Computer Science at Kent

Architectural Description of Dependable Software Systems

Cristina Gacek

School of Computing Science
University of Newcastle upon Tyne, UK

Rogério de Lemos

Computing Laboratory
University of Kent, UK

This paper is a shorter version of the paper to appear in *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, D. Besnard, C. Gacek, and C. B. Jones (eds), Springer, ISBN: 1-84628-110-5, 2006, pp. 127-142.

This technical report will also be published by the School of Computing Science at the University of Newcastle upon Tyne.

Technical Report No. 9-05
October 2005

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent CT2 7NF, UK

Architectural description of dependable software systems

Cristina Gacek[1] and Rogério de Lemos[2]

[1]University of Newcastle upon Tyne, [2]University of Kent

1 Introduction

The structure of a system is what enables it to generate the system's behaviour, from the behaviour of its components (see Chapter 1). The architecture of a software system is an abstraction of the actual structure of that system. The identification of the system structure early in its development process allows abstracting away from details of the system, thus assisting the understanding of broader system concerns [11].

One of the benefits of a well-structured system is the reduction of its overall complexity, which in turn should lead to a more dependable system. The process of system structuring may occur at different stages of the development or at different levels of abstraction. Reasoning about dependability at the architectural level has lately grown in importance because of the complexity of emerging applications, and the trend of building trustworthy systems from existing untrustworthy components. There has been a drive from these new applications for dependability concerns to be considered at the architectural level, rather than late in the development process. From the perspective of software engineering, which strives to build software systems that are free of faults, the architectural consideration of dependability compels the acceptance of faults, rather than their avoidance. Thus the need for novel notations, methods and techniques that provides the necessary support for reasoning about faults at the architectural level. For example, notations should be able to represent non-functional properties and failure assumptions, and techniques should be able to extract from the architectural representations the information that is relevant for evaluating the system architecture from a certain perspective.

2 Software architectures and ADLs

The software architecture of a program or a software system is the structure or structures of the system, which comprises software components, their externally visible

properties and their relationships [2]. It is a property of a system, and as such it may be documented or not. Being the result of some of the first and most important decisions taken about the system under development [3], it is recognised that the software architecture is a key point for the satisfaction of dependability related requirements. A software architecture is usually described in terms of its components, connectors and their configuration [9][12]. The way a software architecture is configured defines how various connectors are used to mediate the interactions among components.

Architecture description languages (ADLs) aim to support architecture-based development by providing a (semi) formal notation to represent architectures, with their abstractions and structures. Some ADLs also provide a corresponding analysis and/or development environment. The number and variety of ADLs in existence today is quite considerable, but it should be noted that most have only been used in research environments and have not really been widely adopted by industry. Many ADLs only support a specific architectural style.

3 Architecting dependability

Although there is a large body of research in dependability, architectural level reasoning about dependability is only just emerging as an important theme in software engineering. System dependability is measured through its attributes, such as reliability, availability, confidentiality and integrity, and there are several means for attaining these attributes, which can be grouped into four major categories [1]: rigorous design, verification and validation, fault tolerance, and system evaluation.

Rigorous design, also known as fault prevention, is concerned with all the development activities that introduce rigor into the design and implementation of systems for preventing the introduction of faults or their occurrence during operation. Development methodologies and construction techniques for preventing the introduction and occurrence of faults can be described respectively from the perspective of development faults and configuration faults (a type of interaction fault) [1]. In the context of software development, the architectural representation of a software system plays a critical role in reducing the number of faults that might be introduced [6]. For the requirements, architecture allows to determine what can be built and what requirements are reasonable. For the design, architecture is a form of high-level system design that determines the first, and most critical, system decomposition. For the implementation, architectural components correspond to subsystems with well-defined interfaces. For the maintenance, architecture clarifies design, which facilitates the understanding of the impact of changes. One way of preventing development faults from being introduced during the development of software systems is the usage of formal or rigorous notations for representing and analysing software at key stages of their development. The starting point of any development should be the architectural model of a system in which dependability attributes of its components should be clearly documented, together with the static and dynamic properties of their interfaces. Also as part of these models, assumptions should be documented about the required and provided behaviour of the components, including their failure assumptions. This architectural representation introduces an abstract level for reason-

ing about structure of a software system and the behaviour of its architectural elements, without getting into lower level details. The role of architecture description languages (ADLs) is to describe software systems at higher levels of abstraction in terms of their architectural elements and the relationships among them [4].

Verification and validation, also known as fault removal, is concerned with all development and post-deployment activities that aim at reducing the number or the severity of faults [1]. The role of architectural representations in the removal of faults during development is twofold: first, it allows faults to be identified and removed early in the development process; and second, it also provides the basis for removing faults late in the process. The early removal of faults entails checking whether the architectural description adheres to given properties associated with a particular architectural style, and whether the architectural description is an accurate representation of the requirements specifications. The late removal of faults entails checking whether the implementation fulfils the architectural specification. While early fault removal is essentially obtained through static analysis, late fault removal is gained through dynamic analysis. Examples of techniques for the static analysis of architectural representations are inspections and theorem proving, while model checking and simulation could be given as examples of dynamic analysis techniques. Testing is a dynamic analysis technique that has been mostly applied to uncover faults late in the development process. The role of architectural representation in the removal of faults after system deployment includes both corrective and preventative maintenance [1]. The software architecture, in terms of components and connectors, provides a good starting point for revealing the areas a prospective change will affect [4].

Fault tolerance aims to avoid system failure via error detection and system recovery [1]. Error detection at the architectural level relies on monitoring mechanisms, or probes, for observing the system states to detect those that are erroneous at the components interfaces or in the interactions between these components. On the other hand, the aim of system recovery is twofold. First, eliminate errors that might exist at the architectural state of the system. Second, remove from the system architecture those elements or configurations that might be the cause of erroneous states. From the perspective of fault tolerance, system structuring should ensure that the extra software involved in error detection and system recovery provides effective means for error confinement, does not add to the complexity of the system, and improves the overall system dependability [10]. To leverage the dependability properties of systems, solutions are needed at the architectural level that are able to guide the structuring of undependable components into a fault tolerant architecture. Hence from the dependability perspective, one of the key issues in system structuring is the ability to limit the flow of errors. Architectural abstractions offer a number of features that are suitable for the provision of fault tolerance. They provide a global perspective of the system, enabling high-level interpretation of system faults, thus facilitating their identification. The separation between computation and communication enforces modularisation and information hiding, which facilitates error confinement, detection and system recovery. Moreover, architectural configuration is an explicit constraint that helps to detect any anomalies in the system structure. The role of software architectures in error confinement needs to be approached from two

distinct angles. On the one hand is the support for fostering the creation of architectural structures that provide error confinement, and on the other hand is the representation and analysis of error confinement mechanisms. Explicit system structuring facilitates the introduction of mechanisms such as program assertions, pre- and post-conditions, and invariants that enable the detection of potential erroneous states in the various components. Thus, having a highly cohesive system with self-checking components is essential for error confinement. However, software architectures are not only composed of a set of components, connectors are also first class entities and as such also require error confinement mechanisms. For error handling during system recovery, exception handling has shown to be an effective mechanism if properly incorporated into the structure of the system. Architectural changes, for supporting fault handling during system recovery, can include the addition, removal, or replacement of components and connectors, modifications to the configuration or parameters of components and connectors, and alterations in the component/connector network's topology [8]. A good example of such an approach is the architectural mechanisms that allow a system to adapt at run-time to varying resources, system errors and changing requirements [7]. Another repair solution of run-time software, which is architecturally-based, relies on events and connectors to achieve required structural flexibility to reconfigure the system on the fly, which is performed atomically [10].

System evaluation, also known as fault forecasting, is conducted by evaluating systems' behaviour with respect to fault occurrence or activation [1]. For the architectural evaluation of a system, instead of having as a primary goal the precise characterisation of a dependability attribute, the goal should be to analyse at the system level what is the impact upon a dependability attribute of an architectural decision [5]. The reason is that, at such early stage of development the actual parameters that are able to characterise an attribute are not yet known, since they are often implementation dependent. Nevertheless, the architectural evaluation of a system can either be done qualitatively or quantitatively. Qualitative architectural evaluation aims to provide evidence as to whether the architecture is suitable with respect to some goals and problematic towards other goals. In particular, the architectural evaluation of system dependability should be performed in terms of the system failure modes, and the combination of component and/or connector failures that would lead to system failure. Qualitative evaluation is usually based on questionnaires, checklists and scenarios to investigate the way an architecture addresses its dependability requirements in the presence of failures [5]. Quantitative architectural evaluation aims to estimate in terms of probabilities whether the dependability attributes are satisfied. The two main approaches for probabilities estimation are modelling and testing. For the modelling approach, two techniques could be used: architectural simulation, and metrics extracted from the architectural representation. Examples of such metrics are, coupling and cohesion metrics for evaluating the degree of architectural flexibility for supporting change, and data-flow metrics for evaluating performance. However, in terms of dependability, most of the approaches rely on the construction of stochastic processes for modelling system components and their interactions, in terms of their failures and repairs.

4 Conclusions

From the perspective of dependability, effective structuring should aim to build fault-free systems (fault avoidance) and systems that cope with faults (fault acceptance) [1]. At the architecture level, fault avoidance is achieved by describing the behaviour and structure of systems rigorously or formally (rigorous design), and by checking system correctness and the absence of faults (verification and validation). Fault acceptance is related to the provision of architectural redundancies that allow the continued delivery of service despite the presence of faults (fault tolerance), and the assessment whether the specified system dependability can be achieved from its architectural representation (system evaluation). There are no ADLs that are able to deal with a wide range of criteria for representing and analysing the dependability concerns of software systems. Architectural views or aspects might be a promising way forward for dealing with dependability concerns when providing the ability of a system to deliver the service that can be trusted, and obtaining confidence in this ability.

5 References

[1] Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1): 11-33

[2] Bass L, Clements P, Kazman R (1998) Software Architecture in Practice. Addison Wesley

[3] Boehm B (1996) Anchoring the Software Process. IEEE Software 13(4): 73-82

[4] Clements P, et al (2003) Documenting Software Architectures: Views and Beyond. Addison-Wesley

[5] Clements P, Kazman R, Klein M (2002) Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley

[6] Garlan D (2003) Formal Modeling and Analysis of Software Architectures. In: Bernado M, Inverardi P (eds). Formal Methods for Software Architectures.. Lecture Notes in Computer Science 2804. Springer. Berlin, Germany. pp 1-24

[7] Garlan D, Cheng SW, Schmerl B (2003) Increasing System Dependability through Architecture-based Self-Repair. In: de Lemos R, Gacek C, Romanovsky A (eds). Architecting Dependable Systems. Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. pp 61-89

[8] Oriezy P, et. al (1999) An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14(3): 54-62

[9] Perry DE, Wolf AL (1992) Foundations for the Study of Software Architectures. SIGSOFT Software Engineering Notes 17(4): 40-52

[10] Randell B (1975) System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering SE 1(2): 220-232

[11] Shaw M (1998) Moving from Qualities to Architecture: Architecture Styles. Software Architecture in Practice. L. Bass, P. Clements, R. Kazman (eds). Addison-Wesley. pp 93-122

[12] Shaw M, Garlan D (1996) Software Architectures: Perspectives on an Emerging Discipline. Prentice-Hall, Inc. Upper Saddle River, NJ