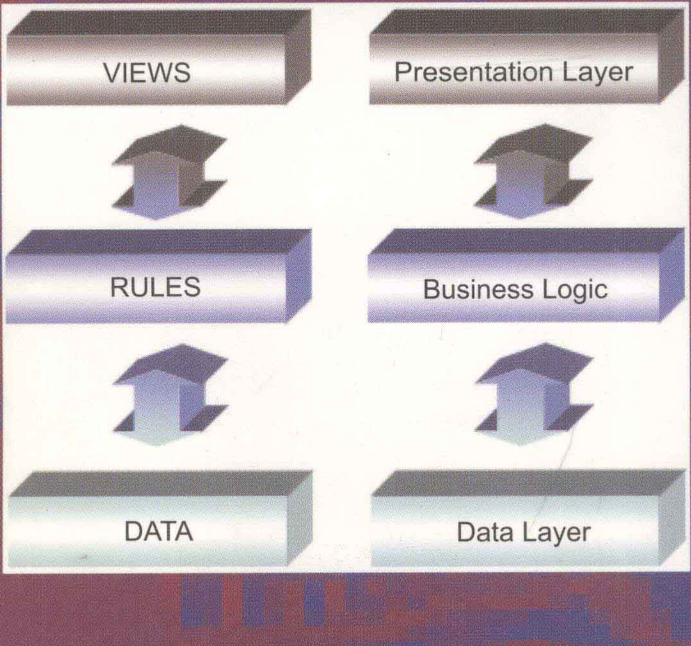
State-of-the-Art Survey

LNCS 4615

Rogério de Lemos
Cristina Gacek
Alexander Romanovsky (Eds.)

Architecting Dependable Systems IV



Springer

Rogério de Lemos
Cristina Gacek
Alexander Romanovsky (Eds.)

Architecting Dependable Systems IV

Springer

Volume Editors

Rogério de Lemos
University of Kent, Computing Laboratory
Canterbury, Kent CT2 7NF, UK
E-mail: r.delemos@kent.ac.uk

Cristina Gacek
Alexander Romanovsky
Newcastle University, School of Computing Science
Newcastle upon Tyne, NE1 7RU, UK
E-mail: {cristina.gacek, alexander.romanovsky}@ncl.ac.uk

Library of Congress Control Number: 2007931900

CR Subject Classification (1998): D.2, D.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-74033-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-74033-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12102341 06/3180 5 4 3 2 1 0

Foreword

On a recent visit to Sweden I had the pleasure of traveling by train between Stockholm and Malmö over several segments that spanned a few days. The trains always ran on time and were very comfortable. Particularly convenient was the fact that a passenger could get on the Internet during the trip simply by using her ticket number as the access code. One of the features on the on-line provider's home page was a map of that area of Sweden, with the train's current location updated in real-time. Impressed by this, I made a point of mentioning it to my Swedish host, and the conversation quickly turned to how much today's systems, such as my train, rely on and are controlled by software.

My host subsequently relayed a somewhat less pleasant experience with the same type of train on which I had just arrived. During one of his recent trips, the software controlling the angle at which one of the train's cars entered and exited curves was not functioning properly. As a result, the G-force experienced by the passengers during turns had almost doubled. The problem was fixed at the next station, where the train sat idle for some time while it literally rebooted. I found myself having two reactions to this story. As a traveler, my first thought was that it is a good thing we do not have to reboot airplanes in mid-flight. As a software engineer, I wondered exactly how the software was constructed and what caused this particular problem.

As this story illustrates, as "regular" people we constantly depend on software in our daily lives, yet frequently do not realize it and rarely, if ever, stop to analyze the implications of that dependence and the extent of the software's actual *dependability*. On the other hand, as software engineering professionals, we are not only becoming increasingly aware of the importance of software dependability, but have amassed an arsenal of techniques and tools to help us ensure it. Many of these techniques and tools have traditionally been used to ensure dependability in existing systems "after the fact," that is, after the system has been designed, and possibly implemented and even deployed. However, a new class of emerging techniques gives dependability first-class status in the development of software-intensive systems by integrating dependability into software engineering processes from their inception. These techniques rely on a software system's architecture as the principal driver of dependability.

This book is the fourth in a series of collected papers on software architecture-based dependability solutions. The book addresses a number of on-going challenges (such as system modeling and analysis for dependability and ensuring dependability in distributed systems) as well as some timely issues (such as the role of the Architecture Analysis and Design Language—AADL—standard in modeling dependable systems, architecture-driven dependability in the automotive domain, and the benefits of following the model-driven architecture paradigm in ensuring software dependability). This book joins its three companion volumes in forming an indispensable source for the fast-growing community of software researchers and practitioners who are confronting the challenges posed by this important topic and architecting the software systems on which we rely every day.

Nenad Medvidovic
University of Southern California

Preface

This is the fourth book in a series on Architecting Dependable Systems we started five years ago that brings together issues related to software architectures and the dependability of systems. This book includes expanded and peer-reviewed papers based on the selected contributions to the Workshop on Architecting Dependable Systems (WADS), organized at the 2006 International Conference on Dependable Systems and Networks (DSN 2006), and a number of invited papers written by recognized experts in the area.

Identification of the system structure (i.e., architecture) early in its development process makes it easier for the developers to make crucial decisions about system properties and to justify them before moving to the design or implementation stages. Moreover, the architectural level views support abstracting away from details of the system, thus facilitating the understanding of broader system concerns. One of the benefits of a well-structured system is the reduction of its overall complexity, which in turn leads to a more dependable system that typically has fewer remaining faults and is capable of dealing with errors and faults of different types in a well-defined, cost-effective and disciplined way.

System dependability is defined as the reliance that can be justifiably placed on the service delivered by the system. It has become an essential aspect of computer systems as everyday life increasingly depends on software. It is therefore a matter for concern that dependability issues are usually left until too late in the process of system development.

Making decisions and reasoning about structure happen at different levels of abstraction throughout the software development cycle. Reasoning about dependability at the architectural level has recently been in the focus of researchers and practitioners because of the complexity of emerging applications. From the perspective of software engineering, traditionally striving to build software systems that are fault-free, architectural consideration of dependability requires the acceptance of the fact that system models need to reflect that it is impossible to avoid or foresee all faults. This requires novel notations, methods and techniques providing the necessary support for reasoning about faults (including fault avoidance, fault tolerance, fault removal and fault forecasting) at the architectural level.

This book comes as a result of bringing together research communities of software architectures and dependability, and addresses issues that are currently relevant to improving the state of the art in architecting dependable systems. The book consists of four parts: Architectural Description Languages, Architectural Components and Patterns, Architecting Distributed Systems, and Architectural Assurances for Dependability.

The first part entitled "Architectural Description Languages" (ADLs) includes four papers focusing on various aspects of defining and using ADLs with an aim to ensure system dependability. The first paper of this part, "Architecting Dependable Systems with the SAE Architecture Analysis and Description Language (AADL)," is prepared by J. Tokar. The Avionics Systems Division of the Society of Automotive Engineers (SAE) has recently adopted this language to support incorporation of formal methods and engineering models into analysis of software and system architectures. The SAE AADL is a standard that has been specifically developed for embedded real-time safety critical systems. It supports the use of various formal approaches to analyzing the impact of system composition from hardware and software components and allows the generation of system glue code with the performance qualities predicted. The paper highlights features of AADL that facilitate the development of system architectures and demonstrates how the features can be used to conduct a wide variety of dependability analysis of the AADL architectural models. To help in the understanding of AADL, the paper begins with a discussion of software and systems architecture and then shows how the AADL supports these concepts.

The second paper, written by A.-E. Rugina, K. Kanoun and M. Kaâniche and entitled "A System Dependability Modeling Framework using AADL and GSPNs," describes a modeling framework that generates dependability-oriented analytical models from Architecture Analysis and Design Language (AADL) specifications, which are then used for evaluating dependability measures, such as reliability or availability. The proposed stepwise approach transforms an AADL dependability model into a Generalized Stochastic Petri Net (GSPN) by applying model transformation rules that can be automated and then processed by existing tools.

P. Cuenot, D. Chen, S. Gérard, H. Lönn, M.-O. Reiser, D. Servat, R. T. Kolagari, M. Törngren and M. Weber contribute to the book with the paper "Towards Improving Dependability of Automotive Systems by Using the EAST-ADL Architecture Description Language." Management of engineering information is critical for developing modern embedded automotive systems. Development time, cost efficiency, quality and dependability all benefit from appropriate information management. System modeling based on an architecture description language is a way to keep this information in one information structure. EAST-ADL is an architecture description language for automotive embedded systems. It is currently refined in the ATESST project. The focus of this paper is on describing how dependability is addressed in the EAST-ADL. The engineering process defined in the EASIS project is used as an example illustrating support for engineering processes in EAST-ADL.

The final paper of the first part is "The View Glue" written by A. Radjenovic and R. Paige. It focuses on domain-specific architecture description languages (ADLs), particularly for safety critical systems. In this paper, the authors outline the requirements for safety critical ADLs, the challenges faced in their construction, and present an example – AIM – developed in collaboration with the safety industry. Explaining the key principles of AIM, the authors show how to address multiple and cross-cutting concerns through active system views and how to ensure consistency across such views. The AIM philosophy is supported by a brief exploration of a real-life jet engine case study.

The second part of this book is entitled "Architectural Components and Patterns" and contains five papers. In the first paper, entitled "A Component-Based Approach to Verification and Validation of Formal Software Models," D. Desovski and B. Cukic present a methodology for the automated decomposition and abstraction of Software Cost Reduction (SCR) specifications. The approach enables one to identify components in an SCR specification, perform the verification component by component, and apply compositional verification methods. It is shown that the algorithms can be used in large specifications.

In the paper "A Pattern-Based Approach for Modeling and Analyzing Error Recovery," A. Ebnenasir and B. H. C. Cheng present an object analysis pattern, called the corrector pattern, that provides a generic reusable strategy for modeling error recovery requirements in the presence of faults. In addition to templates for constructing structural and behavioral models of recovery requirements, the corrector pattern also contains templates for specifying properties that can be formally verified to ensure the consistency between recovery and functional requirements. Additional property templates can be instantiated and verified to ensure the fault-tolerance of the system to which the corrector pattern has been applied. This analysis method is validated in terms of UML diagrams and demonstrated in the context of an industrial automotive application.

The third paper of this part, "Architectural Fault Tolerance Using Exception Handling," is written by R. de Lemos. This paper presents an architectural abstraction based on exception handling for structuring fault-tolerant software systems. The proposed architectural abstraction transforms untrusted software components into idealized fault-tolerant architectural elements (iFTE), which clearly separate the normal and exceptional behaviors, in terms of their internal structure and interfaces. The feasibility of the proposed approach is evaluated in terms of a simple case study.

R. Buskens and O. Gonzalez contribute to the book with the paper "Model-Centric Development of Highly Available Software Systems." They present the Aurora Management Workbench (AMW) as a solution to the problem of integration a high availability (HA) middleware with the system that uses it. AMW is an HA middleware and a set of tools for building highly available distributed software systems. It is unique in its approach to developing highly available systems: developers focus only on describing key architectural abstractions of their system as well as system HA needs in the form of a model. Tools then use the model to generate much of the code needed to integrate the system with the AMW HA middleware, which also uses the model to coordinate and control HA services at run-time. The paper discusses initial successes using the approach proposed in developing commercial telecom systems.

The final paper of this part, written by L. Grunske, P. Lindsay, E. Bondarev, Y. Papadopoulos and D. Parker and entitled "An Outline of an Architecture-Based Method for Optimizing Dependability Attributes of Software-Intensive Systems," provides an overview of 14 different approaches for optimizing the architectural design of systems with regard to dependability attributes and cost. As a result of this study, the authors present a meta-method that specifies the process of designing and optimizing architectures with contradicting requirements on multiple quality attributes.

Part three of the book is on "Architecting Distributed Systems" and includes six papers focusing on approaches to architectural level reasoning about dependability concerns of distributed systems. This part starts with a paper by P. Inverardi and L. Mostarda that is entitled "A Distributed Monitoring System for Enhancing Security and Dependability at an Architectural Level." The paper presents the DESERT tool that allows the automatic generation of distributed monitoring systems for enhancing security and dependability of a component-based application at the architectural level. The DESERT language permits one to specify both the component interfaces and interaction properties in terms of correct component communications. DESERT uses these specifications to generate one filter for each component. Each filter locally detects when its component communications violate the property and can undertake a set of reaction policies.

In their paper, entitled "Architecting Dynamic Reconfiguration in Dependable Systems," A. T. A. Gomes, T. V. Batista, A. Joolia and G. Coulson introduce a generic approach to supporting dynamic reconfiguration in dependable systems. The proposed approach is built on the authors' view that dynamic reconfiguration in such systems needs to be causally connected at runtime to a corresponding high-level software architecture specification. More specifically, two causally connected models are defined, an architecture-level model and a runtime-level model. Dynamic reconfiguration is applied either through an architecture specification at the architectural level, or through reconfiguration primitives at the runtime level. This approach supports both foreseen and unforeseen reconfigurations—these are handled at both levels with a well-defined mapping between them.

T. Dumitraş, D. Roşu, A. Dan and P. Narasimhan, in their paper "Ecotopia: An Ecological Framework for Change Management in Distributed Systems," present Ecotopia, a framework for change management in complex service-oriented architectures (SOA) that is ecological in its intent: it schedules change operations with the goal of minimizing the service-delivery disruptions by accounting for their impact on the SOA environment. Ecotopia handles both external change requests, such as software upgrades, and internal changes requests, such as fault-recovery actions. The authors evaluate the Ecotopia framework using two realistic change-management scenarios in distributed enterprise systems.

In the fourth paper, entitled "Generic-Events Architecture: Integrating Real-World Aspects in Event-Based Systems," A. Casimiro, J. Kaiser, and P. Veríssimo describe an architectural solution consisting of an object model environment, which can be easily composed, representing software/hardware entities capable of interacting with the environment, and an event model that allows one to integrate real-world events and events generated in the system. The architectural solution and the event-model permit one to compose large applications from basic components, following a hierarchical composition approach.

The fifth paper is by C. Heller, J. Schalk, S. Schneele, M. Sorea, and S. Voss and is entitled "Flexible Communication Architecture for Dependable Time-Triggered Systems." The authors propose an approach expressed in terms of a dependable and flexible communication architecture that supports flexibility in the use of time-triggered technologies and delivers a highly effective, reliable and dependable system design. This work is undertaken in the context of safety-critical aerospace applications.

The final paper of this part is by L. Baresi, S. Guinea, and M. Plebani and is entitled "Business Process Monitoring for Dependability." This paper proposes a dynamic technique for ensuring that dependability requirements of service-based business processes are maintained during runtime. The approach is based upon the concept of supervision rules, which are the union of user-defined constraints. These rules are used to monitor how a BPEL process evolves, and specify corrective actions that must be executed when a set of constraints is violated. For facilitating the specification of these rules, the authors provide suitable languages and tools that enable one to abstract from the underlying technologies, and to hide how the system guarantees the dependability requirements.

The fourth part of this book is on "Architectural Assurances for Dependability" and contains three papers. The first paper, "Achieving Dependable Systems by Synergistic Development of Architectures and Assurance Cases" by P. J. Graydon, J. C.

Knight and E. A. Strunk, explains the basic principles of assurance-based development, and shows how the proposed approach can be used to provide assurance case goals for architectural choices. In this approach, first the architecture is developed to provide evidence required in the assurance case, and then the assurance case is refined as architectural choices are made. In this context, choices are better informed than an architecture chosen in an ad hoc manner.

The next paper, entitled "Towards Evidence-Based Architectural Design for Safety-Critical Software Applications," is prepared by W. Wu and T. Kelly. This paper proposes a Triple Peaks process framework, within which a system model, deviation model, and mitigation model are proposed and linked together. The application of this framework is supported by the use of Bayesian Belief Networks and collation of relevant evidence. The link between the three models is elaborated by means of a case study. The core contribution of this paper is addressing safety using evidence available at the architectural level.

The paper "Extending Failure Modes and Effects Analysis Approach for Reliability Analysis at the Software Architecture Design Level," by H. Sozer, B. Tekinerdogan and M. Aksit, shows how the Failure Mode and Effect Analysis (FMEA) and Fault Tree Analysis (FTA) can be extended and used in combination for conducting reliability evaluation of software systems at the architecture design level. The extensions of FMEA and FTA are related to using a failure domain model for systematic derivation of failures, prioritization of failure scenarios based on a user's perception, and an FTA impact analysis model that does not explicitly require a running system. The software architecture reliability analysis approach (SARAH) proposed in the paper is illustrated using an industrial case for analyzing the reliability of the software architecture of a digital TV.

Architecting dependable systems is now a well-recognized area, attracting interest and contributions from many researchers. We are certain that this book will prove valuable for both developers designing complex applications and researchers building techniques supporting them. We are grateful to many people who made this book possible. Our thanks go to the authors of the contributions for their excellent work, the DSN 2006 WADS participants for their active participation in the discussions, and Alfred Hofmann from Springer for believing in the idea of a series of books on this important topic and for helping us to get it published. Last but not least, we very much appreciate the efforts of our reviewers who helped us in ensuring the high quality of the contributions. They are L. Baresi, L. Bass, T. V. Batista, J. Bryans, R. Buskens, F. Castor Filho, B. H.C. Cheng, A. C. Costa, B. Cukic, D. Desovski, T. Dumitras, J. Durães, A. Ebnenasir, L. Grunske, C. Heller, N. Henderson, M. Kaâniche, K. Kanoun, T. Kelly, S. Kharchenko, M. Klein, H. Lönn, T. Maxino, L. Mostarda, P. Narasimhan, R. F. Paige, P. Pelliccione, A. Radjenovic, S. Riddle, G. Rodrigues, D. Rosu, A.-E. Rugina, S. Schneele, E. Strunk, B. Tekinerdogan, M. Tichy, J. L. Tokar, S. Voss and several anonymous reviewers.

Rogério de Lemos
Cristina Gacek
Alexander Romanovsky

Table of Contents

Architecting Dependable Systems with the SAE Architecture Analysis and Description Language (AADL)

Joyce L. Tokar

Pyrrhus Software,
P.O. Box 1352, Phoenix, AZ 85001, USA
tokar@pyrrhusoft.com

Abstract. Architecture Description Languages provide significant opportunity for the incorporation of formal methods and engineering models into the analysis of software and system architectures. The SAE AADL [1] is a standard that has been developed for embedded real-time safety critical systems which will support the use of various formal approaches to analyze the impact of the composition of systems from hardware and software and which will allow the generation of system glue code with the performance qualities predicted. This paper will highlight the components and features of AADL that facilitate the development of system architectures comprised of both hardware and software components. It will demonstrate how the features of AADL may be used to conduct a wide variety of dependability analysis on AADL architectural models. To help in the understanding of AADL the paper will begin with a discussion of software and systems architecture. It will then show how the AADL supports these concepts.

Keywords: Architecture description language, Architecture analysis, Dependability, Modeling.

1 Introduction

An architecture involves multiple views (perspectives) of the system [3] and relies, in whole or part, on patterns or styles of representation. These views enable the exchange of information about a system or system of systems (SOS) across a wide variety of domains of discourse. For example, a logical view of an architecture describes the logical relationships between various components of a system that may be used to assess the logical flow of information through a system. Whereas a physical view of an architecture describes how the architecture is realized in the physical environment.

Architecture is embodied in its components, both hardware and software; their relationships to each other and the environment; and the principles governing its design and evolution. The architecture of a program or computing system is the structure or structures of the system, which comprise software and hardware elements, the externally visible properties of those elements, and the relationships among them.

Thus, architecture helps to organize a system into components and interfaces between these components. There are both functional and nonfunctional characteristics that can be modeled as properties of a component or system. These properties along with the model itself can then be used in analysis of the system.

1.1 Architecture: The Foundation of Good Software and Systems Engineering

Research in Architecture Description Languages (ADLs) has been focused on finding methods to reduce the cost of developing applications and for increasing the potential for commonality between different members of a closely related product family. Software development based on common architectural idioms has shifted from the lines-of-code view to coarser-grained architectural elements and their overall interconnection structure [4]. To support architecture-based development, formal modeling notations, analysis and development tools that operate on architectural specifications are needed. Architecture description languages and their accompanying toolsets have been proposed as the answer. An ADL for software applications focuses on the high-level structure of the overall application rather than the implementation details of any specific source module.

The AADL is an architecture descriptions language that includes support for the development of both the execution platform components and the software components in the system architectural specification. Thus, the characteristics of both the software and the execution platform are available for analysis.

AADL is based upon the ground-breaking work in architecture description languages funded by United States (US) Defense Advanced Research Projects Agency (DARPA) and the US Army Aviation and Missile Command (AMCM). Experiences from the use of the MetaH language and toolset developed my Honeywell Technology Laboratories [4] provided the foundation for the definition and development of the AADL.

1.2 Software and Systems Development with Modeling Languages

With modeling languages the approach to software and systems development is more integrated with the variety of participants from domains across the entire operational embedded system. The architecture model may be refined from the requirements phase through development into integration. This enables the detection of errors early in the process rather than at integration level. Functional interfaces and systems interface are integrated into the overall model development which provides a predictable system at the completion of development.

Analysis of the architecture may take place throughout the development cycle. Preliminary abstract models may be analyzed for feasibility prior to actual system construction. These models may also be used to evaluate interfaces and design constraints. Model analysis facilitates the early detection of errors and flaws in a system.

2 The SAE Architecture Analysis and Description Language (AADL)

A key to an AADL-based engineering process is an architectural specification that is an abstraction of the system. The architectural specification must be semantically strong enough to reflect relevant aspects of the application domain.

Since the AADL was designed for real-time embedded system, the architectural specification focuses on the task structure and interaction topology of a system and captures both the software architecture and hardware architecture. This real-time architecture model is the basis for various analyses, ranging from schedulability analysis to reliability and safety analysis.

The architectural specification is the basis for automated system generation and component integration. The actual components of a system may be hand-coded software, or components modeled in a domain-specific notation and auto-generated.

Although there is a considerable diversity in the capabilities of different ADLs, all share a similar conceptual basis, or ontology, that determines a common foundation of concepts and concerns for architectural description [5]. The main elements of this ontology include: components, connectors, systems, properties, constraints and styles.

2.1 The Elements of AADL

This section shows the correspondence between this ontology and AADL elements. These AADL entities are used to construct analyzable models of real-time, embedded, systems.

2.1.1 Components

Components represent the primary (computational) elements and data stores of a system. Intuitively, they correspond to the boxes in box-and-line descriptions of architectures. Typical examples of components include such things as clients, servers, filters, objects, blackboards, and databases. In most ADLs components may have multiple interfaces, each interface defining a point of interaction between a component and its environment.

In AADL, the definition of *components* is extended to include *execution platform components* such as buses and memories. A system is then the composition of software components, execution platform components, and possibly other system components. AADL supports multiple interfaces between components through the definition of ports. AADL also supports the concept of a family of components through the definition of multiple implementations that correspond to a component type definition.

2.1.2 Connectors

Connectors facilitate the communication channels between components and coordinate activities among components. Examples include simple forms of interaction, such as pipes, procedure call, and event broadcast. Connectors may also represent more complex interactions, such as client-server protocol or an SQL link between a database and an application. Connectors have interfaces that define the roles played by the various participants in the interaction represented by the connector.

In AADL, *connections* are represented as the actual linkage between components. *Ports* may be used to represent the flow of data and events between threads and execution platform components. *Data ports* are used for unqueued state data. *Event data ports* are used for queued message data. *Event ports* are used for events. A *port group* represents a grouping of ports or port groups. Outside a component a port