

Dependability and Robustness: State of the Art and Challenges

Václav Mikolášek

Institute of Computer Engineering,
Vienna University of Technology,
mikolasek@vmars.tuwien.ac.at

Abstract

Dependability is a broad concept which covers such aspects of computer systems like reliability, maintainability, and availability. Also because of this broadness, researchers came up with related yet more focused notions such as self-healing and robustness. In this article, I discuss robustness in relation to dependability and make a distinction between these two. I outline the state of the art of the research in robustness and discuss required design practices leading to robust systems. In analogy to end-to-end argument, I present a new set of arguments called repairability which could help a designer resolve design decisions and choose a solution which potentially leads to a robust system.

1. Introduction

Whereas the notion of dependability is well defined (see for example [3]) and there is a good consensus in the research community on the sense of related taxonomy, the notion of robustness brings still many connotations. A biologist will understand robustness in terms like adaptation, stability, diversity, survivability, and perturbations [11]. A control theorist will express robustness in terms of uncertainty of mathematical models. A software developer might focus on a program's ability to deal with unusual usage or users' input. In this article, we shall assume the position of a computer system designer and consider robustness in terms of structural properties of a system (e.g. *modular decomposition*, *distribution* of tasks, and *hierarchy*) and behavioral aspects, such as *adaptability* and *recovery*.

It is worth pointing out that it makes sense to discuss robustness only in a context of *complex systems*. The notion of complex system itself is subject to research, however, we shall get along with intuitive understanding of that notion. To illustrate the role of complexity, consider the different expectations of a user of a simple device such as wrist watch, and a user of a significantly more complex system such as smart phone. In the first case, robustness of a wrist

watch is understood rather in terms of resistance to environmental (physical) conditions like rain. A wrist watch will be deemed robust if it can sustain bad weather and occasional bumping into walls. In the second case, robustness of a smart phone would be understood in rather logical terms such as data integrity, system stability, interoperability, etc. The focus of this article is on the logical aspects of robustness. The structural and behavioral aspects.

Complex systems are characterized not only by a high number of its constituents (components) but also by a numerous and non-trivial interactions among the components. This complexity allows us to draw a parallel between biological point of view on robust systems and that of a computer system designer. Keeping in mind that the biological metaphor must not be stretched too far, I shall also incorporate some of the terminology of a biologist and define robustness as a capability of a system to maintain its function despite various *unexpected* perturbations. In this definition, the word "unexpected" is emphasized to point out that a system's robustness or fragility shows only when the system is taken out of the realm of its specification – when the system is made to operate in unforeseen conditions under unforeseen requirements. Fragile systems brake down, while robust keep an acceptable (degraded) level of functionality.

I shall further elaborate on the definition, and point out its implications for computer system design. The contribution of this article is twofold:

- It presents a selection of works, which are relevant to the notion of robustness and combined together, they can be regarded as fundamentals of an *overall approach to robust system design*.
- I shall present a new set of arguments called *repairability* which could work as an acid test for "good and bad" design guidelines.

This article aims at laying down a conceptual basis through which we could understand structural and behavioral requirements of robust systems.

2. Dependability and Robustness

Avižienis et al. [3] discuss thoroughly concepts of dependability, security, other related notions, and taxonomy of dependable and secure computing. According to their work, **dependability** of a system is “the ability to avoid service failures that are more frequent and more severe than acceptable”. Major attributes of dependable systems are *availability*, *reliability*, *safety*, *integrity*, *maintainability*, and – if also security aspects are relevant – also *confidentiality*. We can see that dependability is a very broad concept. So, why do we need the notion of robustness and what can it bring to us?

With reference to related works [1, 11, 25], a definition of robustness can be put as follows: **robustness** is the capability of a system to maintain an acceptable level of service despite various *unexpected* perturbations. This definition makes clear that robustness is concerned with a system’s behavior (delivering its function, i.e., avoiding severe service failures) in unforeseen conditions. Only if a system is taken out of the scope of its specification we can observe its robustness or fragility. This attention to out-of-specification behavior is one point where robustness brings us something new compared to dependability. Another point comes from the biological background of that notion.

We do not speak about dependability in connection to complex adaptive systems (e.g. ecosystems, societies, biological cellular networks, etc.) because dependability is closely related to artificial systems. However, the robustness concept can be applied to both biological systems and complex computer systems. Viewed from the perspective of biology, robustness draws our attention to notions such as *adaptation* and *degeneracy* which in the classical understanding of dependability do not get much attention.

Gribble [6] also stresses the need for shifting the focus from fault hypothesis to the fact that a system will eventually operate in conditions which are out of specification. As Gribble puts it: “we believe that any system that attempts to gain robustness solely through precognition is prone to fragility”. In his article [6], Gribble underscores his statements by demonstrating on several examples how seemingly sensible assumptions brake down in a real-world application.

Undoubtedly, designing for robustness will require new design concepts and guidelines. However, many of the already well-established principles and ideas will be preserved. Moreover, some of them, such as design modularity, are prerequisites for robustness. In the following sections, I shall outline some of these well-tested design principles which will constitute a significant part of the future robust architectures.

2.1. Modularity and Fault Containment Units

A *module*, or a component, is a functional unit in a system. It encapsulates logic, data, and possibly resources in order to carry out its function. A module is also, at a given level of abstraction, a compositional unit of a system, it interacts with other modules via its well-specified, preferably tiny interfaces which hide the module’s internal structure [14]. A modular structure ensues weak interactions which “stabilize a community by dampening the effects of strong interactions” [7].

Systems with modular structure and weak interactions are *nearly decomposable* [24], which have the advantage of structural stability and relative cognitive simplicity. Complex adaptive systems, such as ecosystems, show structural modularity [7, 8, 9].

According to Kitano [11], “modularity is an effective mechanism for containing perturbations and damage locally to minimize the effects on the whole system”. In technical terms: modular design is necessary for creating fault containment units. A system should be partitioned into independent fault containment units [13, 19] such that faults remain contained in the affected unit. An error detection takes place on the boundary of the unit and the rest of the system in order to stop the error propagation.

2.2. Distributed Architecture

Kopetz [12] provides a set of arguments in favor of distributed architectures. Distributed architectures support *composability*, *scalability*, and *dependability* among others.

If a function or a service of a system is brought about by cooperation of several independent components (nodes) without a centralized control over the execution, we talk about *distributed control*. Distributed control follows the principles of redundancy: as long as there is a sufficient number of nodes to execute the distributed application, the service can be guaranteed. In *master-slave* architectures, the master component creates a breakpoint of the system and must be carefully protected. By employing distributed control, we abandon the assumption that a specific set of components (masters) will be functioning reliably during the whole period when the system is in field.

2.3. Hierarchical Design and Simplicity

Complex adaptive systems are structured hierarchically – they are composed of nearly independent and stable units which in turn are also composed of subunits, etc. The reason why systems such as organisms and ecosystems have hierarchical structure is explained by Simon [24]; the structural stability of subunits and their weak interactions among each other protect the system against various perturbations.

Once such stable unit exists it can be used to build a larger stable unit.

In complex artificial systems, hierarchical design plays also another, equally important role: hierarchy enables us to reason about a system on different levels of abstractions; it decreases the cognitive complexity of a system [13].

Design errors are severe causes of perturbations to the system. Therefore, good robust design guidelines should limit cognitive complexity of a system in order to abandon the assumption of designer's cognitive super powers.

2.4. Redundancy

Redundancy enhances robustness by providing backups or alternatives to replace damaged components [7]. It is one of the main system features which provisions robustness in complex adaptive systems. In artificial systems, however, the extent of redundancy is strictly limited by economical constraints. Triple modular redundancy [12] can mask a single component failure even if the component is not *fail-silent*. However, simple replication of system components assumes independent failures. But as pointed out in the introduction to this section, this might not be often the case. In order to avoid common mode failure, diversity among the redundant components is necessary.

2.5. Hourglass Model and End-to-end Argument

Hourglass model or *bow-tie structure* [10] and the *end-to-end argument* [10, 21] are well tested concepts which stand behind the robustness of the Internet architecture.

The hourglass model raised from a layered architecture of the TCP/IP protocol and from a requirement that complex functions should not be implemented as modules, but they should rather be broken up into primitive functions implemented across several layers. The IP layer is in the hourglass waist of the protocol suite, and the decision to make it "thin" turned out to be very prudent. It is a stable core which makes it possible to run huge variety of applications over a broad spectrum of communication technologies. This feature is sometimes aphoristically expressed as "IP over everything and everything over IP"

Advantages of the hourglass model are discussed, for example, in [10]. And Kitano [11] also argues in favor of the hour-glass model: "The architectural features of a modularized bow-tie structure are optimal for enhancing the robustness of the various aspects of a system."

Closely related to the hourglass model, the set of arguments collectively called **end-to-end argument** [21] states that: if a function in question can be correctly and completely implemented only with the knowledge and help of the application standing at the endpoints of the communication system, implementing the function in the communica-

tion system itself is not possible.¹ During the design decisions what functions still to implement at the IP layer and which already to omit, the designers followed the end-to-end argument; in Section 5, I would like to introduce another set of arguments called repairability. Repairability could serve a similar purpose as the end-to-end argument: it could direct a designer in the moment of a nontrivial design decision.

3. Resilience, Adaptability, and Recovery

Three general principles can be found in any system which is deemed robust. They are: *resilience*, *adaptability*, and *recovery*. As we will see, these three concepts are closely related and coherent. From this coherence we can draw a confidence that these three notions play a fundamental role in robust design.

Resilience, in our context, is defined as a capability of a component to compensate for a temporary degradation in a requested service. In other words, resilience is the capability of a component to compensate for errors and perturbations in the input. An example of such resilience, is the human's capability to understand a speech even in a noisy environment. Or similarly, to tolerate missing frames in a video stream and still successfully interpret the overall information. This specific kind of resilience is also called *cognitive resilience*. Interesting application of this concept can be found in the work of Nowroth et al. [18]. The authors used a psycho-visual model of human's perception (a quantitative expression of deviations from original picture and a damaged picture as they would be perceived by a human) to pinpoint the areas on a JPEG compressor chip where a transient fault caused a serious picture deviation. The critical areas were then subjected to hardening. An example of non-cognitive resilience is the capability of web browsers to interpret malformed HTML pages and present them to a user.

Adaptability is a system's capability to change its behavior or its logical/physical structure in order to compensate for internal or external perturbations and changing requirements. A typical example of adaptation is the distributed routing protocol of the Internet where a logical structure (topology) is adapted based on a current state of network's connectivity. Another example of an adaptation is a dynamic resource allocation mechanism which can be found in architectures such as *Time-Triggered System on a Chip* [20]. If a node is diagnosed as affected by a permanent fault, a resource manager can migrate the functionality of this component to another node which has sufficient spare resources. With the advent of FPGA technology, the possibilities for adaptability broadened. Mitra et al. [17] proposed an adaptable chip architecture for dependable systems. In this architecture, two FPGA chips execute

¹The definition is taken from [21] and modified.

the same set of functions and peer-check each other. The FPGA chip's area is subdivided into columns which can be individually reconfigured while the others columns keep executing. In the case that a permanent fault is diagnosed on one of the chips and the affected column located, the repair mechanism of the chip repeatedly tries new mappings between the functions and the chip's columns until a new and fault-free configuration is found.

A robust system must support **recovery** of its components in order to avoid accumulation of errors in the system. Once an error is detected, the system has to remove it and restore the correct state of the affected component. Indeed, recovery is only possible if the fault is not permanent. A typical and widely applied mechanism of recovery is a simple restart (more on that in Section 4). In the context of database management systems, two main recovery mechanisms are used: *rollback* and *rollforward*. In the context of dependable computer systems, the errors in components are often masked by some kind of redundancy such as triple modular redundancy (TMR). TMR without a repair is not suitable for longer missions [23] and recovery of the erroneous modules is crucial. See for example [27] for a discussion about recovery techniques in TMR.

Resilience, adaptability, and recovery constitute a self-enforcing trinity. Resilience compensates for temporary shortage/degradation in a required service. The shortage would not be temporary were it not for recovery, therefore *recovery supports resilience*. Recovering a component often implies a temporary shortage of the component's provided service and the rest of the system must either mask this shortage (by redundancy) or it must be resilient to it, hence *resilience supports recovery*. The process of adaptation, that is, finding a new system's configuration, may also entail a temporary service degradation, for example during migration of a service from one node to another. Therefore, *adaptability requires some degree of resilience* from the components. Similarly, during an adaptation, some components may be removed from and others may be integrated into a system and recovery supports this process of (re)integration – *recovery supports adaptability*.

4. Other Robust Design Guidelines

In this section, I present a collection of works which can be thought of as a basis for robust design guidelines – or even, a bit jokingly, as instances of a “grand unifying robust system theory”. Each of these works can be mapped to one or more of the three general principles identified in Section 3, namely resilience, adaptability, and recovery. It is typical for these works that they contribute with design guidelines which aim at dealing with situations where a system is in an erroneous state. These design guidelines are in sharp contrast to techniques such as triple modular redun-

dancy (TMR) which aim at *avoiding* an erroneous state in a system. Indeed, techniques like TMR will be still an integral part of dependable systems and safety critical systems in particular. Robust design guidelines will not replace such techniques, but rather complement them.

One of the causes of fragility of a system is that its components cannot deal with erroneous inputs or a lack of required service. For illustration, consider a simple example when a component *c1* requires a service *S* from a component *c2*. In software, this could be expressed as a direct method call:

```
c2.S();
```

This method call would take place either in the context of *c1*'s execution as one of the steps in *c1*'s thread (process) implementation, or in a body of some *c1*'s function invoked externally. What happens if method *S()* returns an erroneous value or if it blocks the calling process? How should *c1* recover from such situation? A possible answer to these questions is outlined in an inspirational paper by Candea and Fox [4] and further elaborated on in no-less interesting paper by Candea et al [5].

4.1. Recursive Restartability and Microreboots

The main idea in these articles [4, 5] is based on the observation that a simple restart of a system often helps bring the system back to a consistent (and error-free) state and subsequently avoid the error that was previously caused by a *transient* fault. Indeed, the initial state of a system is often the most studied and understood state. But a restart of a whole system is expensive and time demanding. However, if it was possible to restart small parts of the system quickly and independently of the rest of the system, the restart technique could be employed even in systems with strict temporal requirements. These restarts of small subparts of a system were dubbed *microreboots* in [5]. On the same note, a computer system designed so that each of its parts can be independently rebooted is called *recursively restartable* [4]. The “recursiveness” comes from the fact that a system is a hierarchy of subsystems - each of them restartable - with atomic restartable components on the bottom of the hierarchy.

Let us now get back to our example with a component *c1* requiring a service from a component *c2* and suppose that *c2* is recursively restartable. If *c1* detects an error in the *c2*'s service, it could issue a restart on *c2*. Now, *c2* could either restart itself entirely, or it could further issue a restart on one or more of its subcomponents. During the restart, *c1* has to be able to cope with the temporary lack of the service; in other words, *c1* should be *resilient*. When the restart of *c2* completes, *c1* can repeat its request. Of course, if the service invocation still had the form of a direct

method call (i.e., $c2.S()$) then $c1$ still would not be able to recover if method $S()$ blocked the caller, for example, due to a deadlock. For this reason, we need a different paradigm of component interaction – *message passing*.

Besides the need for component resilience and the message passing paradigm, the authors identified several other design guidelines which support recursive restartability [4]. I shall only mention two of them: using *soft state* in component communication (collaboration), and designing an application with *fine grain workloads*.

A **soft state** in component collaboration means that a shared state among components can be eventually lost or be temporarily inconsistent without severe ramifications. A component which crashes can recover its soft state upon a restart and resume a communication without other components having to store this state. Put differently [10], soft state of a communication is lost only when the two of the end points of the communication are lost. That is, a network itself is not responsible for carrying the state. In contrast, hard state must be mirrored and kept consistent in the system in order to resume a computation or a communication after a transient failure. The use of a soft state simplifies the restart procedure of a component, making it possibly the same as the component's cold start procedure.

Structuring an application around fine grain workloads is another design recommendation of Candea et al. Components should be designed such a way, that the services they provide can be invoked in fine-grained steps so that the interactions among components are also fine-grained. Such a design supports quick partial restarts and “leads directly to the simple replication and failover techniques found in large cluster-based Internet services” [4]. A class of algorithms called anytime algorithms follows this guideline very well.

4.2. Anytime algorithms

Algorithms which can provide an approximation of a result in a short time and provide an improved approximations in subsequent runs are called *anytime algorithms*. For example, an anytime path planning algorithm, called *anytime dynamic A** [15], can find quickly a suboptimal path and improve on this solution in next steps of the algorithm. Anytime algorithms operate in cycles and the duration of a cycle is shorter than searching for a solution with classical one-run solvers. Although finding an optimal solution can take more time with an anytime algorithm than with a one-run solver, robust design favors anytime algorithms. They operate in short cycles, therefore they directly follow the recommendation of Candea et al. to structure your application around fine grain workloads (see Section 4.1). Once a cycle of an anytime algorithm completes, a system has a result in hand and if it detects an error in the service, it can issue a restart on the responsible component. Anytime al-

gorithms trade precision for computational resources. This trade-off has implications for scheduling; Shackleton et al. [22] discuss exactly this topic in their work. The authors conclude that “anytime scheduling is a promising approach that helps reduce the design complexity of real-time embedded control applications, and potentially improves the overall system performance”. Note that in robust systems, we expect a degree of resilience towards a degraded service. This resilience complements well the precision/time trade-off made by anytime algorithms.

4.3. Degeneracy

A degree of *degeneracy* in a system, is the degree to which a certain part of a system can take over a function of another system's part. In contrast to redundancy, degeneracy aims at expressing the capability of a subsystem to provide functionality *similar* to functionality of another subsystem. Formally, degeneracy is expressed in terms of *mutual information* among different elements, see for example [16, 26]. Tononi et al. [26] understand redundancy in a rather narrow sense (“same function is performed by identical elements”) and define **degeneracy** as: “the ability of elements that are structurally different to perform the same function”. But they further stress that “unlike redundant elements, degenerate elements may produce different outputs in different contexts”. However, in computer science (Tononi et al. have a neuroscience background), we usually do not understand redundancy in this narrow sense. Designers of fault-tolerant systems are fully aware of desirable effects of diversity (e.g., N-version programming [2]). Still, degeneracy could be a useful concept: designing for degeneracy would lead to a bottom-up approach in which a system is built out of a small set of general purpose components which can be combined in numerous ways. Systems with high degree of degeneracy could be made highly adaptive.

5. Repairability

So far, I have mentioned several robust design guidelines: modularity, hierarchy, distribution/message passing, hour-glass model, soft state in communication, fine grain workloads/anytime algorithms and others. However, design process is a complex task and a designer faces many non-trivial decisions. How to distinguish between a good design decision and a bad one? How to tell in advance that a particular decision will lead potentially to a robust design? In this section, I would like to offer a possible answer to these questions. I shall present a set of argument called *repairability argument*.

Repairability is a system property which guarantees that a system can be relatively *easily* repaired after a permanent

or transient fault has occurred. It is not required that a particular repair technique is actually employed (implemented) but it is required that only those design practices are applied which support an easy and fast repair of an arbitrary system part. The repairability argument states:

From two design alternatives, a designer should choose the one which provides for a higher degree of system's repairability.

It makes sense to focus on system's repairability if we take seriously the argument that we cannot gain robustness solely based on fault avoidance or fault masking. The repairability argument draws our attention to the fact that components' failures are bound to happen and it favors the design practices which ease the process of restoring the system's correct state.

When a transient or a permanent fault causes an error in a system, a part of the system may cease to deliver the intended service and the erroneous components should be repaired. A typical repair consists of the following steps:

1. Error detection
2. Location of the erroneous components
3. Analysis (based on which a particular repair action is chosen)
4. A specific repair action
5. Reintegration of the service into the system

Repairability requires that each of these steps (which themselves can comprise several sub-steps) can be undertaken relatively *fast and cheap*. The step 4 – a specific repair action – will usually be one of the following: a component's recovery (for example via a restart, rollback, rollforward, etc.) or system reconfiguration (e.g., another system component takes over the task from the affected component). *Repairability argument can resolve situations in which a designer has to choose from different design alternatives:* Focusing on the cost and time efficiency of each of the repair steps is the key for resolving the design issues – in other words, two design solutions can be compared based on the degree of effortlessness of repair steps which are necessary for removing a chosen type of errors from the system. But where could we draw our confidence from that a design for repairability leads to robust systems?

I have argued in Section 3 that recovery, adaptability, and resilience are the three basic pillars of robust systems. And in order to have a confidence in the repairability argument, it is necessary to show that 1) applying this argument increases the degrees of recovery, adaptability, and resilience in a system, and 2) the repairability argument favors generally acknowledged robust design guidelines (e.g., soft state

and hour-glass model). To show all of that is a topic for an extensive future work. In the remaining lines, I sketch out only the intuitive reasoning which makes me believe that repairability argument favors robust design principles.

First, adaptability and recovery are themselves means of a repair. Therefore, the repairability argument should naturally prefer those solution which provide for both of them. Second, resilience, as argued in Section 3, supports both recovery and adaptability, and we could therefore believe that repairability argument would also lead to an increase in a components' resilience. Further, designing for repairability would require a shared state among two components be minimal and preferably soft so that a communication of the two components can be quickly disrupted and – after a repair action – again restored. So, soft-state principle is also "recognized" by the repairability argument.

Comparing two design solutions based on the fast-and-easy repair criterion might be a non-trivial task on its own. And that's where limitations of the repairability argument stem from. A repair, as mentioned above, comprises five steps, that is, we must compare two design solutions in five dimensions. To make it more complicated, each of the repair steps can have a distinct weight. For example, fast error detection can have a higher priority than fast service reintegration procedure. Notwithstanding these limitations, I still believe that repairability argument could be helpful during many design decisions.

6. Summary

Dependability is a very broad concept which covers various aspects of computing like safety, availability, reliability, maintainability, and others. Robustness is a narrower concept with a focus on a system's functioning in unforeseen conditions under unforeseen requirements.

I have argued in this article that three basic principles can be found in any system which is deemed robust. They are: resilience, adaptability, and recovery. Resilience is a capability of a component to compensate for errors in an input or temporary lack of a required service. Adaptability is a capability of a system to change its logical or physical structure in order to compensate for errors and components' failures. Mechanisms of recovery can remove errors which were caused by transient faults. These three concepts constitute a self-enforcing trinity. Adaptation requires a degree of components' resilience and recovery. Moreover, recovery and resilience mutually support each other.

Robust design guidelines will incorporate a big body of already well-established design principles such as modularity, hour-glass model and end-to-end argument, decentralized architectures, hierarchical design, and redundancy. In this article, I have also presented a selection of various works which identify some new (or less established) robust

design guidelines. The principles mentioned here were: soft state in communication, structuring application around fine grain workloads, and usage of anytime algorithms. Further, designing for degeneracy was identified as a possible candidate for another robust design guideline.

Finally, I have introduced, although in rather intuitive terms, a new set of arguments called repairability. It could serve as an acid test for “good and bad” robust design guidelines. A system is called repairable if it is highly dependable. In other words, an arbitrary error in a system can be quickly and cheaply repaired. The reparability argument states that a designer should favor those design solutions which increase the degree of reparability of a system.

Acknowledgment Many thanks go to Sven Bunte, Albrecht Kadlec, and Roman Obermaisser for their reviews of early drafts and their valuable comments and suggestions. I’m also grateful to Algirdas Avizienis for the discussion we had and his ideas on the repairability argument.

This work was in part supported by European research project GENESYS under the grant number FP7-213322.

References

- [1] ARTEMIS strategic research agenda working group. Strategic research agenda reference, designs and architectures. <http://www.artemis-office.org>, 2006.
- [2] A. Avizienis. The N-version approach to fault-tolerant software. *Software Engineering, IEEE Transactions on*, SE-11(12):1491–1501, Dec. 1985.
- [3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan.-March 2004.
- [4] G. Candea and A. Fox. Recursive restartability: turning the reboot sledgehammer into a scalpel. *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 125–130, 20-22 May 2001.
- [5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *OSDI’04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.
- [6] S. Gribble. Robustness in complex systems. *Hot Topics in Operating Systems*, pages 21–26, 20-22 May 2001.
- [7] E. Jen, editor. *Robust Design*, chapter Cross-System Perspectives on the Ecology and Evolution of Resilience, pages 151–172. Oxford University Press, 2005.
- [8] E. Jen, editor. *Robust Design*, chapter Robustness in Ecosystems, pages 173–190. Oxford University Press, 2005.
- [9] E. Jen, editor. *Robust Design*, chapter Directing the Evolvable: Utilizing Robustness in Evolution, pages 105–134. Oxford University Press, 2005.
- [10] E. Jen, editor. *Robust Design*, chapter Robustness and the Internet: Design and Evolution, pages 231–271. Oxford University Press, 2005.
- [11] H. Kitano. Biological robustness. *Nature*, 5(11):826–837, Nov 2004.
- [12] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston: Kluwer Academic Publishers, 1997.
- [13] H. Kopetz. The complexity challenge in embedded system design. *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 3–12, 5-7 May 2008.
- [14] H. Kopetz and N. Suri. Compositional design of rt systems: a conceptual basis for specification of linking interfaces. *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 51–60, 14-16 May 2003.
- [15] M. Likhachev, D. Ferguson, G. Gordon, A. T. Stentz, and S. Thrun. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.
- [16] J. Macia and R. V. Solé. Distributed robustness in cellular networks: insights from synthetic evolved circuits. 2008.
- [17] S. Mitra, W.-J. Huang, N. Saxena, S.-Y. Yu, and E. McCluskey. Reconfigurable architecture for autonomous self-repair. *Design & Test of Computers, IEEE*, 21(3):228–240, May-June 2004.
- [18] D. Nowroth, I. Polia, and B. Becker. A study of cognitive resilience in a JPEG compressor. *International Conference on Dependable Systems and Networks*, June 2008.
- [19] R. Obermaisser, C. El-Salloum, B. Huber, and H. Kopetz. Fundamental design principles for embedded systems: The architectural style of the cross-domain architecture GENESYS. In *IEEE International symposium on Object-Oriented Real-Time Distributed Computing*, Mar. 2009.
- [20] C. E. Salloum, R. Obermaisser, B. Huber, H. Paulitsch, and H. Kopetz. A time-triggered system-on-a-chip architecture with integrated support for diagnosis. *DATE’07 Workshop on “Diagnostic Services in Network-on-Chips - Test, Debug, and On-Line Monitoring”*, Apr. 2007.
- [21] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.
- [22] J. Shackleton, D. Cofer, and S. Cooper. Anytime scheduling for real-time embedded control applications. In *Digital Avionics Systems Conference*, Oct. 2004.
- [23] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (2nd ed.): design and evaluation*. Digital Press, Newton, MA, USA, 1992.
- [24] H. A. Simon. *The sciences of the artificial (3rd ed.)*. MIT Press, Cambridge, MA, USA, 1996.
- [25] J. Stelling, U. Sauer, Z. Szallasi, F. J. Doyle, and J. Doyle. Robustness of cellular functions. *Cell*, 118:675–685, Sep. 2004.
- [26] G. Tononi, O. Sporns, and G. M. Edelman. Measures of degeneracy and redundancy in biological networks. In *Proceedings of the National Academy of Sciences of the United States*, 1999.
- [27] S.-Y. Yu and E. McCluskey. On-line testing and recovery in TMR systems for real-time applications. *Test Conference, 2001. Proceedings. International*, pages 240–249, 2001.