Architectural Description of Dependable Software Systems

Cristina Gacek

School of Computing Science
University of Newcastle upon Tyne

Rogério de Lemos

Computing Laboratory
University of Kent

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 [62].

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 rid 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.

In addition to the provision of facilities that enable the reasoning about faults at the architectural level, there are other issues that indirectly might influence the dependability of systems, and that should be observed for achieving effective structuring. These include understandability, compositionality, flexibility, refinement, traceability, evolution, and dynamism [17] (see chapter 1).

Architectural description languages (ADLs) are used within the software engineering community to support the description of high-level structure, or architecture, of software systems. A major advantage of this is the ability to analyze and evaluate trade-offs among alternative solutions. One can envision extending that definition to cover also non-technical aspects of a computer-based system's structure. However, there is one problem in trying to generalise the scope of this definition in those terms, mainly that software architectures are explicitly (formally) modelled in order to support reasoning about and analysis of the system under discussion. Modelling all possible ways in which humans that are a part of a system may interact is almost impossible, as is modelling their potential behaviour.

This chapter will discuss the role of ADLs for representing and analyzing the architecture of software systems. Since ADLs vary considerably on the modelling aspects that they cover, we will focus our discussion on how ADLs support structuring dependability issues. This discussion will be carried out from the perspective of the means to attain dependability. But first, we will provide a brief introduction to software architectures and ADLs.

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 [9]. 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 [13], it is recognized 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 [55][66]. The way a software architecture is configured defines how various connectors are used to mediate the interactions among components.

An architectural style imposes a set of constraints on the types of components and connectors that can be used and a pattern for their control and/or data transfers, thus restricting the set of configurations allowed. It simplifies descriptions and discussions by restricting the suitable vocabulary. A software architecture may conform to a single given style or to a mix of those.

During system development and evolution, a software architecture can be used for specifying its static and dynamic structure(s), for supporting analysis, and for guiding development, acting as a roadmap for designers and implementers. A software architecture is used throughout the software life-cycle to facilitate communication among the various stakeholders involved. Concepts such as conceptual integrity find their realm in the architectural models.

Software architectures can be thought of as high-level design or blue-print of a software system. They are derived from the various requirements and constraints imposed on the system, and later refined into lower level design and subsequently into an implementation. In Model Driven Architecture [53], a platform-independent software architecture is created (using an appropriate specification language) that is

then translated into one or more platform-specific ones that are used to guide implementation.

Architecture description languages (ADLs) aim at supporting 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.

Differing architectural styles focus on different system characteristics. Hence, the architectural style(s) that an ADL aims to support will establish the aspects that need to or can be expressed and limit the scope of valid descriptions. ADLs may also represent aspects that are not style specific (e.g., non-functional requirements). Furthermore, ADLs may concentrate only on the description of static aspects, but some also support the description of dynamic information about the architecture. Only a few ADLs explicitly support refinement, ensuring that higher level constraints are not violated at lower levels [52]. All these variations in ADLs imply that any specific ADL establishes the system features that it can describe, as well as what corresponding analysis can be performed. Although some good work exists that discusses different ADLs [16][50], there are not yet discussions and/or comparisons of ADLs with respect to dependability concerns.

Much discussion about supporting different architectural views exists [44]. The main point being that it would be beneficial to have diverse representations of systems for the purpose of supporting different types of analysis while avoiding information overload on a single view. Therefore the objective would be to have ADLs with multiple views, yet in reality that is not the case. ADLs still tend to focus on a single graphical and single textual description for supporting a particular type of analysis [50]. UML, as an ADL, can be said to support different views based on the different models included, but the relationships between these views are only enforced in terms of the entities represented and not on their semantics.

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. This is due to the fact that dependability concerns are usually left until too late in the process of development. In addition, the complexity of emerging applications and the trend of building trustworthy systems from existing, untrustworthy components are urging dependability concerns be considered at the architectural level.

In this section, we discuss the features that architectural description languages (ADLs) should possess for structuring dependable systems. System dependability is measured through its attributes, and there are several means for attaining these attributes, which can be grouped into four major categories [4]. *Rigorous design*, which aims at preventing the introduction or the occurrence of faults. *Verification and validation*, which aims at reducing the number or severity of faults. *Fault tolerance*,

which aims at delivering correct service despite the presence of faults. *System evaluation*, which aims at estimating the present number, the future incidence, and the likely consequences of faults. Since system structuring is relevant across all the dependability means, the ensued discussion will be partitioned in terms of these four categories.

Rigorous Design

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 faults) [4].

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 [28]. 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 reasoning 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 [17].

Although UML, and now UML 2.0 [12], has become the de facto standard in terms of notations for describing systems' design [17][31] (see chapter 1), there are several languages that could be more appropriate for representing the architecture of systems [50]. Nevertheless, industry still relies heavily on UML for obtaining models for their business, software architectures and designs, and also to obtain metamodels that allow defining dialects that are appropriate for describing their applications. However, UML offers a number of alternatives for representing architectures, and this lack of precision might lead to problems when obtaining a common understanding of an architectural model [17] [31].

Beyond the rigorous "box-and-line" notations (components and connectors view type) like UML, there are ADLs that have a formal underpinning that allow precise

descriptions and manipulations of the architectural structure, constraints, style, behaviour, and refinement [28]. In general, existing formal semantics for architectural notations can be divided into three categories [14]: graph, process algebra, and state. For example, in graph-based approaches, while a graph grammar is used to represent a style, a graph represents the actual architecture.

In principle, system structuring should enforce high cohesion and low coupling (see chapter 1), this can be supported by the use of an ADL or a specific architectural style (constraints on the types of architectural elements and their interaction [66]). However, coupling and cohesion are not necessarily features imposed or can be enforced by ADLs or by architectural styles. The C2 ADL [69], for example, fosters the reduction of coupling, yet its usage has no impact with respect to cohesion. SADL [52] fosters an increase in cohesion and has no impact in terms of coupling. Languages such as Wright [2] or Rapide [45] should have no impact with respect to either coupling or cohesion. In UML, as an ADL, coupling and cohesion characteristics would be system specific and not enforced by the language itself [31]. A similar phenomenon can be observed with respect to architectural styles. The blackboard style enforces low coupling of computational units, high coupling of the data centre, and high cohesion; the event-based style fosters low coupling; layered encourages high cohesion; pipe and filter enforces low coupling. The vast majority of the other architectural styles has no impact on the coupling or the cohesion of specific systems. Consequently, coupling and cohesion are mainly application (usage) specific and constrained to the process of structuring the system itself.

One of the major difficulties during software development is to guarantee that implementation conforms to its architectural representation. In case there are several representations between architecture and implementation, the process of relating representations is equally important to make sure that they are consistent. For instance, the implementation of architectural strategies that enforce security policies should guarantee that buffer overflows do not introduce vulnerabilities during system operation. In this direction there has been some work that relates, in a consistent way, dependability concerns from the early to late stages of software development by following the principles of Model Driven Architecture (MDA) [53][56]. The challenge is to define rules that transform an architectural model into code, guaranteeing at the same time that all the dependability properties are maintained [56][60].

One way of preventing configuration faults from occurring during system operation is to protect a component, or the context of that component, against potential mismatches that might exist between them, i.e., architectural mismatches [28] (design faults). These vulnerabilities can be prevented by adding to the structure of the system architectural solutions based on integrators (more commonly known as wrappers) [23]. The assumption here is that the integrators are aware of all incompatibilities that might exist between a component and its environment [61].

Verification and Validation

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 [4].

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. Example of techniques for the static analysis of architectural representations is 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, however depending of the ADL employed, it can also be employed for localizing any faults that might exist at the architectural description of a system [51].

Examples of architectural inspection techniques, like Architecture Tradeoff Analysis Method (ATAM) and Software Architecture Analysis Method (SAAM) [17], will be discussed under the section on system evaluation. In general, these techniques are based on questionnaires, checklists and scenarios to uncover faults that might exist on the architectural representation of the system.

When building systems from existing components, it is inevitable that architectural mismatches might occur [29]. The static analysis approaches associated with the removal of this type of design fault aim at localising architectural mismatches during the integration of arbitrary components. Existing approaches for identifying architectural mismatches are applied either during the composition of components while evaluating the architectural options [26], or during architectural modelling and analysis [25].

Model checking analyzes systems behaviour with respect to selected properties. Its algorithms offer an exhaustive and automatic approach to analyze completely the system. Model checking provides a simple and efficient verification approach, particularly useful in the early phases of the development process. For example, from an architectural description, a corresponding state-based model is extracted for checking its correctness against the desired properties. There are two major limitations associated with model checkers, they are limited to finite-state systems, and they suffer from state explosion. Model checking has been successfully applied in analyzing software architectures that are described using ADLs based on process algebras, the most prominent ones being Wright [1][2] and Darwin [46]. Wright, which is based on CSP, allows behavioural checks to be performed using the model checker FDR [62]. Darwin, which is based on Pi-calculus for describing structural aspects and FSP for describing the behavioural aspects, checks properties expressed in Linear Temporal Logic using LTSA [43]. Another approach, which uses UML for describing an architecture, relies on an automated procedure for mapping architectural elements into constructs of PROMELA, the modelling language for the SPIN model checker [37]. Most of the work so far has been on static structures, the challenge ahead lies on model checking ADLs that provide mobility and dynamicity mechanisms [48].

Simulation, as a dynamic analysis technique, is not widely supported by existing ADLs. Rapide is one of the few ADLs that allows the simulation and behavioural analysis of architectures [43]. Rapide is an event based concurrent language, designed for prototyping architectures of distributed systems. It provides event pattern mappings, an approach for defining relationships between architectures, to define how a system is related to a reference architecture.

The role of software architecture in testing is evident since an architectural description is a high-level design blueprint of the system that can be used as a reference model to generate test cases. Moreover, the testability of a system is related to its architecture. In addition to the documentation that architectures provide from which specification-based integration and system testing can be obtained, there are other features in an architectural description that promote testability: modularisation, encapsulation of components for information hiding, and separation of concerns. In software architecture based testing, the dynamic description of the architecture can be useful in the systematic derivation of test cases to be executed on the implemented system. In case the architectural description is a formal model, the synthesis of the test cases can be automated for architectural based conformance testing [6].

The role of architectural representation in the removal of faults after system deployment includes both corrective and preventative maintenance [4]. The software architecture, in terms of components and connectors, provides a good starting point for revealing the areas a prospective change will affect [17]. For example, an architecture might define the components and their responsibilities, and the circumstances under which each component has to change.

Fault Tolerance

Fault tolerance aims to avoid system failure via error detection and system recovery [4]. Error detection at the architectural level relies on monitoring mechanisms, or probes [8], for observing the system states for detecting erroneous states 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 [57]. 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.

Architectural monitoring consists of collecting information from the system execution, analysing it and detecting particular events or states. However, there is an inherent gap between the architectural level and the information that is actually collected, and mapping solution is necessary for integrating the primitive events and the architectural (high) level composed events. Without incurring into a large volume of data, or limiting the analysis by collecting interesting events only, monitoring solutions should be based on languages that are able to define events independently of the system implementation, the purpose of the analysis, and the monitoring system [24].

A key issue in dependability is error confinement, which is the ability of a system to contain errors (see chapter 1). The role of ADLs in error confinement needs to be approached from two distinct angles. On 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. Yet, since components and connectors do not exist on their own within systems, but as parts of a configuration of components and connectors, it is easier to include error confinement mechanisms within components and their ports rather than in arbitrary connectors. Some examples of mechanisms for error confinement at the level of interactions between components are coordinated atomic actions (CA actions) and atomic transactions [72], co-operative connectors [21], and monitored environments [74]. In particular, when dealing with components of the shelf (COTS) error confinement mechanisms might not have been originally included and are not easily added on. In these cases, error confinement mechanisms can be included in ‘smart connectors’ [7] or wrappers [36].

Some ADLs lend themselves to easily add (some) explicit error confinement checks, as in the work using the language C2 [69] that explicitly adds exception handling to specific software architectures, providing a coordinated (controlled) propagation of errors [35]. However, some languages might not offer the facilities to include checks that are fundamental for error confinement. Nevertheless, if there is a system description using an ADL, it helps highlight the connection points in the system where error confinement is relevant, and what behavioural variables (variables that describe the behaviour associated with a component interface) to check.

Nevertheless, although some ADLs can be used to represent error confinement mechanisms, they do not yet provide embedded means for error confinement analysis, nor do they automatically include error confinement mechanisms into structures that they describe. The introduction of error confinement mechanisms in architectural structures and error confinement analysis must both be explicitly performed by the software architects on a case by case basis.

For error handling during system recovery, exception handling has shown to be an effective mechanism if properly incorporated into the structure of the system. Such an architectural solution for structuring software architectures compliant with the C2 architectural style [69] is the idealised C2 component (iC2C) [35]. This architectural solution is based on the idealised fault-tolerant component concept [3], which provides a means for system structuring which makes it easy to identify *what* parts of a system have *what* responsibilities for trying to cope with *which* sorts of fault. This approach was later extended to deal with commercial off-the-shelf (COTS) software components [36]. A more general strategy for exception handling for the development of component-based dependable systems is based on the integration of two complementary strategies, a global exception handling strategy for inter-component composition, and a local exception handling strategy for dealing with errors in reusable components [15]. Another means to obtain error recovery is to enforce transaction processing either based on backward or forward error recovery. In the particular context of dependable composition of Web services, one solution lies in structuring the system using Web Services Composition Actions (WSCA) [68].

Outside the context of fault tolerance, compensation has been used to ensure dependable system composition and evolution when upgrading components, by employing an approach that makes use of diversity between old and new versions of components. While the core idea of the Hercules framework [19] is derived from concepts associated with recovery blocks [59], the notion of multi-versioning connectors (MVC) [57], in the context of architectures compliant with the C2 architectural style [69], is derived from concepts associated with N-version programming [4].

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 [54]. 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 [32]. 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 [20][54]. Exception handling can be useful when dealing with configuration exceptions, which are exceptional events that have to be handled at the configuration level of architectures [38].

System Evaluation

System evaluation, also known as fault forecasting, is conducted by evaluating systems' behaviour with respect to fault occurrence or activation [4]. 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 [18]. 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 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 [18].

The Architecture Tradeoff Analysis Method (ATAM) is a method for architectural evaluation that reveals how well an architecture satisfies particular quality goals, and provides insight into how those quality goals interact with each other [41]. It provides a way to articulate desired quality attributes and to expose the architectural decisions relevant to those attributes. For that, it uses questioning techniques that are based on scenarios, and template questions related to the architectural style being used and the attribute under analysis [18]. For guiding the process of architectural evaluation, a specialised architectural style called attribute-based architectural style (ABAS) is particularly useful in ATAM. Together with the style, there is an explanation on how quality attributes are achieved, and this explanation provides a basis for attribute-specific questions associated with the style. An example of a domain specific ABAS was the definition of a specialised ABAS that facilitates the automated dependability analysis of software architectures [33].

Another example of an architectural evaluation method is Software Architecture Analysis Method (SAAM), which is useful to assess quality attributes, such as modifiability, as well as functional coverage [40]. Based on a description of the architecture, system stakeholders enumerate scenarios that represent known and likely system's changes. In this context, a scenario is a short statement describing an interaction of a stakeholder with the system. As an outcome of the evaluation process, stakeholders gain more in-depth understanding of the architecture, and can compare two or more candidate architectures [18].

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.

In terms of modelling approaches, instead of manipulating stochastic models at the architectural level, several approaches have used standardised design notations for representing architectures, like the Unified Modeling Language (UML) [37][46] and the Specification and Description Language (SDL) [33]. From these representations, stochastic models can be generated automatically from the attributes embedded in the architectural descriptions, which were created from the augmented standard notations. Quantitative architectural evaluation can then be performed on these stochastic models, which can be based on several different formalisms: Markov Chains [34], Stochastic Reward Nets (SRN) [33], Timed Petri Nets (TPN) [46], or state space

models [37]. For the above approaches, it is assumed that there is complete knowledge of the parameters that characterise the failure behaviours of the components and connectors of the system, however at the architectural level the knowledge about the system operational profile might be partial. An alternative approach is to use Hidden Markov Models for coping with this imperfect knowledge [62].

A more radical approach in performance evaluation was the proposal of Æmilia, a performance-oriented ADL [10]. Æmilia combines a process-algebra-based ADL that incorporates architectural checks for deadlocks, and a stochastic process algebra that allows functional and performance evaluation of concurrent and distributed systems. In a later work, Æmilia has been combined with queuing networks for obtaining quick predictions when comparing the performance of different software architectures [6].

In terms of testing approaches for probabilities estimation, fault injection techniques have been proposed for evaluating the dependability of the system [51]. Software architecture provides the basis for planning the analysis early in the development process, since dependencies can be established before the source code is available. Relationships among components establish these dependencies based on the interactions through their provided and required interfaces. This can be helpful for fault injection because it determines the components that are worth injecting into.

Also as part of fault forecasting is the analysis of service degradation. Architectural representation of systems plays a major role for measuring the degradation of services in the presence of multiple component faults [67].

In summary, if architectural decisions determine the dependability attributes of a system, then it should be possible to evaluate these architectural decisions in terms of their impact [18]. Architectural evaluation of a software system is a wise risk-mitigation effort and is relatively inexpensive, comparing with the costs of fixing a system late in the development process. However, existing ADLs lack the support to specify quality attributes both at the component and connector level [50]. One of the few exceptions is MetaH [71], which allows the representation of attributes needed for real-time schedulability, reliability and security analysis.

Tool Support

Tool support exists for many ADLs, for example for C2, Rapide, xADL, and UML. All of them support the description of a software system in that language and some provide means for analysis, such as checking for deadlocks or even simulating the execution of the architecture.

Unfortunately, no tool supports the analysis of a very comprehensive set of characteristics. They all focus on at most a few of those, and more frequently solely on the correctness of description with respect to the language. This means that to get the required coverage analysis, one needs to describe the same system in more than one language.

The ACME ADL has been developed in an effort to help leverage from tool support offered by the various tool suites [30][64]. It is an interchange language that has very few elements of its own, such as components and connectors, with close to no attributes. It permits the addition of language specific attributes just by tagged

fields. It attaches no semantics to the various elements/attributes and aims at transforming descriptions from one language into another. This transformation process requires human intervention on adding the attributes that are specific to the target language. The major drawback with ACME is its lack of semantics and the fact that there is no embedded support for checking the consistency among attributes that are specific to different languages (that is not a problem when the attributes are truly independent, yet that is not always the case).

Conclusions

The architecture of a software system is an abstraction of its structuring, and structuring is fundamental when developing dependable systems (see chapter 1). Architectural Description Languages (ADLs) provide the notation for representing software architectures that support effective structuring in terms of error confinement, coupling, cohesion and flexibility. Moreover, ADLs promote, among other properties, modularisation by structuring a system in terms of components, connectors and configurations, information hiding by restricting information access only through the interfaces of components and connectors, and strongly typed by using architectural styles.

From the perspective of dependability, effective structuring should aim to build fault-free systems (fault avoidance) and systems that cope with faults (fault acceptance) [5]. 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 analyze 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.

References

1. R. J. Allen. *A Formal Approach to Software Architecture*. Technical Report CMU-CS-97-144. Carnegie Mellon University. May 1997.
2. R. Allen, D. Garlan. “A Formal Basis for Architectural Connection”. *ACM Transactions on Software Engineering and Methodology 6(3)*. July 1997. pp. 213-249.
3. T. Anderson, P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 1981.
4. A. Avizienis. “The N-Version Approach to Fault-tolerant Software”. *IEEE Transactions on Software Engineering 11(2)*. December 1995. pp. 1491-1501.

5. A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr. “Basic Concepts and Taxonomy of Dependable and Secure Computing”. *IEEE Transactions on Dependable and Secure Computing 1(1)*. January-March 2004. pp. 11-33.
6. S. Balsamo, M. Bernado, M. Simeoni. “Performance Evaluation at the Architecture Level”. *Formal Methods for Software Architectures*. M. Bernado, P. Inverardi (Eds.). Lecture Notes in Computer Science 2804. Springer. Berlin, Germany. 2003. pp. 207-258.
7. R. Balzer. *Instrumenting, Monitoring, and Debugging Software Architectures*. http://www.isi.edu/divisions/index.html. 1999.
8. R. Balzer. *The DASADA Probe Infrastructure*. Internal Report. Teknowledge Corporation. USA.
9. L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley, 1998.
10. M. Bernado, L. Donatiello, P. Ciancarini. “Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language”. *Performance Evaluation of Complex Systems: Techniques and Tools*. Lecture Notes in Computer Science 2459. Springer. Berlin, Germany. 2002. pp. 236-260.
11. A. Bertolino, P. Inverardi, H. Muccini. “Formal Methods in Testing Software Architecture”. *Formal Methods for Software Architectures*. M. Bernado, P. Inverardi (Eds.). Lecture Notes in Computer Science 2804. Springer. Berlin, Germany. 2003. pp. 122-147.
12. M. Björkander, C. Kobryn. “Architecting Systems with UML 2.0”. *IEEE Software*. July/August 2003. pp. 57-61.
13. B. Boehm. “Anchoring the Software Process”. *IEEE Software 13(4)*. July 1996. pp. 73-82.
14. J. S. Bradbury, J. R. Cordy, J. Dingel, M. Wermelinger. “A Survey of Self-Management in Dynamic Software Architecture Specifications”. *Proceedings of the 2004 ACM SIGSOFT Workshop On Self-Managed Systems (WOSS’04)*. October/November 2004. Newport Beach, CA, USA.
15. F. Castor Filho, P. A. de C. Guerra, V. A. Pagano, C. M. F. Rubira. “A Systematic Approach for Structuring Exception Handling in Robust Component-Based Software”. *Journal of the Brazilian Computer Society 3(10)*. April 2005.
16. P. Clements. “A Survey of Architecture Description Languages”. *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD’96)*. Paderborn, Germany. 1996. pp. 16-25.
17. P. Clements, et al. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley. 2003.
18. P. Clements, R. Kazman, M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley. 2002.
19. J. E. Cook, J. A. Dage. “Highly Reliable Upgrading of Components”. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. Los Angeles, CA, USA. May 1999. ACM Press. pp. 203-212.
20. E. Dashofy, A. van der Hoek, R. N. Taylor. “Towards Architecture-Based Self-Healing Systems”. *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*. Charleston, SC, USA. November 2002. pp. 21-26.
21. R. de Lemos. “Analysing Failure Behaviours in Component Interaction”. *Journal of Systems and Software 71(1-2)*. April 2004. pp. 97-115.
22. R. de Lemos, C. Gacek, A. Romanovsky. “Architectural Mismatch Tolerance”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 175-196.
23. R. DeLine. “A Catalog of Techniques for Resolving Packaging Mismatch”. *Proceedings of the 5th Symposium on Software Reusability (SSR'99)*. Los Angeles, CA. May 1999. pp. 44-53.

24. M. Dias, D. Richardson. “The Role of Event Description in Architecting Dependable Systems”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 150-174.
25. A. Egyed, N. Medvidovic, C. Gacek. “Component-Based Perspective on Software Mismatch Detection and Resolution”. *IEE Proceedings on Software 147(6)*. December 2000. pp. 225-236.
26. C. Gacek. *Detecting Architectural Mismatches during System Composition*. PhD Dissertation. Center for Software Engineering. University of Southern California. Los Angeles, CA, USA. 1998.
27. C. Gacek, A. Abd-Allah, B. Clark, and B. Boehm. “On the Definition of Software Architecture”. *Proceedings of the First International Workshop on Architectures for Software Systems (ISAW)*. Seattle, WA. April 1995. pp. 85-95.
28. D. Garlan. “Formal Modeling and Analysis of Software Architectures”. *Formal Methods for Software Architectures*. M. Bernado, P. Inverardi (Eds.). Lecture Notes in Computer Science 2804. Springer. Berlin, Germany. 2003. pp. 1-24.
29. D. Garlan, R. Allen, J. Ockerbloom. “Architectural Mismatch: Why Reuse is so Hard”. *IEEE Software 12(6)*. November 1995. pp. 17-26.
30. D. Garlan, R. T. Monroe, D. Wile. “Acme: Architectural Description of Component-Based Systems”. *Foundations of Component-Based Systems*. G.T. Leavens, M. Sitaraman (Eds). Cambridge University Press. 2000.
31. D. Garlan, S.-W. Cheng, A. J. Kompanek. “Reconciling the Needs of Architectural Description with Object-Modeling Notations”. *Science of Computer Programming Journal* 44. Elsevier Press. 2001. pp. 23-49.
32. D. Garlan, S.-W. Cheng, B. Schmerl. “Increasing System Dependability through Architecture-based Self-Repair”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 61-89.
33. S. S. Gokhale, J. R. Horgan, K. Trivedi. “Specification-Level Integration of Simulation and Dependability Analysis”. ”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 245-266.
34. V. Grassi. “Architecture-based Reliability Prediction for Service-oriented Computing”. *Architecting Dependable Systems III*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 3549. Springer. Berlin, Germany. 2005. pp. 291-309.
35. P. A. de C. Guerra, C. Rubira, R. de Lemos. “A Fault-Tolerant Software Architecture for Component-Based Systems”. *Architecting Dependable Systems*. Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 129-149.
36. P. A. de C. Guerra, C. Rubira, A. Romanovsky, R. de Lemos. “A Dependable Architecture for COTS-based Software Systems using Protective Wrappers”. *Architecting Dependable Systems II*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 3069. Springer. Berlin, Germany. 2004. pp. 144-166.
37. G. J. Holzmann. “The SPIN Model Checker”. *IEEE Transactions on Software Engineering 23*. 1997. pp. 279–295.
38. V. Issarny, J.-P. Banatre. “Architecture-based Exception Handling”. *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS'34)*. IEEE, 2001.
39. V. Issarny, A. Zarras. “Software Architecture and Dependability”. *Formal Methods for Software Architectures.* M. Bernado, P. Inverardi (Eds.). Lecture Notes in Computer Science 2804. Springer. Berlin, Germany. 2003. pp. 259-285.

40. R. Kazman, G. Abowd, L. Bass, M. Webb. “SAAM: A Method for Analyzing the Properties of Software Architectures”. *Proceedings of the 16th International Conference on Software Engineering (ICSE 1990)*. Sorrento, Italy. May 1994. pp. 81-90.
41. R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, J. Carriere. “The Architecture Tradeoff Analysis Method”. *Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS98)*. August 1998.
42. M. Klein, R. Kazman, L. Bass, S. J. Carriere, M. Barbacci, H. Lipson. “Attribute-based Architectural Styles”. *Proceedings of the 1st IFIP Working Conference on Software Architecture (WICSA-1)*. 1999. pp. 225-243.
43. J. Kramer, J. Magee, S. Uchitel. “Software Architecture Modeling & Analysis: A Rigorous Approach”. *Proceedings of SFM’2003*. Lecture Notes in Computer Science 2804. 2003. pp. 44–51.
44. P. Kruchten. “The 4+1 View Model of Architecture”. *IEEE Software 12(6)*. November 1995. pp. 42-50.
45. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann. “Specification and Analysis of System Architecture Using Rapide”. *IEEE Transactions on Software Engineering 21(4)*. April 1995, pp. 336-355.
46. J. Magee, N. Dulay, S. Eisenbach, J. Kramer. Specifying Distributed Software Architectures. *Proceedings of the Fifth European Software Engineering Conference (ESEC’95)*. Lecture Notes in Computer Science 989. Barcelona, Spain. September 1995. pp. 137–153.
47. I. Majzik, A. Pataricza, A. Bondavalli. “Stochastic Dependability Analysis of UML Designs”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 219-244.
48. R. Mateescu. “Model Checking for Software Architectures”. *Proceedings of the 1st European Workshop on Software Architecture (EWSA'2004)*. Lecture Notes in Computer Science 3047. St Andrews, Scotland. Springer. Berlin, Germany. May 2004. pp. 219-224.
49. N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, J. E. Robbins. “Modeling Software Architectures in the Unified Modeling Language”. *ACM Transactions on Software Engineering and Methodology 11(1)*. January 2002. pp. 2-57.
50. N. Medvidovic, R. N. Taylor. “A Classification and Comparison Framework for Software Architecture Description Languages”. *IEEE Transactions on Software Engineering 26(1)*. 2000. pp. 70-93.
51. R. L. de O. Moraes, E.Martins. “Fault Injection Approach based on Architectural Dependencies”. *Architecting Dependable Systems III*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 3549. Springer. Berlin, Germany. 2005. pp. 310-334.
52. M. Moriconi, X. Qian, R. A. Riemenschneider. “Correct Architecture Refinement”. *IEEE Transactions on Software Engineering, 21(4)*. 1995. pp. 356-372.
53. Object Management Group. Model Driven Architecture. Technical Report. July 2001. http://cgi.omg.org/docs/ormsc/01-07-01.pdf.
54. P. Oriezy, et. al. “An Architecture-Based Approach to Self-Adaptive Software”. *IEEE Intelligent Systems 14(3)*. May/June 1999. pp. 54-62.
55. D.E. Perry, A.L. Wolf. “Foundations for the Study of Software Architectures”. *SIGSOFT Software Engineering Notes 17(4)*. October 1992. pp. 40-52.
56. C. Raistrick, T. Bloomfield. “Model Driven Architecture – an Industry Perspective”. *Architecting Dependable Systems II*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 3069. Springer. Berlin, Germany. 2004. pp. 341-362.
57. M. Rakic, N. Medvidovic. “Increasing the Confidence in On-the-Shelf Components: A Software Connector-Based Approach”. *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*. May 2001. pp. 11-18.

58. B. Randell. “System Structure for Software Fault Tolerance”. *IEEE Transactions on Software Engineering 1(2)*. June 1975. pp. 220-232.
59. B. Randell, J. Xu. “The Evolution of the Recovery Block Concept”. *Software Fault Tolerance*. John Wiley Sons Ltd., 1995.
60. G. N. Rodrigues, G. Roberts, W. Emmerich. “Reliability Support for the Model Driven Architecture”. *Architecting Dependable Systems II*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 3069. Springer. Berlin, Germany. 2004. pp. 80-100.
61. M. Rodriguez, J.-C. Fabre, J. Arlat. “Wrapping Real-Time Systems from Temporal Logic Specification”. *Proceedings of the European Dependable Computing Conference (EDCC-4)*. Toulouse, France. October 2002 pp. 253-270.
62. A.W. Roscoe. “Model-Checking CSP”. *A Classical Mind: Essays in Honor of C.A.R. Hoare*. Prentice-Hall, 1994.
63. R. Roshandel, N. Medvidovic “Toward architecture-based reliability prediction”. *Proceedings of the ICSE 2004 Workshop on Architecting Dependable Systems (WADS 2004)*. Edinburgh, Scotland, UK. The IEE. London, UK. May 2004. pp. 2-6.
64. B. Schmerl, D. Garlan. “AcmeStudio: Supporting Style-Centered Architecture Development”. *Proceedings of the 26th International Conference on Software Engineering*. Edinburgh, Scotland, UK. May 2004. pp. 704-705.
65. M. Shaw. “Moving from Qualities to Architecture: Architecture Styles”. *Software Architecture in Practice*. L. Bass, P. Clements, R. Kazman (Eds.). Addison-Wesley. 1998. pp. 93-122.
66. M. Shaw, D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ. 1996.
67. C. Shelton, P. Koopman. “Using Architectural Properties to Model and Measure Graceful Degradation”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 267-289.
68. F. Tartanoglu, V. Issarny, A. Romanovsky, N. Levy. “Dependability in the Web Services Architectures”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 90-109.
69. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow. “A Component- and Message-based Architectural Style for GUI Software”. *IEEE Transactions on Software Engineering 22(6)*. June 1996. pp. 390-406.
70. S. Vestal. *A Cursory Overview and Comparison of Four Architecture Description Languages*. Technical Report. Honeywell Technology Center. February 1993.
71. S. Vestal. *MetaH Programmer’s Manual, Version 1.09*. Technical Report. Honeywell Technology Center. April 1996.
72. J. Xu, et al. “Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery”. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*. Pasadena, CA, USA. 1995. pp. 499-509.
73. A. Zarras, C. Kloukinas, V. Issarny. “Quality Analysis for Dependable Systems: A Developer Oriented Approach”. *Architecting Dependable Systems*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 2677. Springer. Berlin, Germany. 2003. pp. 197-218.
74. J. Zhang, B. H.C. Cheng, Z. Yang, P. K. McKinley. “Enabling Safe Dynamic Component-Based Software Adaptation”. *Architecting Dependable Systems III*. R. de Lemos, C. Gacek, A. Romanovsky (Eds.). Lecture Notes in Computer Science 3549. Springer. Berlin, Germany. 2005. pp. 200-219.