl.ac.uk

Abstract

We survey seminal work on software architecture evaluation methods. We then look at an emerging class of methods that explicates evaluating software architectures for stability and evolution. We define architectural stability and formulate the problem of evaluating software architectures for stability and evolution. We draw the attention on the use of Architectures Description Languages (ADLs) for supporting the evaluation of software architectures in general and for architectural stability in specific.

Keywords

Architectural evaluation; Architectural stability; Economic-Driven Software Engineering Research.

1 INTRODUCTION

The architecture of the system is the first design artifact that addresses the *quality goals* such as security, reliability, usability, modifiability, stability, and real-time performance. As the manifestation of early design decisions, the architecture represents those design decisions that are hardest to change [42] and need to be validated against the quality goals for mitigating risks. A software *architecture review* is an activity to develop an evaluation of an architecture against the quality goals. *Architecture evaluation* aims to assess and validate the software architecture using systematic methods or procedures. The evaluation is done with the objective to ensure that the architecture under question satisfies one or more quality goals, the so called the *review objectives*. Evaluation also aims to ensure that the architecture is *buildable*. That is, the system can be built using the resources at hand: the staff, the budget, the legacy software (if any), and the time allotted before delivery. From an evolutionary perspective, reviews are *preventive* activities to delay the decay (as referred by Parnas) and limit the effect of software aging [43]. Architectural evaluations represent a wise risk-mitigation effort and are relatively inexpensive [12].

Effort on architectural evaluation goes back to the seminal work of Parnas and Weiss [44]. Their paper entitled "Active Design Reviews: Principles and Practices" is regarded as the cornerstone to the architectural review/evaluation area. In their paper, Parnas and Weiss expressed one of the fundamental principles behind the architectural evaluation methods: undirected and unstructured design reviews for software design do not work. Their work was motivated by the observations that approaches to design review tend to be spotty, ad hoc, and not repeatable. The common practice was –and still is- to identify a group of reviewer, drop a stack of read-ahead material on their desk a week or so prior the meeting, haul them in a conference room for a few tedious hours, ask for comments on the material read, and hope for the best [13]. The outcome of such practice is predictable and entirely disappointing: failing to uncover any serious problems with the design under consideration and propagating the problem to other phases. Obviously, this is attributed to human nature: participants will not have cracked the read-ahead material until the last minute (if not at all), or perhaps they have read to make some intelligent comments. In short, the outcome is an unexercised design artifact.

Parnas and Weiss prescribed a better way. Active Design Reviews (ADRs) [44] were suggested and contrasted with unstructured reviews in which people are asked to read a document, attend a long meeting, and comment on whatever they wish [13]. ADRs are particularly well suited for evaluating the designs of single components before the entire architecture has been solidified [12, 13]. Reviewers are chosen because of their areas of expertise, not simply because of their availability. Each reviewer is given a questionnaire and/or some exercises to complete. The questionnaires and/or the exercises compel them to use the documentation and think about the architecture. For example, an exercise might be, "How would you use the facilities provided by this module to send a message to the user and wait a response?" The reviewer would then be obliged to sketch out the answer in pseudo-code, using facilities described in the design and the documentation. The result is that the artefact being reviewed is actually exercised.

The Software Engineering Institute (SEI), CMU has played a notable role in evolving and flourishing the principles and the practices of reviews that address Parnas and Weiss concerns. They have argued to consider the architecture evaluation as a standard part of the development cycle [12]. With a particular focus on architectural design, the SEI has developed a number of methods. Examples include the Tradeoff Analysis Method (ATAM) [25], the Software Architecture Analysis Method (SAAM) [24], the Active Review for Intermediate Designs (ARID) [11]. These methods have been applied for years on dozens of projects

of all sizes and in a wide variety of domains. Other SEI methods include the Attribute-Based Architectural Styles (ABAS)[27], and the Cost Benefit Analysis Method (CBAM) [26]. Notable efforts outside SEI are the Software Performance Engineering (SPE) [47,48, 55] and ArchOptions [5, 6], our work in progress on the evaluation of software architectures for stability and evolution using options theory.

The evaluation using these methods generally identifies what the quality goals of interest are and then highlights the strengths and weaknesses of the architecture to meet the identified goals. These methods either explicitly address a single quality goal or multi-quality goals of interest.

The contributions of this paper are in three folds: the first fold contributes to a review of the seminal work on software architectures evaluation methods. Many of the ideas presented in this review pertain to the use of software evaluation methods in general. The second fold explicates evaluating software architectures for stability and evolution. It contributes to a definition of architectural stability, formulation, and insights on the evaluation of architectural stability problem. It highlights and critically discusses ArchOptions[5, 6] in span of the surveyed architectural evaluation methods, as it presents our position on the subject. The third fold draws the attention on the use of Architectures Description Languages (ADLs) in supporting the evaluation of software architectures in general and for architectural stability in specific.

The review is further structured as follows. Section 2 lays down the groundwork for evaluating software architectures: it discusses why and when to evaluate an architecture; who is involved in the evaluation; and lists approaches to evaluation. Section 3 provides a comprehensive overlook on software architecture evaluation methods. Section 4 presents methods that explicate evaluating software architectures for stability and evolution. Section 5 draws the attention on the use of Architectures Description Languages (ADLs) in supporting the evaluation of software architectures in general and for architectural stability in specific. Section 6 summarizes.

2 APPROACHES TO EVALUATION

Architecture evaluation can be applied at any stage of an architecture life-time. The classical evaluation of an architecture occurs when the architecture has been specified but before implementation has begun. Users of iterative or incremental life-cycles models can evaluate the architectural decisions made at the end of each iteration or during the most recent architectural cycle. For instance, the Rational Unified Process (RUP) [28] splits the development (evolution) process into four phases. These phases are Inception, Elaboration, Construction, and Transition. The four phases (I, E, C, and T) constitute a development (evolution) cycle and produce a software generation. Under the RUP context, the architectural evaluation can span iteratively and intertwined the Inception phase and iterations of the Elaboration phase, and/or can take place at the Life-Cycle Architecture (LCA) milestone. At the LCA milestone, the detailed system objectives and scope are examined; the choice of the architecture is considered; and the major risks are identified.

Early evaluation need not wait until an architecture is fully specified. It can be used at any stage in the architecture creation process to examine those architectural decisions already made and choose among architectural options that are pending. Early evaluations may take the form of *discovery reviews*. A discovery review is a very early mini-review activity. It aims to analyse whatever "proto-architecture" may have been crafted. The output of a discovery review is an "iterated" or a "revised" set of requirements and an initial architectural approach to satisfying them, which is subject in turn to later iterative evaluation. Note that the architecting process is best conducted iteratively and intertwinedly through requirements, architecting, and validation

Late evaluation is a form of evaluating an existing architecture. It takes place when the architecture already exists and the implementation is complete. This occurs when an organization inherits some sort of legacy system and need be integrated with the existing one. The evaluation at this level helps the new owners understand the legacy system, and let them know whether the system can be counted on to meet its quality and behavioural requirements.

Clements et al. [12] provides two rules of thumb on when to hold the evaluation: i) hold the evaluation when the development team start to make decisions that depend on the architecture; and ii) when the cost of undoing those decisions would outweigh the cost of holding the evaluation.

Generally speaking, architectural evaluation is a human-centred activity. The reviews are typically conducted in the presence of key stakeholders, clients, designers, and the evaluation team. Architecture evaluation may involve thought experiments, modelling, and walking-through scenarios that exemplify requirements, as well as assessment by experts who look for gaps and weaknesses in the architecture based on their experience. The evaluation may be supported by analytic models, simulation tools, and other architectural analysis means (e.g. parsers, …etc). These may be suitable to reason about a single quality goal (e.g. performance), or multi-quality goals of interest.

Abowd et al. [1] broadly categorize existing techniques to architectural evaluation as either questioning, measuring techniques, or hybrid. *Questioning* techniques use scenarios, questionnaires, checklists, and as the like for architectural investigation. *Measuring* techniques use metrics, simulation, prototypes, or experimentations on running systems. Measuring techniques result in quantitative re-

sults. These techniques differ from each other in applicability, but they are all used to elicit discussion about the architecture and increase understanding of the architecture's "fitness" with respect to its requirements. *Hybrid* may combines both questioning and measuring. Most of the architecture evaluation methods described in this review are generally hybrid; they tend to elicit "discussion" about the architecture using questioning techniques and use sorts of measurements for reasoning.

3 EVALUATING SOFTWARE ARCHITECTURES

In subsequent sections, we provide a comprehensive overlook on software architecture evaluation methods. We describe the evolution of the principle and practice to software architecture evaluation through the following methods: the Software Architecture Analysis Method (SAAM) [24]; the Architecture Trade-off Analysis Method (ATAM) [25]; the Active Attribute-Based Architectural Styles (ABASs) [27]; the PASA Software Performance Engineering (SPE) [47, 48, 55]; Reviews for Intermediate Designs (ARID) [11]; and the Cost Benefit Analysis Method (CBAM) [26].

Conceptually, all the architecture evaluation methods described in this review are *Active Design Reviews*: they require the participation of experts for their specific stake in the architecture. They pursue a path of directed analysis such as eliciting a specific statements on quality goal(s) that the architecture must meet to be acceptable, and then follow an measuring path to demonstrate how the architecture satisfy (or does not satisfy) the quality goal(s).

3.1 The Architecture Trade-off Analysis Method (ATAM)

The Architecture Trade-off Analysis Method (ATAM) [25] does not only reveal how well an architecture satisfies particular quality goals, but it also provides insight into how these quality interact with each other – how they *trade off* against each other [12]. ATAM is a *scenario* based architecture evaluation method. A scenario describes the interaction with the system from the stockholder's point of view. The ATAM uses three types of scenarios. These are *use case scenarios*, *growth scenarios*, and *exploratory scenarios*. Use case scenarios describe the typical uses of the completed running system. Growth scenarios represent typical anticipated changes of the system. Exploratory scenarios expose the limits or boundary conditions of the current design; in other words, they tend to expose the extreme changes that are expected to "stress" the system.

The input to the ATAM consists of an architecture, the business goals of a system, and the perspectives of the stakeholders involved with the system. The ATAM achieves its evaluation of an architecture by utilizing an understanding of the architectural approach that is used to achieve particular quality goals and the implications of that approach. The quality attributes that compromise system "utility" (e.g. performance, availability, security, modifiability, usability, and so on) are elicited, specified down to the level of scenarios, annoted with stimuli and responses, and prioritized. The scenarios are used for the evaluators to understand the inherent architectural risks, non-risks, sensitivity points to particular quality attributes, and tradeoffs among quality attributes.

The ATAM can be used at various stages of development (conceptual, before code, during development, or after deployment). The ATAM is fully described in [25, 12].

3.2 The Software Architecture Analysis Method (SAAM)

The Software Architecture Analysis Method (SAAM) [24] elicits stakeholder's input to identify explicitly the quality goals that the architecture is intended to satisfy. Unlike the ATAM, which operates around a broad collection of quality attributes, the SAAM concentrates on attributes for modifiability, variability (suitable for product line), and achievement of functionality. The development of SAAM was motivated by the observation that practitioners regularly make claims about their software architectures (e.g. " This system is more robust than its predecessor", "Using CORBA will make the system easy to modify and upgrade") that are untestable [12]. SAAM tends to make these claims testable; it replaces claims with quality attributes (like maintainability, modifiability, robustness, flexibility, and so forth) and uses scenarios to operationalize these attributes.

SAAM indicates places where the architecture fails to meet its modifiability requirements and in some cases shows obvious alternative designs that would work better. Like ATAM, SAAM is a *scenario-based* method. A scenario in SAAM is a brief description of the some anticipated or desired use of the system. Scenarios are classified as either *direct* or *indirect* scenarios. Direct scenarios are those scenarios that are directly supported by the architecture, meaning that anticipated use require no modification to the architecture for the scenario to be accommodated. An indirect scenario is one that requires a modification to the architecture to be satisfied; the architect describes how the architecture would need to be changed to accommodate the scenario. When two or more indirect scenarios require changes to a single component of an architecture, they are said to *interact* in that component. Areas of high scenario interaction reveal potentially poor separation of concerns in a component; they indicate that the architecture is not documented to the right level of structural decomposition.

The input to SAAM consists of an enumerated set of stakeholder's scenarios that represent known or likely changes that the system will undergo in the future. These scenarios are prioritized and mapped onto the architecture representation. The activity of mapping indicates problem areas in the

architecture: areas where the architecture is overly complex (e.g. if distinct scenarios affect the same component(s)) and areas where changes tend be problematic (e.g. if a scenario causes changes to a large number of components). A complete description of SAAM is provided in [12,24].

3.3 Active Reviews for Intermediate Designs (ARID)

The Active Reviews for Intermediate Designs (ARID) [11] combines the philosophy of ADRs with scenario-based methods like ATAM or SAAM. ARID is a method for evaluating sub-designs of partial architectures in their early or conceptual phases. Designs of partial architectures are architectural in nature; they are sub-designs that represent the stepping stones to the full architecture. ARID aims to validate the suitability of the sub-design being proposed with respect to other parts of the architecture. ARID is motivated by the fact that if the architectural sub-designs are inappropriate, then the entire architecture can be undermined. Hence, reviewing a sub-design in its early pre-release stage provides valuable early insights into the design's viability and allows for timely discovery of errors, inconsistencies, or inadequacies.

ARID can be carried out in the absence of complete documentation. In ARID, the reviewers are the design's stakeholders. Like ATAM and SAAM, the reviewers prepare a set of scenarios. After scenarios are gathered, a winnowing process occurs: two or more scenarios that are versions of the same scenario or one that subsumes another are merged. Prioritisation is by voting: each reviewer is allowed to vote up to 30 percent of the number of scenarios. Beginning with the scenarios that have received the most votes, the reviewers craft code or pseudo-code that uses the design to carry each scenario. At the end of the review, ARID may have helped to discover (i) the sufficiency, fitness, and suitability of the services provided by the design, and (ii) the quality and the completeness of the documentation.

3.4 Attribute-Based Architectural Styles (ABAS)

Attribute-Based Architectural Styles (ABASs) [27] build on *architectural styles* to provide a foundation for reasoning about architectural design. An architectural style is a generic description of an architecture. A style specifies the component types, the topological structure relevant to the specific style, and patterns of data and control interaction among the components. A single architectural style may result in several ABASs, where every ABAS reasons about a specific quality attribute. For example, an architecture with a Client-Server architectural style might have a Security Client-Server ABAS, a Modifiability Client-Server ABAS, a Performance Client-Server ABAS, and so forth.

ABAS explicitly associate a reasoning framework with an architectural style that facilitate the evaluation. The reasoning framework may be quantitatively grounded (example based on rate monotonic analysis, queuing theory, or other metrics which exist in the various quality attribute communities) or it may be qualitative in nature (such as checklists, questionnaires, or scenario-based analysis). For example, associating a Rate Monotonic Analysis to the pipe-and-filter style allows creating the Performance Concurrent Pipelines ABAS. This ABAS supports the architect in reasoning about worst-case latency quantitatively. Similarly, associating a scenario-based reasoning using SAAM or ATAM, allow creating the Modifiability Layering ABAS. This ABAS supports the architect to reason about the effects of changes on modifiability and maintainability. As far as evaluation is concerned, a style may be "stressed" by stimuli on a quality of interest. The objective is to gain insight into the *responses* of the architecture under evaluation to these *stimuli* using the associated quality-specific models. This aids the architect in understanding how to achieve a desired response by manipulating the architectural parameters.

3.5 PASA: Performance Assessment of Software Architectures & Software Performance Engineering (SPE)

Software Performance Engineering (SPE) is systematic quantitative approach to proactively analyse and mange software performance [47,48,55]. The SPE technique can be used to examine an architecture to see whether the designed system will meet it performance goals. It uses model predictions to evaluate trade-offs in software functions, hardware size, quality of results, and resource requirements. It also includes techniques for collecting data, principles and patterns for performance-oriented design, and anti-patterns for recognizing and correcting common performance problems. PASA, a Method for the Performance Assessment of Software Architectures, is SPA based. Participants in PASA are key developers and project managers. Performance assessment starts by the identification of critical *use cases* that are important to the responsiveness or scalability of the system. For each critical use case, the *scenarios* that are important to performance are recognised. Measurable performance objectives are then identified for each key scenario. The architecture is analysed to determine whether it will support the performance objectives. In the face of a performance discrepancy, the designer has many choices to make: the performance requirements can be relaxed, functionality can be omitted, hardware capability can be increased, or alternatives architectural designs for meeting the performance objectives are recommended. Conceptually, PASA resembles the ATAM, in which the singular quality of interest is performance.

3.6 The Cost Benefit Analysis Method (CBAM)

The Cost Benefit Analysis Method (CBAM) [26] is an architecture-centric method for analysing the costs, benefits, and schedule implications of architectural decisions. The CBAM builds upon the ATAM to model the costs and benefits of architectural design decisions and to provide means of optimising such decisions. Conceptually, CBAM continues where the ATAM leaves; it adds a monetary dimension to ATAM as an additional attribute to be traded-off. The CBAM consists of the following steps: i) choosing scenarios and architectural strategies (AS); ii) assessing Quality Attribute (QA) benefits; iii) quantifying the architectural strategies; iv) costs and schedule implications; v) calculating desirability; and vi) making decisions.

Upon completion of the evaluation using CBAM, CBAM could have guided the stakeholders to determine a set of architectural strategies that address the highest priority scenarios. These chosen strategies furthermore represent the optimal set of architectural investments. They are optimal based on: benefit, cost, and schedule. To quantify the architectural strategies benefits, stakeholders are asked to rank each AS in terms of its contribution to each quality attribute. A scale of –1 to +1 is used. A +1 means that this AS has substantial positive effect on the QA (for example, an AS under consideration might have substantial positive effect on performance) and –1 means the opposite. Each AS can be assigned a computed benefit score from –100 to +100. CBAM doesn't provide a way to determine the cost; it considers that cost determination is a well-established component of software engineering and is outside its scope. The benefits and scores result in the ability to calculate desirability metrics for each architectural strategy. The magnitude of desirability can range from 0 to 100.

Plausible improvements of the existing CBAM include the adoption of Real-Options theory to reason about the value of postponing an investment decisions in architectural strategy. For example, let AS2 and AS3 be two architectural strategies, where AS2 is low-cost, low-benefit, and AS3 is high-cost, high-benefit. Analysis of the *dependency structure* may show, for example, that AS2 must be first be implemented, deferring the implementation of AS3. In other word, CBAM uses Real Options theory to calculate the value of option to defer or delay the investment into an architectural strategy until more information will be available.

4 EVALUATING SOFTWARE ARCHITECTURES FOR STABILITY AND EVOLUTION

In subsequent sections, we focus an emerging class of methods that explicates evaluating software architectures for stability and evolution. We define and formulate the problem of evaluating software architectures for stability. We discuss why and how to evaluate an architecture for stability. We differentiate between two types of approaches to evaluation: these are retrospective and predictive [23]. We review notable research effort in this direction. These include: (i) Jazayeri's retrospective method [23] to evaluation, and (ii) ArchOptions [5,6] a predictive approach to the evaluation of software architectures for stability using Real options theory. We critically discuss ArchOptions in span of the reviewed methods.

4.1 On Architectural Stability and Evolution

The architecture represents those design decisions that are hardest to change [42]. If the business goal is that the system should be long-lived, should *evolve* to accommodate future requirements, and should create future value, stability becomes an important goal to evaluate an architecture for. *Architectural stability* is a quality that refers to the extent an architecture is flexible to endure *evolutionary* changes in stakeholder's requirements and the environment, while leaving the architecture intact. An architecture, which lacks flexibility, may entail large and disruptive changes for the requirements to be accommodated. The change may "break" the architecture necessitating changes to the architectural structure (e.g. changes to components and interfaces), architectural topology (e.g. architectural *style*, where a style is a generic description of a software architecture), or even changes to the underlying architectural infrastructure (e.g. middleware). It may be expensive and difficult to change the architecture as requirements evolve [16]. Consequently, failing to accommodate the change leads ultimately to the degradation of the usefulness of the system.

Evaluation of software architectures for stability aims to understand the *impact of evolution* on the architecture of the software system. The impact can be understood through assessing the impact of the change on one or more *architectural aspects*. The aspects may be *structural* (e.g. components, connectors, and topological such as the architectural style used etc.); *quality* (e.g. run time behavioral quality attribute such as performance or static quality attribute such as maintainability); or *economical* (e.g. cost and value implications).

Let A ={*S*, *Q*, *E*} where *S* stands for the structural aspects of the architecture A; *Q* the quality-goals supported by the architecture A; and *E* the economical characteristics of the architecture A. Suppose the architecture need to evolve or has evolved to A'=*{S', Q', E'}*. Stability can be evaluated by *understanding* the impact of evolution on A' relative to A, or *predicting* such impact on the architecture-to-be, A', based on examining A and other relative inputs (e.g. the likely changes, business cases, stakeholders' concerns, standards, requirements, calibrated market information etc.). The impact can be understood or predicted through assessing the impact of the change on one or more architectural aspects.

4.2 Retrospective & Predictive Evaluation for Stability

Approaches to evaluating software architectures for stability can be *retrospective* or *predictive* [23]. Both retrospective and predictive evaluation start with the assumption that the software architecture's primary goal is to guide the system's evolution. Retrospective evaluation looks at successive releases of the software system to analyze how smoothly the evolution took place. The analysis relies on comparing properties from one release of the software to the next. The intuition is to see if the system's architectural decisions remained intact throughout the evolution of the system, that is, through successive releases of the software.

Predictive evaluation for stability provides "insights" on the evolution of the software system based on examining a set of *likely* changes and the extent the architecture can endure these changes. Predictive evaluation for stability is *preventive* with the objective of understanding potential threats on one or more architectural aspect (i.e. structural, economical, or quality) if the *likely* changes need to be accommodated.

Retrospective analysis can be used for empirically evaluating an architecture for stability; calibrating the predictive evaluation results; and predicting trends in the system evolution [23]. In other words, retrospective analysis can also provide basis for predictive analysis. For example, previous evolution data of the system may be used to anticipate the resources needed for the next release, or to identify the components that most likely require attention, need restructuring or replacements, or to decide if it is time to entirely retire the system. In principle, predictive analysis and retrospective analysis should be combined. However, perfect predictive evaluations would render retrospective analysis unnecessary [23].

4.2.1 Retrospective evaluation

Up to the authors' knowledge, the only retrospective evaluation technique to architectural stability is [23]. Jazayeri has applied retrospective analysis to successive releases of a large telecommunication software system. The analysis uses simple metrics such as software size metrics (e.g. module size, number of modules changed, and the number of modules added in the different releases); coupling metrics; and color visualization to summarize the evolution pattern of the software system across the releases. The evaluation appears not to be preventive; it summarizes how smoothly the evolution of the software system has taken place across several releases. The evaluation is done on detailed design artifacts of the system using design metrics such as modules number and coupling metrics. Further, the evaluation uses fine-grained artifacts such as size metrics, which are not architectural in essence. Although the use of these metrics may be justified in the absence of metrics that explicates architectures, Jazayeri has not shown how the reasoning links to the conceptual architecture. Moreover, the evaluation appears to be expensive and unpractical; it requires information to be kept for each release of the software. Yet, such data is not commonly maintained, analyzed, or exploited. The evaluation assumes that the system already exist and has evolved making this approach unsuitable for early evaluation.

4.2.2 Predictive evaluation

ArchOptions [5, 6] is a predictive method for evaluating software architectures for stability. ArchOptions provides a novel model that builds on options theory to predict architectural stability. The model takes *value-based* [9, 45, 50, 51, 52] reasoning to prediction: ongoing work in the economic-driven software engineering research [45] has drawn the attention that software design and engineering activities need to be judged by their contribution to the added value and value creation [9, 50, 51, 52]. This need becomes more intense especially incase of evolution, where the economic value is among the primary considerations that determine whether or not to retire a system, or if the architecture is "evolutionary friendly". This claim is *indirectly* supported by observations [29, 30] and other studies [43], which suggests that evolving software eventually, reaches a condition where, from an economic point of view at least, replacement is indicated [30]. Added that the biggest tradeoffs in large, complex systems always have to do with economics [4, 26].

ArchOptions argues that *real options theory* [39, 40] is well suited to assist evaluating software architectures for stability. The major idea of this work is that the *flexibility* of an architecture to endure changes in stakeholders' requirements and the environment has a value. This value can assist in the evaluation of an architecture for stability. More specifically, flexibility adds to the architecture values in the form of *real options*- that give the right but not a symmetric obligation- to evolve the software system and enhance the opportunities for strategic growth by making future follow-on investments. The added value under the stability context is strategic in essence and not immediate. It takes the form of (i) accumulated savings through enduring the change without "breaking" the architecture; (ii) supporting reuse; (iii) enhancing the opportunities for strategic "growth" (e.g. regarding an architecture as an asset and instantiating the asset to support new market products); and (iv) giving the enterprise a competitive advantage by banking the stable architecture like any other capitalized asset.

ArchOptions can be applied during the early stages of the development life-cycle to *predict* threats of evolution on an architecture. Given likely evolutionary changes, ArchOptions value the *flexibility* of an architecture to expand in the face of these changes. ArchOptions builds on Black and Scholes options pricing model [8] (Noble Prize winning) to achieve this objective.

ArchOptions predictive results can have different usages: valuing the long-term investment in a particular architec-

ture; analysing trade-offs between two or more candidate software architectures for stability; analysing the strategic position of the enterprise- if the enterprise is highly centred on the software architecture (as it is the case in web-based service providers companies e.g. amzon.com); and validating the architecture for evolution.

Critically relating ArchOptions to the exiting architectural evaluations methods presented in section 3:

Existing methods to architectural evaluation have ignored any economic considerations, with CBAM [26] being the only notable exception. The evaluation decisions using these methods tend to be driven by ways that are not connected to, and usually not optimal for value creation. Factors such as flexibility, time to market, cost and risk reduction often have higher impacts on value creation [9]. These factors also provide an indication on the "friendliness" of the architecture to evolution. Hence flexibility is in the essence. In ArchOptions, we link flexibility to value as a way to make the value of stability tangible.

Relating CBAM to our work, the following distinctions can be made: with the motivation to analyse the cost and benefits of architectural strategies, where an architecture strategy is subset of changes gathered from stakeholder, CBAM does not address stability and does not explicitly target evolution. Further, CBAM does not tend to capture the long-term and the strategic value of the specified strategy. ArchOptions, in contrast, views stability as a strategic architectural quality that adds to the architecture values in the form of *growth options*. A growth option [39, 40] is an option to expand with *strategic* importance. When CBAM complements ATAM to reason about qualities related to change such as modifiability, CBAM does not supply rigorous predictive basis for valuing the impact of change. Plausible improvements of the existing CBAM include the adoption of real options theory to reason about the value of postponing investment decisions. CBAM uses Real Options theory to calculate the value of option to defer the investment into an architectural strategy. The delay is based on cost and benefit information. In the context of the real options theory, CBAM tends to reason about the *option to delay* the investment in a specific strategy until more information becomes available as other strategies are met. However, ArchOptions uses real options to value flexibility provided by the architecture to expand in the face of evolutionary requirements, and henceforth referred as the options to expand or growth options.

None of the described methods (described in section 3) addresses stability or explicitly target the evolution of the architecture over time. Even when methods like SAAM and ATAM are used to analyze qualities that are related to change (such as modifiability), the analysis is not calibrated with analytical models of predictive power. This renders their predictive effectiveness as myopic. For example, ATAM and SAAM indicate places where the architecture fails to meet its modifiability requirements and in some cases shows obvious alternative designs that would work better. When used for evaluating modifiability, the input to these methods consist of an enumerated set of stakeholder's scenarios that represent known or likely changes that the system will undergo in the future. These scenarios are prioritised and mapped onto the architecture representation. The activity of mapping indicates problem areas in the architecture: areas where the architecture is overly complex (e.g. if distinct scenarios affect the same component(s)) and areas where changes tend be problematic (e.g. if a scenario causes changes to a large number of components). The approaches to evaluation do not go beyond thought experiments, walking-through scenarios that exemplify requirements, and/or assessment by experts who look for gaps and weaknesses in addressing modifiability based on their experience. ArchOptions, however, provides a predictive model that is built on sound theory.

Conceptually, ArchOptions is an ADR that explicitly focuses on evaluating the stability of architectural design with respect to evolution. In other words, ArchOptions tends to address Parnas and Weiss concerns. It provides structured and disciplined ways to evaluation. It tends to pursue a path of directed analysis: it requires the participation of experts for their specific stake in the architecture and it takes a analytical path based on real options to demonstrate how the architecture satisfy (or does not satisfy) the evaluation goal (i.e. stability).

5 ADLs AND THEIR POTENTIAL SUPPORT FOR SOFTWARE ARCHITECTURES EVALUATION

Although software evaluation methods are typically human-centred, formal notations for representing and analysing architectural designs, generically referred to as *Architectural Description Languages* (ADLs), has provided new opportunities for architectural analysis [18] and evaluation. In this section, we briefly survey efforts on ADLs as they have positive implications on supporting the evaluation of software architectures. We explain how ADLs can be used to support the evaluation of software architectures in general and the evalution of software architectures for stability in specific.

ADLs are languages that provide features for modelling a software system's conceptual architecture [36]. ADLs provide a concrete syntax and a conceptual framework for characterizing architectures [20]. The conceptual framework typically subsumes the ADL's underlying semantic theory (e.g., CSP, Petri nets, finite state machines).

A number of ADLs have been proposed for modeling architectures both within a particular domain and as general-purpose architecture modeling languages [36]. Examples are Aesop [19], Darwin [33, 34], MetaH [54], C2 [37], Rapide [31, 32], Wright [3], UniCon [46], SADL [38], and ACME [20].

Architectural descriptions are often intended to model large, distributed, and concurrent systems. Evaluating the properties of such systems upstream, at the architectural level, can substantially lessen the costs of any errors. The formality of ADL renders them suitable for the manipulation by tools for architectural analysis. In the context of architectural evaluation, the usefulness of an ADL is directly related to the kinds of analysis a particular ADL supports. The types of analyses and evaluation for which an ADL is well suited depends on its underlying semantic model. We refer to [36] to state few examples: Wright is based on CSP; it analyses individual connectors for deadlocks. MetaH and UniCon both support schedulability analysis by specifying non-functional properties, such as criticality and priority. SADL can establish relative correctness of two architectures with respect to refinement map. Rapide's and C2's event monitoring and filtering tools also facilitate analysis of an architecture. C2 uses critics to establish adherence to style rules and design guidelines.

Another aspect of analysis, that supports architectural evaluation, is enforcement of constraints. Parsers and compilers enforce constraints implicit in types, non-functional attributes, component and connector interfaces, and semantic models. Static and dynamic analyses are used. Static analysis verifies that all possible executions of the architecture description conform to the specification. Static analysis helps the developers to understand the changes that need to be made to satisfy the analysed properties. They span approaches such as reachability analysis [21, 22, 53], symbolic model checking [10], flow equations, and data-flow analysis [15]. The applicability of such techniques to architecture descriptions has been demonstrated in [41] using two static analysis tools. These tools are INCA [14] and FLAVERS [15, 35]. Rapide [31, 32] provides a support to simulate the executions of the system. The simulation verifies that the traces of those executions conform to high-level specifications of the desired behaviour. Allen and Garlan [3] use the static analysis tool FDR [17] to prove freedom from deadlock as well as compatibility between the component and connectors in an architecture description. Dynamic software architectures denote that application's architecture evolves during runtime [33, 34]. Darwin [33, 34] and its associated analysis tools is an example that supports dynamic analysis of software architectures.

In the context of evaluating software architectures for stability, no notable research effort has explored the role of ADLs in supporting such evaluation. However, we believe that ADLs have the *potentials* to support such evaluation. For instance comparing properties of an ADL of different releases of the software can provide insights on how the change(s) or the likely change(s) tends to threat the stability of the architecture. This can be achieved by analyzing parts of the new version that represent syntactic and semantic changes. Also the analysis can provide insights into possible architectural breakdown upon accommodating the change. For example, the analysis may show how the change may break the architectural topology (e.g. the architectural style) and/or the architectural structure (e.g. components, connectors, interfaces.. ect.). We note that ADLs have potentials for performing retrospective evaluation for stability. In this context, the evaluation can be performed at correspondingly high level of abstraction. Moreover, the evaluation may be relatively less expensive as when compared, for example, to the approach of [23].

6 SUMMARY

We have reviewed methods for evaluating software architectures. These methods provide frameworks for architects to evaluate architectural decisions with respect to quality attributes that need to be met by the system. They can reason about a single quality goal or multi-quality goals of interest. Examples of these quality attributes include performance, security, modifiability, and suitability of design. They take different approaches to evaluation: questioning, measuring, or both.

If the business goal that the architecture should be long-lived, should evolve to accommodate future requirements, or should create future value, stability becomes an important architectural quality to evaluate an architecture for. We have defined architectural stability and have formulated the problem of the evaluation for stability. We have spotted the light on an emerging class of methods that explicates evaluating software architectures for stability and evolution. These methods assume that the software architecture's primary goal is to guide the system's evolution. They take retrospective or predictive approaches to evaluation. We have critically related our work in progress to the reviewed methods- as it represents our position on the subject.

ADLs have provided new opportunities for architectural analysis and evaluation. We have briefly surveyed efforts on ADLs as they have positive implications on supporting the evaluation of software architectures. We have explained how ADLs can be used to support the evaluation of software architectures in general and the evaluation of software architectures for stability in specific.

7 REFERENCES

1. Abowd, G., Bass, L., Clements,P., Kazman, R., Northrop, L., and Zaremski, A. (1996). Recommended Best Industrial Practice for Software Architecture Evaluation (CMU/SEI-96-TR-025), Software Engineering Institute, Carnegie Mellon University.
2. Abowd, G., Allen, R., and Garlan, D. (1993). Using Style to Understand Descriptions of Software Architecture. In Proceedings of SIGSOFT'93: Foundations of Software Engineering, ACM Press.

3. Allen, R. and Garlan, D. (1994). Formalizing Architectural Connection. In Proceedings of the 14th International Conference on Software Engineering, pp. 71-80.
4. Asundi, J. and Kazman, R. (2001). A Foundation for the Economic Analysis of Software Architectures. In Proceedings of the Third Workshop on Economics-Driven Software Engineering Research.
5. Bahsoon, R. (2003). Evaluating Software Architectures for Stability: A Real Options Approach. Research Abstract. In: Proceedings of the 25 th International Conference on Software Engineering, Portland, USA.
6. Bahsoon, R., and Emmerich, W. (2003). ArchOptions: A Real Options-Based Model for Predicting the Stability of Software Architecture. In: Proceedings of the Fifth Workshop on Economics-Driven Software Engineering Research, EDSER 5, held in conjunction with the 25 th International Conference on Software Engineering.
7. Belady, L.A.and Lehman, M.M. (1976). A Model of Large Program Development. IBM Systems Journal, Vol. (15) 3, pp. 225-252.
8. Black, F., and Scholes, M. (1973). The Pricing of Options and Corporate Liabilities. Journal of Political Economy.
9. Boehm, B. and Sullivan, K.J. (2000). Software Economics: A Roadmap. In: A. Finkelstein (ed): The Future of Software Engineering.
10. Burch, J., Clarke, E., McMillan, E., Dill, D., and Hwang, L. (1990). Symbolic Model Checking: 10^{20} States and Beyond. In: Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science, pp. 428-439.
11. Clements, P. (2000). Active Reviews for Intermediate Designs (CMU/SEI-2000-TN-009), Software Engineering Institute, Carnegie Mellon University.
12. Clements, P., Kazman, R., Klein, M. (2002). Evaluating Software Architectures: Methods and Case Studies. Addison Wesley, Boston, USA.
13. Clements, P. and Northrop, L. (2002). Software Product Lines: Practices and Patterns. Addison Wesley, Boston, USA.
14. Corbett, J., and Avrunin, G. (1995). Using Integer Programming to Verify General Safety and Liveness Properties. Formal Methods in System design. Vol. (6), pp. 97-123.
15. Dwyer, M., and Clarke, L. (1994). Dataflow Analysis for Verifying Properties of Concurrent Programs. In: Proceedings of the Second ACM Sigsoft Symposium on Foundations of Software Engineering. Vol. (19), pp. 62-75.
16. Finkelstein, A. (2000). Architectural Stability, Some Preliminary Comments. http://www.cs.ucl.ac.uk/staff/a.finkels-tein.
17. Formal Systems (Europe) Ltd. (1992). Failures Divergence Refinement: User Manual and Tutorial.
18. Garlan, D. (2000). Software Architecture: A Roadmap. In: A. Finkelstein (ed): The Future of Software Engineering.
19. Garlan, D., Allen, R., and Ockerbloom (1994). Exploiting Style in Architectural Design Environments. In: Proceedings of SIGSOFT'94, Foundations of Software Engineering, New Orleans, Louisiana, USA, pp. 175-188.
20. Garlan, D., Monroe, R. and Wile, D. (1995). ACME: An Architectural Interconnection Language. Technical Report, CMU-CS-95-219, Carnegie Mellon University.
21. Godefroid, P., and Wolper, P. (1991). Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In: Proceedings of the Third Workshop on Computer Aided Verification, pp. 417–428.
22. Holzman, G. (1991). Design and Validation of Computer Protocol. Prentice Hall Software Series.
23. Jazeyeri, M. (2002). On Architectural Stability and Evolution. Lecture Notes in Computer Science, Springer Verlag, Berlin.
24. Kazman, R., Abowd, G., Bass, L. and Webb, M. (1994). SAAM: A Method for Analyzing the Properties of Software Architectures. In: Proceedings of the 16th International Conference on Software Engineering (Sorento, Italy), pp. 81-90.
25. Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T., and Carrière, S.J. (1998). The Architecture Tradeoff Analysis Method. In: Proceedings of ICECCS, Monterey, CA.
26. Kazman, R., Asundi, J., and Klein, M. (2001). Quantifying the Costs and Benefits of Architectural Decisions. In: Proceedings of the 23rd International Conference on Software Engineering, Toronto, Canada, pp. 297-306.
27. Klein, M. and Kazman, R. (1999). Attribute-Based Architectural Styles. CMU/SEI-99-TR-22, Software Engineering Institute, Carnegie Mellon University.
28. Kruchten, P. (2000). The Rational Unified Process: An Introduction. Addison Wesley Longman.
29. Lehman, M.M., Feedback, Evolution and Software Technology, FEAST/2, http://www-dse.doc.ic.ac.uk/~mml/feast/
30. Lehman, M.M., Kahen, G., and Ramil, J.F. (2000). Replacement Decisions for Evolving Software. In

Proceedings of the Second Workshop on Economics-Driven Software Engineering Research.

31. Luckham, D. C., Augustin, L. Kenney, J., Veraa, J, Bryan, M., and Mann W. (1995). Specification Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering. Vol. 21(4), pp. 366-355.
32. Luckham, D.C. and Vera, J. (1995). An Event-Based Architecture Definition Language. IEEE Transactions on Software Engineering, pp. 717-734.
33. Magee, J., and Kramer, J. (1996). Dynamic Structure in Software Architectures. In: Proc. ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering, San Francisco, CA, pp. 3–14.
34. Magee, J., Dulay, D., Eisenbach, N., and Kramer, J. (1995). Specifying Distributed Software Architecture. In: proceedings of the Fifth European Software Engineering Conference (ESEC'95), Barcelona, Spain.
35. Masticola, S. and Ryder, B. (1991). A Model of ADA Programs for Static Deadlock Detection in Polynomial Time. In: Proceedings of the Workshop on Parallel and Distributed Debugging, pp. 97–107.
36. Medvidovic, N. and Taylor, R. (1997). A Framework For Classifying and Comparing Architecture Description Languages. In: Proceedings of the Sixth European Software Engineering Conference, together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, pp. 60-76.
37. Medvidovic, N. and Rosenblum, D., and Taylor, R.(1999). A Language and Environment for Architecture-Based Software Development and Evolution. In: Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, pp. 44-53.
38. Moriconi, M., Qian, X., and Riemenschneider, R. (1995). Correct Architecture Refinement. IEEE Transactions on Software Engineering, pp. 356-372.
39. Myers, S.C. (1977). Determinants of Corporate Borrowing. Journal of Financial Economics. Vol. (5) 2, pp. 147-175.
40. Myers, S.C. (1987). Finance Theory and Financial Strategy. Midland Corporate Finance Journal. Vol. (5) 1, pp. 6-13.
41. Naumovich, G., Avrunin, G.S, Clarke, L.A., and Osterweil, L.J. (1997). Applying Static Analysis to Software Architectures. Technical Report UM-CS-1997-008, University of Massachusetts, Amherst.
42. Parnas, D.L. (1976). On the Design and Development of Program Families. IEEE Transactions on Software Engineering. Vol. (1), pp. 1-9.
43. Parnas, D.L. (1994). Software Aging. In: I6th International Conference on Software Engineering, Sorento, Italy, pp. 279-287.
44. Parnas, D.L. and Weiss, D. (1985). Active Design Reviews: Principles and Practices. In: Proceedings of the 18th International Conference on Software Engineering.
45. Proceedings of the Workshops on Economics-Driven Software Engineering Research (1999- 2003). EDSER 1 to 5. Workshops held in conjunction with the 21st through 25th International Conference on Software Engineering, 1999 to 2003.
46. Shaw, M., DeLine, R., Klein, D., Ross, T., and Young, D. (1995). Abstractions for Software Architecture and Tools to Support them. IEEE Transactions on Software Engineering, pp. 314-335.
47. Smith, C. (1990). Performance Engineering of Software Systems. Addison-Wesley, Reading, Ma.
48. Smith, C. and Woodside, M. (1999). System Performance Evaluation: Methodologies and Applications. CRC Press.
49. Stafford, J.A., Richardson, D..J., and Wolf, A.L. (1997). Chaining: A Software Architecture Dependence Analysis Technique. Department of Computer Science, University of Colorado, Boulder, CO, Technical Report CU-CS-845-97.
50. Sullivan, K.J. (1996). Software Design: The Options Approach. In: 2nd International Software Architecture Workshop, Joint Proceedings of the SIGSOFT '96 Workshops, San Francisco, CA, pp. 15–18.
51. Sullivan, K.J., Chalasani, P., Jha, S. and Sazawal, V. (1999). Software Design as an Investment Activity: A Real Options Perspective, In: Real Options and Business Strategy: Applications to Decision Making, L. Trigeorgis, consulting editor, Risk Books.
52. Sullivan, K.J., Griswold, W., Cai, Y. and Hallen, B. (2001). The Structure and Value of Modularity in Software Design. In: Proc. ESEC/FSE-9, Vienna, Austria, pp. 99-108.
53. Valmari, A. (1991). A Stubborn Attack on State Explosion. In: E. M. Clarke and R. Kurshan, editors, Computer-Aided Verification 90. American Mathematical Society, Providence RI. Number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pp. 25–41
54. Vestal, S. (1996). MetaH Programmer's Manual, Version 1.09. Technical Report. Honeywell Technology Center.
55. Williams, L.G. and Smith, C.U. (1998). Performance Evaluation of Software Architectures. In: Proceedings of the Workshop on Software and Performance (WOSP98), Santa Fe, NM.