# Closing the Dependability Gap: Converging Software Engineering with Middleware

Karl M. Göschka and Lorenz Froihofer
Vienna University of Technology, Institute for Information Systems,
Distributed Systems Group, Argentinierstrasse 8/184–1,
A–1040 Vienna, Austria
{karl.goeschka|lorenz.froihofer}@tuwien.ac.at

## Abstract

*The inertness of today's software systems turns innovative applications into an obstacle rather than an enabler and results in dependability degradation during the systems' lifetime. Even more so, heterogeneity, scale, and dynamics open up what Laprie called the dependability gap. In this position paper, we identify the need to converge methods from software engineering with traditional middleware and dependable systems research to close the dependability gap. In particular, we suggest a nested control loop approach, where the inner loop addresses short-term changes autonomously, while the outer loop addresses long-term evolution by run-time software engineering.*

## 1. Dependability Gap

While computing is becoming a utility and software services increasingly pervade our daily lives, dependability is no longer restricted to critical applications, but rather becomes a cornerstone of the information society. Dependability clearly is a holistic concept: Contributing factors are not only technical, but also social, cultural (i.e. corporate culture), psychological (perceived dependability), managerial, and economical. Fostering learning is a key, and simplicity is generally an enabler for dependability.

Among technical factors, software development methods, tools, and techniques contribute to dependability, as defects in software products and services may lead to failure and also provide typical access for malicious attacks. In addition, there is a wide variety of fault tolerance techniques available, ranging from persistence provided by databases, replication, transaction monitors to reliable middleware with explicit control of quality of service properties.

Unfortunately, heterogeneous, large-scale, and dynamic software systems that typically run continuously often tend to become inert, brittle, and vulnerable after a while. The key problem is, that the most innovative systems and applications are the ones that suffer most from a significant decrease in (deterministic) dependability when compared to traditional critical systems, where dependability and security are fairly well understood as complementary concepts and a variety of proven methods and techniques is available today [1]. In accordance with Laprie [5] we call this effect the dependability gap, which is widened in front of us between demand and supply of dependability, and we can see this trend further fueled by an ever increasing cost pressure.

This is caused by some of the following reasons [7, 5]:

- Change of context and user needs: It is impossible to reasonably predict all combinations of change during design, implementation, deployment, and — most importantly — during run-time.
- Imprecise (and sometimes even competing or contradictory) requirements: Users are either inarticulate about their precise criteria for correctness, performance, dependability, and other system qualities, or different users impose competing or contradictory requirements on the system, partially because of inconsistent needs.
- Interdependencies between systems and software artefacts, and emerging behaviour: The system may be too complex to predict even its internal behaviours precisely.

As a result, traditional systems experience permanent *dependability degradation* throughout their life-time. This in turn requires continuous and highly responsive human maintenance intervention and repetitive software development processes. While this need for intervention is costly, error-prone, and hence further impairs dependability, it may, in some cases, even become prohibitively slow compared to the system's pace in normal operation.

We can see two complementary approaches to address the problem of dependability degradation: Adaptive coupling and run-time software engineering. We contribute with the proposal to integrate these two approaches in a nested control loop approach that converges methods from software engineering with methods from traditional dependability research.

## 2. Adaptive and autonomous coupling

Adaptiveness is envisaged in order to react to observed, or act upon expected (temporary) *short-term* changes of the system itself, the context/environment (e.g., resource variability or failure scenarios) or users' needs (e.g., day/night setting) and expectations (e.g., responsiveness). As this kind of adaptivity should be provided without explicit user intervention, it is also termed autonomous behavior or self-properties, and often involves monitoring, diagnosis (analysis, interpretation), and reconfiguration (repair) [4].

One of the main reasons why many approaches fell short in the past, however, lies in the major focus on the system's components (e.g., by focusing on recompilation, reconfiguration, and redeployment of components), while complexity theory [6] on the other hand clearly shows that the overall properties of large and complex software systems are largely determined by the internal structure and interaction of its parts and less by the function of its individual components. Even more so, a complex software system provides a mixture of tightly and loosely coupled parts. As an important consequence, the overall system properties are determined not only by the structure but also by the strength of coupling of its relationships.

Thus the inner control loop has to adaptively configure the strength of the architectural coupling between the system's constituents as the most promising approach to explicitly balance competing dependability and security properties of the overall system according to the respective situation. This control should flexibly be performed as interaction between infrastructure and application (or even the end user), typically through run-time selection and reconfiguration of dependability protocols, e.g., consistency of replication protocols [3].

## 3. Run-time software engineering

As not all possible evolvements can be foreseen for long-running software, *long-term* evolution has to be supported to regulate the emerging behavior of large and dynamic systems, again, with respect to the evolvement of the requirements and user expectations, but also in response to long-term changes in the context.

This will be performed by changing the system's design during run-time, which in turn requires run-time processable requirements and design-views in the form of constraints [2], models (”UML virtual machine”), or (partial) architectural configurations. The ultimate idea here is to move into run-time what previously could only be done by modifying an application off-line during design-time.

These run-time accessible and processable requirements can be stored in repositories or be accessed via reflection, aspect-oriented programming, or protocols for meta-data exchange. They can explicitly be manipulated and configured, which allows such a system to balance or negotiate certain properties against each other or against user needs during run-time.

Clearly, this requires middleware services to support manipulation of requirements and negotiation of properties and needs. The vision here is a convergence of software development tools with middleware (including traditional dependability, fault tolerance, and adaptivity concepts), to provide for *run-time software development tools in the form of middleware services* to compensate for dependability degradation by *re-engineering running software*.

## 4. Future work

Regardless of the pace of change, both approaches address the imprecise, emerging, and ever-changing nature of large and long-running software systems and introduce iterative steps of adaptation and evolvement during run-time. Both approaches are needed in practice and will need different solutions, but have in common the need for (i) run-time measurement of dependability properties, and (ii) run-time processable meta-data representing the current architectural structure and design-view. This clearly shows the need for research to converge methods from software engineering, middleware, and traditional dependable systems.

## References


[1] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[2] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka. Overview and evaluation of constraint validation approaches in Java. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 313–322, Washington, DC, USA, 2007. IEEE Computer Society.

[3] L. Froihofer, K. M. Goeschka, and J. Osrael. Middleware support for adaptive dependability. In *Proc. of the 8th ACM Int. Middleware Conference*. Springer, 2007.

[4] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM Press.

[5] J.-C. Laprie. Resilience for the scalability of dependability. In *Proc. 4th Int. Symposium on Network Computing and Applications*, pages 5–6. IEEE CS, 2005.

[6] S. M. Manson. Simplifying complexity: a review of complexity theory. *Geoforum*, 32(3):405–414, 2001.

[7] M. Shaw. “self-healing”: softening precision to avoid brittleness: position paper for woss '02: workshop on self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 111–114, New York, NY, USA, 2002. ACM Press.