Architectural Stability

Rami Bahsoon[1] and Wolfgang Emmerich[2]

[1] School of Computer Science,
The University of Birmingham
Edgbaston, B15 2TT, Birmingham, UK
r.bahsoon@cs.bham.ac.uk
[2] London Software Systems, Dept. of Computer Science, University College London,
Gower Street, WC1E 6BT, London, UK
w.emmerich @cs.ucl.ac.uk

Abstract. One of the major indicators of the success (failure) of software evolution is the extent to which the software system can endure changes in requirements, while leaving the architecture of the software system intact. The presence of this "intuitive" phenomenon is referred to as architectural stability. The concept is still far from being understood and many architectural stability related questions are remained unanswered. Reflecting on our extensive research into the problem, we explore perspectives in handling the problem. We review existing research effort and discuss their limitations. We outline research challenges and opportunities.

1 Introduction

Software requirements, whether functional or non-functional, are generally volatile; they are likely to change and evolve over time. The change is inevitable as it reflects changes in stakeholders' needs and the environment in which the software system works. *Software architecture* is the earliest design artifact, which realizes the requirements of the software system. It is the manifestation of the earliest design decisions, which comprise the architectural structure (i.e., components and interfaces), the architectural topology (i.e., the architectural style), the architectural infrastructure (e.g., the middleware), the relationship among them, and their relationship to the other software artifacts (e.g., low-level design). One of the major implications of a software architecture is to render particular kinds of changes easy or difficult, thus constraining the software's evolution possibilities [1]. A change may "break" the software architecture necessitating changes to the architectural structure (e.g., changes to components and interfaces), architectural topology, or even changes to the underlying architectural infrastructure. It may be expensive and difficult to change the architecture as requirements evolve. Conversely, failing to accommodate the change leads ultimately to the degradation of the usefulness of the system. Hence, there is a pressing need for flexible software architectures that tend to be *stable* as the requirements evolve. By a stable architecture, we mean the extent to which a software system can endure changes in requirements, while leaving the architecture of the software system intact. We refer to the presence of this "intuitive" phenomenon as *architectural stability*.

R. Meersman, P. Herrero, and T. Dillon (Eds.): OTM 2009 Workshops, LNCS 5872, pp. 304–315, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Developing and evolving architectures, which are *stable* in the presence of change and *flexible enough* to be customized and adapted to the changing requirements is one of the key challenges in software engineering [2]. Ongoing research on relating requirements to software architectures has considered the architectural stability problem as an open research challenge [3; 4]. This is because the conflict between requirements volatility and architectural stability is a difficult one to handle [3]. As a result, many architectural stability related questions are remained unanswered [4]: For example, what software architectures (or architectural styles) are stable in the presence of the changing requirements, and how do we select them? What kinds of changes are systems likely to experience in their lifetime, and how do we manage requirements and architectures (and their development processes) in order to manage the impact of these changes? Meanwhile, industrial evidence reveals situations where high requirements volatility is the norm and much of the promise is leaved to the architecture in accommodating the changes. For example, the number of mergers between companies is increasing and this trend is bound to continue. The different divisions of a newly merged company have to deliver unified services to their customers and this usually demands an integration of their IT systems into the core architecture. The time frame is often so short that building a new system is not an option and therefore existing system components have to be integrated into a distributed system architecture to appear as an integrated computing facility. Secondly, the trend of providing new services or evolving existing services to target new customers, devises and platforms, and distribution settings (e.g., mobility setting) is increasing. For example, moving from a fixed distributed setting to mobility carries critical changes, mainly to non-functionalities, such as changes in availability, security, and scalability requirements. Often the core "fixed" architecture falls short in accommodating the requirements; henceforth, changes to the architecture becomes necessary. Thirdly, it is often the case that components are procured off-the-shelf, rather than built from scratch, in response to changes in requirements and then need to be integrated into the core architecture. These components often have incompatible requirements on the hardware and operating system platforms they run on. In many software systems, the architecture is the level that has the greatest inertia when external circumstances change and consequently incurs the highest maintenance costs when evolution becomes unavoidable [5]. Hence, a stable architecture which addresses such changes in requirements within limited resources and shorter time-to-market is a significant asset for surviving the business, cutting down maintenance costs and creating value.

Reflecting on our extensive research into the problem, we define architectural stability and explore perspectives in handling the problem. We review existing research effort and discuss their limitations. We outline research challenges and opportunities.

The paper is further structured as follows. Section 2 looks at architectures and evolution. Section 3 explores perspectives in handling the architectural stability problem. Section 4 outlines research challenges and opportunities. Section 5 concludes.

2 Architecture-Centric Evolution

In Lehman's terminology [6], there are two types of systems: these are E-type systems and S-type systems. E-Type systems that are embedded in real world applications and are used by humans for everyday business functions. Examples might be customer service, order entry, payroll, operating systems, databases engines. S-Type systems are executable models of a formal specification. The success of this software is judged by how well it meets the specification. For E-Type systems the "real world" is dynamic and ever changing. As the real world changes the specification changes and the E-Type systems need to adapt to these changes. Hence, E-Type systems tend to be evolvable. For S-Type systems the specification becomes invalid in the presence of change. In this paper, we deal with evolution and architectural stability of E-type systems.

In software engineering, it has been known that focusing the change on program code leads to loss of structure and maintainability [7]. Upon managing the change of requirements considerable emphasis is thus placed on the architecture of the software system as the key artifact involved [2]. *Architecture-centric evolution* approaches pursue the software architecture as the appropriate level of abstraction for reasoning about, managing and guiding the evolution of complex software systems, and "synchronizing" the software requirements with its detailed design and implementation. A distinctive feature of these approaches is that they explicitly account for the non-functional requirements, the so-called quality attributes. As the quality attributes comprise the most substantial properties of the system, the evolution of such properties can be best reasoned about and managed at the architectural level. For example, the current trend is to build distributed systems architectures with middleware technologies such as Java 2 Enterprise Edition (J2EE) and the Common Object Request Broker Architecture (CORBA), resulting in the so-called middleware-induced architectures. Middleware-induced architectures follow an architectural-centric evolution approach, as the emphasis is placed on the induced architecture for simplifying the construction of distributed systems by providing high-level primitives for realizing quality attributes. Another example is from product-line architectures. Product lines, a family of products sharing the same architecture, inherently require domain-specific variation and evolution of various products. Due to the higher level of interdependency between the various software artifacts in a product-line, software evolution is too complex to be dealt with at the code level. An essential property of these architectures is that they should be stable over the projected life of the system.

3 Perspectives into Architectural Stability

3.1 Requirements Engineering Perspective

Ongoing research on relating requirements to software architectures has considered the architectural stability problem as an open research challenge and difficult to handle. [4] proposed the "Twin Peaks" model, a partial and simplified version of the spiral model. The cornerstone of this model is that a system's requirements and its architecture are developed concurrently; that is, they are "inevitably intertwined" and

their development is interleaved. [4] advocated the use of various kinds of patterns – requirements, architectures, and designs- to achieve the model objectives. As far as architectural stability is concerned, Nuseibeh had only exposed a tip of the "iceberg": development processes that embody characteristics of the Twin Peaks are the first steps towards developing architectures that are stable in the face of inevitable changes in requirements. Nuseibeh noted that many architectural stability related questions are difficult and remain unanswered. Examples include: what software architectures (or architectural styles) are stable in the presence of changing requirements, and how do we select them? What kinds of changes are systems likely to experience in their lifetime, and how do we manage requirements and architectures (and their development processes) in order to manage the impact of these changes?

With the motivation of bridging the gaps between requirements and software architectures, [3] noted that the goal-oriented approach to requirements engineering may support building and evolving software architectures guaranteed to meet its functional and non-functional requirements. In [8; 9], we reflected on the architectural stability problem with a particular focus on developing distributed software architectures induced by middleware. We advocated adjusting requirements elicitation and management techniques to elicit not just the current non-functional requirements, but also to assess the way in which they will develop over the lifetime of the architecture. These ranges of requirements may then inform the selection of distributed components technology, and subsequently the selection of application server products. Specifically, we considered the architecture stability problem from the distributed components technology in the face of changes in *non-functional requirements*. We argued that addition or changes in functional requirements could be easily addressed in distributed component-based architectures by adding or upgrading the components in the business logic. However, changes in non-functional requirements are more critical; they can stress architecture considerably, leading to architectural "breakdown". Such a "breakdown" often occurs at the middleware level and is due to the incapability of the middleware to cope with the change in non-functional requirements (e.g., increased load demands). This may drive the architect/developer to consider ad-hoc or propriety solutions to realize the change, such as modifying the middleware, extending the middleware primitives, implementing additional interfaces, etc. Such solutions could be costly and unacceptable [9].

3.2 A Value-Driven Design Perspective

An established route to manage the change and facilitate evolution is a universal "design for change" philosophy, where the architecture is conceived and developed such that evolution is possible [12]. Parnas's notion of the "design for change" is based on the recognition that much of the total lifecycle cost of a system is expended in the change and incurred in evolution. A system that is not designed for evolution will incur tremendous costs, which are disapropionate to the benefits. For a system to create value, the cost of a change increment should be proportional to the benefits delivered [21]. "Design for change" is thus promoted as a value-maximizing strategy provided one could anticipate changes [27]. The "Design for change" philosophy is believed to be a useful heuristic for developing flexible architectures that tend to be

stable as requirements evolve. However, the challenge is that there is a general lack of adequate models and methods, which connect this technical engineering philosophy to value creation under given circumstances [27].[27] notes, "The problem in the field is that no serious attempt is made to characterize the link between structural decisions and value added". That is, the traditional focus of software architecture is more on structural and technical perfection than on value. In addressing the architectural stability problem, linking structural decisions to future value becomes more necessary, as presumably evolvable but stable architecture should add value to the system that outweigh what expended in designing for change, as the change materializes[9]. Furthermore, from an economic perspective, the change in requirements is a source of uncertainty that confronts an architecture during the evolution of the software system. The change places the investment in a particular architecture at risk. Conversely, designing for change incurs upfront costs and may not render future benefits. The benefits are uncertain, for the demand and the nature of the future changes are uncertain. The worthiness of designing or re-engineering an architecture for change should involve a tradeoff between the upfront cost of enabling the change and the future value added by the architecture, if the change materializes. The value added, as a result of enabling the change on a given architecture, is a powerful heuristic which can provide a basis for analyzing: (i) the worthiness of designing for change, (ii) the worthiness of re-engineering the architecture, (iii) the retiring and replacement decisions of the architecture or its associated design artifacts, (iv) the decisions of selecting an architecture, architectural style, middleware, and/or design with desired stability requirements, and/or (v) the success (failure) of evolution. In ArchOptions [9], we have taken an economics-driven software engineering perspective to evaluate architectural stability using real options theory. We have adopted the view that software design and engineering activity is one of investing valuable resources under uncertainty with the goal of maximizing the value added. In particular, we have viewed evolving software as a value-seeking and value-maximizing activity: software evolution is a process in which software is undergoing an incremental change and seeking value [9]. We attribute the added value to the *flexibility* of the architecture in enduring changes in requirements. Means for achieving flexibility are typical architectural mechanisms or strategies that are built-in or adapted into the architecture with the objective of facilitating evolution and future growth. This could be in response to changes in functional (e.g., changes in features) or non-functional requirements (e.g., changes in scalability demands). As we are assuming that the added value is attributed to flexibility, arriving at a "more" stable software architecture requires finding an architecture which maximizes the yield in the embedded or the adapted *flexibility* in an architecture relative to the likely changing requirements [9]. Optimally, a stable architecture is an architecture that shall add value to the enterprise and the system as the requirements evolve. By valuing the flexibility of an architecture to change, we have aimed at providing the architect/analyst with a useful tool for reasoning about a crucial but previously intangible source of value. This value can be then used for deriving "insights" into architectural stability and investment decisions related to evolving software. To value flexibility, we have contributed to a novel model, ArchOptions, which builds on an analogy with real options theory [9]. The model examines some critical likely changes in requirements and values the extent to which the architecture is flexible to endure

these changes. The model views an investment in an architecture as an upfront investment plus "continual" increments of future investments in likely changes in requirements. We have applied ArchOptions to two architecture-centric evolution problems: assessing the worthiness of re-engineering a "more" stable architecture in face of likely changes in future requirements, where we have taken refactoring as an example of re-engineering [9]; and informing the selection of a "more" stable middleware-induced software architecture in the face of future changes in non-functional requirements [9]. Our perspective has provided a compromise through linking technical issues to value creation. The approach has the promise to provide insights and a basis for analyses to support many of the concerns highlighted in previous sections.

3.3 Architectural Evaluation Perspective

Evaluating architectural stability aims at assessing the extent to which the system of a given architecture is evolvable, while leaving the architecture and its associated design decisions unchanged as the requirements change. Approaches to evaluating software architectures for stability can be *retrospective* or *predictive* [1]. Both approaches start with the assumption that the software architecture's primary goal is to facilitate the system's evolution. Retrospective evaluation looks at successive releases of the software system to analyze how smoothly the evolution took place. Predictive evaluation provides insights into the evolution of the software system based on examining a set of *likely* changes and the extent to which the architecture can endure these changes.

Retrospective Approaches. Jazayeri [1] motivated the use of retrospective approaches for evaluating software architectures for stability. His analyses rely 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. Jazayeri refers to this "intuitive" phenomenon as architectural stability. Retrospective analysis can be used for empirically evaluating an architecture for stability; calibrating the predictive evaluation results; and predicting trends in the system evolution [1] or to identify the components most likely that require attention, need restructuring or replacements, or to decide if it is time to entirely retire the system. Jazayeri's approach uses simple metrics such as software size metrics, coupling metrics, and color visualization to summarize the evolution pattern of the software system across its successive releases. The evaluation assumes that the system already exists and has evolved. This approach is therefore tend to be unpreventive and unsuitable for early evaluation (unless the evolution pattern is used to predict the stability of the next release).

Predictive Approaches. Predictive approaches to evaluating architectural stability can be applied during the early stages of the development life cycle to predict threats of the change on the stability of the architecture. Unlike retrospective approaches, predictive approaches are *preventive*; the evaluation aims to understand the impact of the change on the stability of the architecture *if* the *likely* changes need to be accommodated, so corrective design measures can be taken. Therefore, in predictive

approaches, the effort to evaluation is justified as the evaluation is generally cost effective, when compared to retrospective approaches.

A comprehensive survey [10] of architectural evaluation methods indicates that current approaches to architectural evaluation focus explicitly on construction and only implicitly, if not at all, on the phenomenon of software "evolution". The survey includes representative methods like ATAM [11], SAAM [12], ARID [14], PASA/SPE [15] and ABAS [17]. These methods provide frameworks for software architects to evaluate architectural decisions with respect to quality attributes such as performance, security, reliability, and modifiability. Despite the concern with "change" and accommodating changes, none of these methods, addresses stability of an architecture over time. 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. 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 high impact on value creation [16]. Such ignorance is in stark contrast to the objective of architectural evaluation and where cost reduction, risk mitigation, and long-term value creation are among the major drivers behind conducting an evaluation for stability [9]. Such provision is important for it assists the objective assessment of the lifetime costs and benefits of evolving software, and the identification of legacy situations, where a system or component is indispensable but can no longer be evolved to meet changing needs at economic cost [5]. Interested reader may refer to [9], where we have highlighted the requirements for evaluating architectural stability, which address the pitfalls in existing methods.

4 Architecture Stability: Challenges and Opportunities

Rapid technological advances and industrial evidence are showing that the architecture is creating its own maintenance, evolution, and economics problems. Part of the problem stems in (i) the rapid technological advancements where evolution is not limited to a specific domain but extends to "horizontally" cover several domains, (ii) the current practice in engineering requirements, which ignore the above, (iii) and the improper management of the evolution of these requirements and across different design artifacts of the software system. In subsequent sections, we highlight some open issues that future research may consider to address some architectural-centric software evolution problems.

4.1 Architectures Description Languages

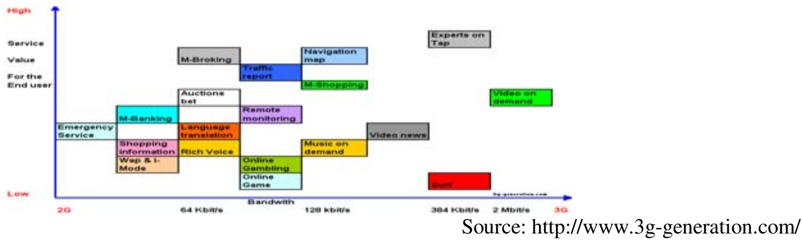
Although software evaluation methods are typically human-centred, formal notations for representing and analyzing architectural designs, generically referred to as Architectures Description Languages (ADLs), have provided new opportunities for architectural analysis [2] and validation ADLs are languages that provide features for modelling a software system's conceptual architecture [18]. ADLs provide a concrete syntax and a conceptual framework for characterizing architectures. Examples are ACME[19], Darwin [20], C2, Rapide [22], Wright [14], UniCon [23], SADL [24],

etc. ADLs 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 kind of analyses a particular ADL tends to support. The type of analyses and evaluation for which an ADL is well suited depends on its underlying semantic model.

No notable research effort has explored the role of ADLs in supporting evaluating architectural stability. However, ADLs have the *potential* to support such evaluation. For instance comparing properties of ADL specifications for different releases of a 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 the parts of newer versions that represent syntactic and semantic changes. Moreover, 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., style) and/or the architectural structure (e.g., components, connectors, interfaces ect.). We note that ADLs have potential for performing retrospective evaluation for stability, where the evaluation can be performed at a correspondingly high level of abstraction. Hence, the evaluation may be relatively less expensive as when compared, for example, to the approach taken by [1].

4.2 Coping with Rapid Technological Advancements and Changes in Domain

Assume that a distributed e-shopping system architecture, which relies on a fixed network needs to evolve to support new services, such as the provision of mobile e-shopping. Moving to mobility, the transition may not be straightforward: the original distributed system's architecture may not be respected, for mobility poses its own non-functional requirements for dynamicity that are not prevalent in traditional distributed setting such as change in location; resource availability; variability of network bandwidth; the support of different communication protocols; losses of connectivity when the host need to be moved; and so forth. These requirements may not be satisfied by the current fixed architecture, the built-in architectural caching mechanisms, and/or the underlying middleware. Replacement of the current architecture may be required. The challenge is thus to cope with the co-evolution of both the architecture and the non-functional requirements as we change domains. This poses challenges in understanding the evolution trends of non-functional requirements; designing architectures, which are aware of how these requirements will change over the projected lifetime of the software system and tend to evolve through the different domains. In software engineering, the use of technology roadmapping, for example, is left unexplored in predicting and eliciting change in requirements. Technology roadmapping is an effective technology planning tool which help identifying product needs, map them into technology alternatives, and develop project plans to ensure that the required technologies will be available when needed [25]. Technology roadmapping, as a practice, emerged from industry as a practical method of planning for new technology and product requirements. According to [25], a roadmap is not a prediction of future breakthroughs in the technology, but rather an articulation of requirements to support future technical



Source: http://www.3g-generation.com/

Fig. 1. Company's x technology road mapping showing the evolution of its mobile services as it moves from 2G to 3G and its value to the end user

needs. A roadmap is part of the business and/or the product strategy towards growth and evolution.

Figure 1 is a product roadmapping of Company x, a mobile service provider. Figure 1 shows how the mobile services are said to evolve as we transit from 2G to 3G networking. As the bandwidth is improved, an emerging number of content-based services, ranging from voice, multi-media, data, and location-based services might be possible. This, in turn, will translate into future requirements (functional and non-functional), which need to be planned in advance so it can be accommodated by the architecture responsible for delivering the services. Note that many of the likely changes in the requirements may be derived from the roadmapping process, rather than the roadmap itself. As an example, M-banking is a service, which allows customers to check bank balances, view statements, and carry bank transactions using mobile phones. A distributed architecture of a banking system, which envisions providing such a service as the bandwidth is improved, may need to anticipate changes due to mobility like changes in security requirements, load, availability, etc. The architect may anticipate relevant change scenarios and ways of accommodating them on the architecture.

4.3 Traceability of Requirements to Architectures

According [7], there is a positive feedback between the loss of software architecture coherence and the loss of software knowledge. Less coherent architectures requires more extensive knowledge in order to evolve the system of the given architecture. However, if the knowledge necessary for evolution is lost, the changes in the software will lead to faster deterioration of the architecture. Hence, planning for evolution and stable software architectures urges the need for traceability techniques, which traces requirements and their evolution back and forth into the architecture and aid in "preserving" the team knowledge.

We define *requirement to architecture traceability* as the ability to describe the "life" of a requirement through the requirements engineering phase to the architecture phase in both forwards and backwards. Forwards demonstrates which (and how) architectural element(s) satisfy an individual requirement in the requirements specification. Backwards demonstrates which requirement(s) in the requirements specification an individual architectural element relate to and satisfy. Current architectural practices, however, do not provide support for traceability from the requirements specification to the architectural description. Maintaining traceability

"links" is necessary for managing the change, the co-evolution of both the requirements and the architecture, confining the change, understanding the change impact on both the structure and the other requirements, providing a support for automated reasoning about a change at a high level of abstraction. Further, such traceability "links" make it easier to preserve the acquired knowledge of the team through guided documentation. This may then minimize the impact of personnel losses, and may allow the enterprise to make changes in the software system without damaging the architectural integrity and making the software system unevolvable.

4.4 Architectural Change Impact Analysis

Although change impact analysis techniques are widely used at lower levels of abstractions (e.g., code level) and on a relatively abstract levels (e.g., classes in O.O. paradigm), little effort has been done on the architectural level (i.e., architectural impact analysis).

Notable effort using dependency analysis on the architectural level includes the "chaining" technique suggested by [26]. The technique is analogous in concept and application to program slicing. In chaining, dependence relationships that exist in an architectural specification are referred to as links. Links connect elements of the specification that are directly related. The links produce a chain of dependencies that can be followed during analysis. The technique focuses the analysis on components and their interconnections. Forward and/or backward chaining are then performed to discover related components. The applicability of this technique is demonstrated on small scale architectures and could be extended to address current architectural development paradigms. For example, how such a concept could be refined to perform what-if analysis on large-scale software architectures such as product-line or model-driven architectures? For product-line architectures, this is necessary for reasoning about how the change could impact the commonality, variability, and their interdependence. These techniques could be then complemented by analysis tools, which could facilitate automated reasoning and provide a basis for what-if analyses to manage the change across instances of the core architecture. Understanding how the change could then ripple across different products might be feasible. For model-driven architectures, for example, this could help in reasoning about how the change could affect the Platform Independent Model (PIM) and ripple to affect the Platform Specific Models (PSM). These techniques could be complemented by automated reasoning to manage evolution. When combined with traceability links, the combination could provide a comprehensive framework for managing the change and guiding evolution.

5 Conclusion

Reflecting on our research into the problem, we have defined architectural stability and explored perspectives in handling the problem. We have reviewed existing research effort, have discussed their limitations, and have outlined research challenges and opportunities. The implications of such contribution need not be overstated: advancing the understanding of the architectural stability, stimulating and possibly motivating future research in architectural stability and related problems.

References

1. Jazayeri, M.: On Architectural Stability and Evolution. LNCS, pp. 13–23. Springer, Heidelberg (2002)
2. Garlan, D.: Software Architecture: A Roadmap. In: Finkelstein, A. (ed.) The Future of Software Engineering, pp. 91–101. ACM Press, New York (2000)
3. van Lamsweerde, A.: Requirements Engineering in the Year 00: A Research perspective. In: Proc. 22nd Int. Conf. on Software Engineering, pp. 5–19. ACM Press, New York (2000)
4. Nuseibeh, B.: Weaving the Software Development Process between Requirements and Architectures. In: Proc. of the First Int. workshop from Software Requirements to Architectures, Toronto, Canada (2001)
5. Cook, S., Ji, H., Harrison, R.: Dynamic and Static Views of Software Evolution. In: Int. Conf. on Software Maintenance, Florence, Italy, pp. 592–601. IEEE CS, Los Alamitos (2001)
6. Lehman, M.M.: Feedback, Evolution and Software Technology, FEAST 1-2, http://www-dse.doc.ic.ac.uk/~mml/feast/
7. Bennet, K., Rajilich, V.: Software Maintenance and Evolution: A Roadmap. In: Finkelstein, A. (ed.) The Future of Software Engineering, pp. 73–90. ACM Press, New York (2000)
8. Emmerich, W.: Software Engineering and Middleware: A Road Map. In: Finkelstein, A. (ed.) Future of Software Engineering, pp. 117–129. ACM Press, New York (2000b)
9. Bahsoon, R.: Evalauting Architectural Stability with Real Options Theory, PhD thesis, U. of London, UK (2005)
10. Bahsoon, R., Emmerich, W.: Evaluating Software Architectures: Development, Stability, and Evolution. In: Proc. of IEEE/ACS Computer Systems and Applications, pp. 47–57. IEEE CS Press, Los Alamitos (2003a)
11. Kazman, R., Klein, M., Barbacci, M., Lipson, H., Longstaff, T., Carrière, S.J.: The Architecture Tradeoff Analysis Method. In: Proc. of 4th. Int. Conf. on Engineering of Complex Computer Systems, pp. 68–78. IEEE CS Press, Los Alamitos (1998)
12. Kazman, R., Abowd, G., Bass, L., Webb, M.: SAAM: A Method for Analyzing the Properties of Software Architectures. In: Proc. of 16th Int. Conf. on Software Engineering, pp. 81–90. IEEE CS, Los Alamitos (1994)
13. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Transaction on Software Engineering 5(2) (1979)
14. Allen, R., Garlan, D.: Formalizing Architectural Connection. In: Proc. of the 14th Int. Conf. on Software Engineering, pp. 71–80. ACM Press, New York (1994)
15. Smith, C., Woodside, M.: System Performance Evaluation: Methodologies and Applications. CRC Press, Boca Raton (1999)
16. Boehm, B., Sullivan, K.J.: Software Economics: A Roadmap. In: Finkelstein, A. (ed.) The Future of Software Engineering, pp. 320–343. ACM Press, New York (2000)
17. Klein, M., Kazman, R.: Attribute-Based Architectural Styles. CMU/SEI-99-TR-22, Software Engineering Institute (1999)
18. Medvidovic, N., Taylor, R.: A Framework for Classifying and Comparing Architecture Description Languages. In: Proc. of 6th. European Software Engineering Conf., with the Fifth ACM SIGSOFT Symp. on the Foundations of Software Engineering, pp. 60–76. ACM Press, New York (1997)
19. Garlan, D., Monroe, R., Wile, D.: ACME: An Architectural Interconnection Language. Technical Report, CMU-CS-95-219 (1995)

20. Magee, J., Dulay, D., Eisenbach, N., Kramer, J.: Specifying Distributed Software Architecture. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989, pp. 137–153. Springer, Heidelberg (1995)
21. Parnas, D.L.: On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the Association of Computing Machinery 15(12), 1053–1058 (1972)
22. Luckham, D.C., Vera, J.: An Event-Based Architecture Definition Language. IEEE Trans. on Software Engineering 29(9), 717–734 (1995)
23. Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D.: Abstractions for Software Architecture and Tools to Support them. IEEE Transactions on Software Engineering 21(4), 314–335 (1995)
24. Moriconi, M., Qian, X., Riemenschneider, R.: Correct Architecture Refinement. IEEE Trans. on Software Engineering 21(4), 356–372 (1995)
25. Schaller, R.R.: Technology Roadmaps: Implications for Innovation, Strategy, and Policy, The institute of Public Policy, George Mason University Fairfax, VA (1999)
26. Stafford, J.A., Wolf, A.W.: Architecture-Level Dependence Analysis for Software System. International Journal of Software Engineering and Knowledge Engineering 11(4), 431–453 (2001)
27. Sullivan, K.J., Chalasani, P., Jha, S., Sazawal, V.: Software Design as an Investment Activity: A Real Options Perspective. In: Trigeorgis, L. (ed.) Real Options and Business Strategy: Applications to Decision-Making, pp. 215–260. Risk Books (1999)