

VIENNA UNIVERSITY OF TECHNOLOGY

Institut für Technische Informatik
Research Report 182-1/2008/26

Robustness in Complex Systems – State of the Art Report

Václav Mikolášek



TECHNISCHE
UNIVERSITÄT
WIEN
VIENNA
UNIVERSITY OF
TECHNOLOGY



Abstract

The research of robustness aims at identifying underlying structural and organizational principles of robust systems, that is, systems that keep delivering their function in spite of various *unexpected* perturbations. This report gives an account of the state of the art of the research of robustness in the context of complex adaptive systems (such as ecosystems) and complex computer systems.

In this report, we point out the differences between robustness and dependability, and particularly fault-tolerance. We also discuss the roles of assumptions, constraints, and models when designing a system and argue that some of the prevailing assumptions should be reconsidered for the sake of robustness. Later, we list and outline a few well-tested design practices that will surely make up a significant part of a body of robust design guidelines.

We see robustness as an interplay of resilience, adaptability, and recovery. These three concepts constitute a self-enforcing trinity. Resilience copes with temporary degradation of a service and therefore needs recovery which turns otherwise permanent failures into temporary ones. Recovery needs resilience because the process of a component recovery often implies a temporary service shortage, and the rest of the system must be able to deal with the shortage. Adaptability needs both resilience and recovery. The former because the means of adaptation may also ensue temporary lack of service, and the later because adaptation may consist of removing from and (re)integrating components into a system, both actions being supported by recovery.

Contents

1	Introduction	4
1.1	The Need for Robustness in Computer Systems	4
2	Demarcation of Robustness	5
2.1	Reconciling Robustness with Dependability, Stability, and Resilience	5
2.2	Highly Optimized Tolerance	6
2.3	Interplay of Resilience, Adaptability, and Recovery	6
3	Assumptions, Constraints, and Models	7
3.1	Bounded Synchrony	9
3.2	Memory Leaks and Independent Failure Modes	9
4	Required Design Practices	10
4.1	Modularity and Fault Containment Units	10
4.2	Distributed Architecture	10
4.3	Hierarchical Design and Simplicity	11
4.4	Redundancy	11
4.5	Hourglass Model and End-to-end Argument	12
4.6	Error Detection	12
5	Resilience	13
5.1	Cognitive Resilience	14
5.2	Resilience Based on a priori Knowledge	14
5.3	Quantifying Resilience	15
5.4	Modeling Resilience of Composed Timed Systems	15
6	Adaptability	16
6.1	The Role of Diversity and Heterogeneity	17
6.2	Modeling Adaptation	18
6.2.1	Contingency Schedules – An Example of Static Adaptability	19
6.3	Adaptation in Communication	19
6.4	Self-organization	19
7	Recovery	20
7.1	Recursive Restartability	21
7.1.1	Design guidelines for RR systems	21
7.2	Ground State and its Monitoring	22
8	Conclusion	23
A	Remarks on Robustness in Real-Time Systems	24

1 Introduction

Fragile systems, as opposed to *robust* ones, tend to break down when they face unexpected situations or changes in environmental conditions. Many natural systems show a high degree of robustness while many artificial systems show fragility. The questions “how to build robust systems?” and “what architectural, organizational, and strategic decisions lead to robust systems?” are in the spotlight of the research on robustness.

Any concept which we want to study scientifically must not be too broad; the part of reality which a concept excludes is as important as the one it includes and concepts which cover too much cannot be studied with a rigor [38]. Regarding the conceptual span of robustness, John Holland wittily pointed out [18] that “we don’t usually care about a robustness of a rock”. This report will provide information about the current state of the research of robustness as it is studied in the context of *complex artificial systems*. Although we concentrate on the aspects of robustness that relate to real-time system, we shall also overview the areas of *complex adaptive systems* (such as ecosystems, economies, or societies). However, we do not cover the already well studied and specialized areas such as *robust control* or *robustness of numerical methods*. With regards to robust control, Kitano says [27]: “Control theory is often used to explain robustness that involves feedback regulation, but this only covers one aspect of robustness”.

Complex adaptive system are rarely built or designed by a human¹ and they qualitatively differ from complex artificial systems such as real-time embedded applications. Later we shall have a closer look on the differences between these two system categories and we will try to estimate the limitations of drawing analogies between them. Notwithstanding the dissimilarities, the study of robustness in complex adaptive systems is often an important inspiration and source of terminology and key concepts. The Santa Fe Institute [1], founded in 1984, notably contributes to the research of complex adaptive systems and robustness.

1.1 The Need for Robustness in Computer Systems

The work of Shivakumar et al. [42] clearly showed that the soft error rate on chips will be the pervasive source of transient faults in hardware overtaking even the soft error rates of memories.² Malik [34] pointed out that during the shift from $600\eta m$ to $50\eta m$ the soft error rate increased by nine orders of magnitude. And Kopetz [30] called our attention to the fact that today many embedded computer applications are developed under the assumptions that

1. the hardware works always as described in the documentation,
2. the software is free of design errors, and
3. the clients use the system as specified.

In reality, these assumptions do not hold and that in turn leads to systems that are fragile. Gribble [15] argues that “any system that attempts to gain robustness solely through precognition is prone to fragility”.

¹Counter examples exist – Internet is clearly a complex adaptive system.

²The authors made, in 2002, a prediction of the soft error rates for currently available technologies.

2 Demarcation of Robustness

In general terms, *robustness is a capability of a system to deliver its function or maintain its property of an acceptable level in spite of external and internal perturbations*. The acceptable level of the system’s service (or function) is always determined by the user of the service or eventually by an observer of the system if we have a system’s property in mind. In more technical terms [45]:

[The robustness is] the capability of a system to deliver an acceptable level of service despite the occurrence of transient and permanent hardware faults, design faults, imprecise specifications, and accidental operational faults.

It follows from the definition, and it is also stressed in [19], that we must always speak about robustness in the context of a concrete property (or a function) and a perturbation. System can be seen as robust only after it is specified what property the system maintains in spite of what perturbations. We call every property which should be maintained despite given perturbations a *critical property*³. A critical property under certain perturbation need not to be a critical property under different perturbations. For example, network connectivity is a critical property under links and nodes failures but is not a critical property under application’s Heisenbugs.

There is a somewhat intrinsic contradiction in this definition: in [18] the perturbation are also regarded as *inherently unforeseeable*, yet we need to specify them prior to any discussion about a system’s robustness. We argue that there is no reconciliation of this contradiction, moreover the contradiction is the driving force of the research in robustness: to come up with design guidelines which, if followed, make a system resilient to unforeseeable changes and threats. Such design guidelines would be surely a technological achievement and as such they should be also rigorous in order to allow reasoning about their effects. The research of robustness occupies this space between rigorous and undefined.

Regarding computer systems, perturbations and *faults* overlap to such a degree that we consider them as synonyms. Avizienis et al. [4] provided a fault taxonomy which covers a wide span of faults ranging from those caused by cosmic radiation to human-made. However, as we shall see in the next section, robustness and *fault tolerance* (eventually *dependability*) are not synonyms.

2.1 Reconciling Robustness with Dependability, Stability, and Resilience

Why do we need robustness if we have dependability? Real-time systems in safety critical applications are designed for dependability. A dependable system is reliable, highly available, safe, and secure as well as maintainable [29, 4]. Although reliability, safety, and availability are also hallmarks of robust systems, the classical design of dependable systems rely on *precognition*; *fault hypothesis* and its *coverage* must be stated prior to developing and deploying concrete precautions which make the system *fault-tolerant*. Such systems are dependable as long as the system and environmental parameters stay in the realm of a specification; Avizienis et al. [4] put that very clearly: “the classes of faults that

³Eventually also *critical function* or *critical feature* with analogical meaning.

can be tolerated depend on the fault assumption that is being considered in the development process [...]”. In contrast, robust design should imply dependability as well as reasonable operability in *unexpected* conditions, that is, the classes of tolerated faults reach beyond specifications and fault assumptions. There are no doubts that a vast body of good design principles, such as modularity, employed in the current design of dependable systems, will also be part of the robust design guidelines.

Regarding stability, Erica Jen [19] argues that the theory of stability cannot provide us with the answers that we seek in robustness. Despite the fact that both concepts talk about preserving an acceptable level of a system’s function in spite of perturbations, stability does not answer the fundamental question what is the interplay of structural organization of a system and its stability. The research of robustness, in contrast to theory of stability, deals with parameters of systems which are not easily *quantifiable* such as clients using the system discordantly with the specification. Nonetheless, stability remains a part of the conceptual suite of robustness and we should pay attention to it whenever it is useful.

Robustness is sometimes called *resilience* [20], especially in the studies of complex adaptive systems such as ecosystems. We shall, however, reserve the term resilience for a specific meaning – capability of a component to compensate for erroneous input – and it should not be considered as a synonym for robustness.

2.2 Highly Optimized Tolerance

Highly optimized tolerance (HOT) is a theory of robustness and distribution of power laws in complex systems formulated by Carlson and Doyle in [8]. The authors focus on systems which are optimized through *natural selection* or engineering design, to provide a robustness despite uncertain environments. The HOT received a deal of attention, see, for example, [27]. The authors see the investigated systems as results of tradeoffs among yield, cost of resources, and tolerance to risks. As one of its key points, the HOT theory suggests that the designed, eventually evolved, systems are robust to general perturbations but extremely fragile against certain types of rare perturbations. In other words, the systems are *robust yet fragile* [21]: “These ‘robust yet fragile’ features [of complex systems] are neither accidental nor artificial and derive from a deep and necessary interplay between complexity and robustness, modularity, feedback, and fragility.”

Kitano [27], in relation to the work of Csete and Doyle [11], also claimed that “the nature of robustness and fragility that is predicted by the HOT model is generally consistent with observed properties of designed complex systems and is notably different from other models. Csete and Doyle further argued that trade-offs between robustness and fragility indicate that robustness is a conserved quantity, a concept that also applies to biological systems.”

2.3 Interplay of Resilience, Adaptability, and Recovery

Webb and Levin [20] regard robustness as an interplay among *diversity and heterogeneity*, *modularity*, and *redundancy*. Diversity and heterogeneity support system adaptability. Modularity facilitates segmentation of a complex system

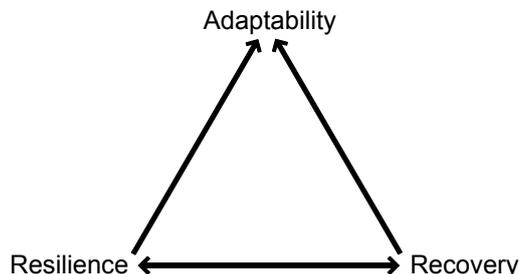


Figure 1: The *support* relation among the resilience, adaptability, and recovery. The RAR is a self enforcing trinity.

into quasi fault-containment regions and thus lessens the impact of severe perturbations which do not cross such modules. Modules are also functional units which are, by theory, replaceable. Redundancy provides a buffer for system-wide perturbations and makes individual subsystems (modules) resilient to disastrous events. Webb and Levin further argue that there exist tradeoffs between the three identified principles [20]: “some forms of redundancy may compromise other structural features, in particular diversity (which allows adaptation) and modularity (which limits disturbances)”.

We see robustness as an interplay of *resilience*, *adaptability*, and *recovery* (RAR), which is a self enforcing trinity of concepts. Resilience copes with temporary lack or degradation of service. The lack of service would not be temporary, were it not for recovery. Recovery is the capability to recover a component and reintegrate the component and its service back to the system after an error is detected. Recovery of a node would not be of any use, if the rest of the system could not cope with the temporal shortage of the component’s service, therefore recovery is best accompanied with resilience. Adaptability of a system ensures that a system can handle severe changes in the environment or its internal structure. The means of adaptability can cause a short-term lack of some services, thus adaptability requires resilience to compensate the service shortages. Furthermore, in order to adapt it might be necessary to disconnect some components from as well as integrate new ones into the system, therefore adaptation requires recovery because recovery entails component (re)integration. This mutual support relation is depicted in Figure 1. This report is structured around RAR and in the Sections 5-7 we concentrate on each of the concepts separately.

3 Assumptions, Constraints, and Models

In this section, we advocate that when designing for robustness some of the assumptions, constraints, and models, which otherwise serve as good means in the common design approach, must be reconsidered. We shall present conclusions which stem mainly from the current research of robust design of computer systems.

Gribble, in his inspirational paper [15], summarized his experience gained during a development and deploying of a large distributed application which, despite of a cautious design, turned out to be fragile when it faced unexpected situations. He generalizes these experiences into a set of high level design guide-

lines and points at the danger of trying to gain robustness solely based on assumptions made about the system and its environment. Likewise, Willinger and Doyle [22] stressed that “a careful design aimed at preventing failures from happening is a hopeless endeavor, creating overly complex and unmanageable systems”.

A system designer must have a sufficiently clear picture about the environment in which the given system will operate. This picture is called a model and it is a simplified description of reality. The designer should also have a model of the system itself in order to be able to reason about it. Humans are not capable of thinking about reality without building such simplified models and there is always a tradeoff between the accuracy of a model and its *cognitive complexity* [31]. We make assumptions about the part of reality which is uncertain or unknown to us. Such assumptions are built in into our models and are the *breakpoints* of the system. We have already mentioned in Section 1 several such assumptions: hardware works as documented, software are free of design faults, clients use systems according to specification. We cannot design a system without stating some assumptions, but we also must not assert hypotheses which in turn lead to fragile design. It is often the case that a higher degree of robustness is gained when we abandon a certain assumption⁴, therefore it is useful, whenever we speak about concrete means to achieve robustness, to mention also the assumptions which were abandoned and those which were added.

A designer of a system component opts between imposing constraints on the component or on the environment. For instance, a component may either strictly require that an input is syntactically correct, or it might tolerate some deviations. In the former case, the designer imposes constraint on the component’s environment, in the later case on the component itself. The choice is not often trivial and there is no simple universal answer. It is not the case that a designer should always impose constraints on the component rather than on its environment. Consider again a component which is (1) strict about its input syntax and another (2) which tolerates a degree of incorrect syntax. The design in the first case constraints the environment (thus making assumptions) and although the implementation could be relatively simple, the component is fragile with regards to perturbations in message syntax. In the second case, the component is made robust against the syntax perturbations but for the price of increased implementation complexity which in turn can again lead to fragility. We can see that there is a nontrivial tradeoff between complexity and resilience. In the context of designing a communication protocol, namely Internet protocol, Willinger and Doyle [22] argue in favor of placing constraints on the end nodes rather than on the communication subsystem. They follow the *end-to-end argument* of Saltzer et al. [41]. Naturally, a designer cannot impose constraints on the part of the system or the environment that is out of *sphere of control* [29] of the system.

What follows are some examples, taken from Gribble [15], of assumptions which are seemingly reasonable but which in fact led to fragility. The system under discussion is *distributed data structures* (DDS). The DDS is a high-capacity, high-throughput, virtual hash table that is partitioned and replicated across many individual storage nodes called bricks.

⁴Specifically assumptions which are optimistic, that is, those which place constraints on what can go wrong.

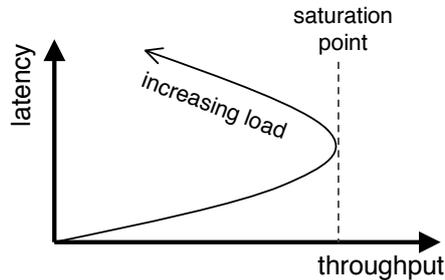


Figure 2: Performance response with saturation point. The graph depicts the parametric curve of latency and throughput as a function of load. Taken from [15].

3.1 Bounded Synchrony

Various components in the DDS relied on timeouts to detect the failure of remote components. Because of the low-latency ($10\text{-}100\mu\text{s}$), redundant network in the cluster, the timeout values were set to several seconds, which is four orders of magnitude higher than the common case round trip time of messages in the system.[15] The designers then assumed that components that did not respond within this timeout had failed: they assumed *bounded synchrony*.

The bricks (storage nodes) were implemented in Java and therefore made use of garbage collection. The higher the number of requests on a brick, the longer it took the garbage collector to reclaim a fixed amount of memory. If the DDS were operating *near saturation point*, slight (random) fluctuations in the load received by bricks in the system would increase the pressure on their garbage collector, causing the effective throughput of these bricks to drop. Because the request load is independent of this effect, the degraded bricks started to accumulate more active objects in the memory, that in turn led to increased delays caused by garbage collection and further degradation of performance. The performance response is depicted in Figure 2.

Once the system was pushed past saturation (by slight perturbations), the described catastrophic effect slowed down the affected bricks until their latencies exceeded the timeouts in the system. The presence of garbage collection caused the system to violate the assumption of bounded synchrony as it approached and then exceeded saturation.

3.2 Memory Leaks and Independent Failure Modes

The DDS made use of replication in order to gain fault-tolerance: by replicating data in more than one location, the system gained the ability to survive the faults of individual component. The underlying assumption was such, that *failures would be independent*, and therefore the probability that multiple replicas would simultaneously fail is vanishingly small.

The brick had a latent unprotected race condition that did not affected correctness but caused a memory leak. Under full load, the rareness of this race condition caused memory to leak at the rate about $10\text{KB}/\text{minute}$. The Java Virtual Machine was configured to limit its heap size to 50MB . Given this leak rate, the bricks' heaps would fill after approximately 3 days.

The request load was roughly balanced across the system. When launching the system, all the bricks were launched at nearly the same time. These two facts caused a *correlated failure of nodes*: after several days of running, the bricks ran out of memory and started to fail within 10-20 minutes of each other. The assumption of independent failures was violated.

4 Required Design Practices

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 this section, we shall outline these well-tested design principles which will constitute a significant part of the future robust architectures.

4.1 Modularity and Fault Containment Units

A *module*, or a component, is a functional unit in a system. It encapsulates the logic, data, and possibly resources in order to carry out its function. A module is also, at a given level of abstraction, the compositional unit of a system, it interacts with other modules via its well-specified, preferably tiny interfaces which hide the module's internal structure [28]. Modular structure ensues weak interactions which “stabilize a community by dampening the effects of strong interactions” [20]. Systems with modular structure and weak interactions are *nearly decomposable* [43], which have the advantage of structural stability and relative cognitive simplicity. Complex adaptive systems, such as ecosystems, show structural modularity [20, 23, 24].

According to Kitano [27], “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. System should be partitioned into independent fault containment units [31] such that faults remain contained in the affected unit. The error detection takes place on the boundary of the unit and the rest of the system in order to stop the error propagation.

4.2 Distributed Architecture

Kopetz [29] 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.

4.3 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 [43]; 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 [31].

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

4.4 Redundancy

Redundancy enhances robustness by providing backups or alternatives to replace damaged components [20]. 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 [29] can mask a single component failure even if the component is not *fail-silent*. However, simple replication of system components assumes independent failures. As described in Section 3.2, this might not be often the case. In order to avoid common mode failure, diversity among the redundant components is necessary.

We talk about *replica determinism* [29] if the set of replicated error-free components provide the same output (i.e., a service or a result) at points in time not further from each other than a given constant d . Replica determinism is a desired property particularly in real-time systems since the interval d gives the number of time units after which a system can detect and mask failures of individual replicas. A computation of a replica-determinate ensemble is characterized by the interval d , during which all the error-free replicas provide their outputs followed by an interval, $D > d$ during which they are silent. In other words, the outputs are confined to short, sparse time intervals of the otherwise dense time line, and we say that they occur on a *sparse time base*. From a designer’s point of view, handling components replication is a complex task and the system architectures which exploit the notions of replica determinism and sparse time base simplify it significantly (see, for example, [29]). On the other hand, both concepts introduce new assumptions, such as components synchronization and value consistency across the ensemble, the violation of which could cause a system failure. Therefore, the system architecture itself must carefully protect the appropriate properties and services which maintain such assumptions e.g., *clock synchronization*.

Redundancy in the representation of an information is necessary for error detection and error correction.

4.5 Hourglass Model and End-to-end Argument

Hourglass model or *bow-tie structure* [22] and *end-to-end argument* [41, 22] are well tested concepts which stand behind the robustness of the Internet architecture.

The hourglass model raise from a layered architecture of the TCP/IP protocol and from the 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. As stated in [22]:

The layers below the waist (i.e., physical and link) deal with the wide variety of existing transmission and link technologies and provide the protocols for running IP over whatever bit-carrying network infrastructure is in place (“IP over everything”). [...] Above the waist is where enhancements to IP (e.g., TCP) are provided that simplifies the process of writing applications through which user usually interact with Internet (“everything over IP”)

And also:

“IP over everything and everything over IP” results not only in enormous robustness to changes below and above the hourglass’ waist but also provides the flexibility needed for constant innovation and entrepreneurial spirit at the physical substrate of the network as well as at the application layer.

Kitano [27] 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 [41] states: 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 communication system itself is not possible.

The term *fate-sharing*, coined by Clark [10], denotes a feature of a protocol design which follows the end-to-end argument. This feature is described in Carpenter [9] as follows: “end-to-end protocol design should not rely on the maintenance of state (i.e., information about the state of the end-to-end communication) inside the network. Such state should be maintained only in the endpoints, in such a way that the state can only be destroyed when the endpoint itself breaks.”

4.6 Error Detection

Good error detection coverage is a precondition for many successful measures of robustness. It accompanies the principle of fault containment units (FCU) for which it provides a “safety net”. The faults remain contained in the FCU, but the caused error would spread further into the system if not detected and subsequently masked or disposed.

Sosnowski [44] surveys various error detection mechanisms and their principles for different hierarchical levels of a system. On the *circuit level*, Sosnowski mainly discusses detectors based on information redundancy, these include among others *Hamming codes*, *Berger code*, and *CRC*. Research in the circuit level detectors, online checking (memory contents as well as control signals), and recovery is presented in [37]. The *system-level detectors* verify the correctness of the system operation by checking its various general properties [44]. The techniques here include hardware replication and voting mechanism (such as TMR) and *2-version programming*. On *application level*, the error detection comprises, for example, algorithm-based techniques such as *assertions*.

5 Resilience

A user of a service is called *resilient* if it can tolerate temporary degradation or lack of the required service, or the errors in it. In other words, resilience is the capability of a component to compensate for errors and perturbations in the input. In the environment where components failures are rules rather than exceptions – and we argued in the introduction that it is actually the case – the occasional lack of required service is an inevitable reality and provision of resilience in components is a necessity. When we talk about a service provided to a component, we do not limit ourselves only to the *value domain*, but whenever it is useful, we extend the concept of service also to the *temporal domain* [29]. Particularly in real-time system, a service must be provided correct and on time. A real-time system component which can tolerate deviations in the time and value domains of the required service will be considered resilient. Similarly to robustness, we should always talk about a concrete perturbation and maintained property/functionality of a component prior deciding about its resilience. For example, a non-preemptive scheduler can be considered resilient if it can maintain desired properties of execution, such as hard-deadlines and tasks' precedences, despite slight deviations in the worst-case execution time (WCET) estimations for the tasks.

Resilience in the context of complex adaptive systems [20, 23, 22] has a more wider meaning and often becomes a synonym for robustness i.e., the capability to withstand perturbations. We shall however adhere to the more specific definition given above; we regard resilience as one of the three pillars of robustness, the other two being adaptability and recovery.

Resistance is closely related to resilience. According to Walker et al. [23], resistance is the measure of relationship between physical stress and perturbation size.⁵ For example, we can talk about resistance of insulation material used on a communication medium and its effect on error-rate of transmission. The error-rate of a transmission is then a perturbation for a communication network interface (CNI) and subsequently for the component. The component itself might be to some degree resilient to data errors in a message. The resistance only limits the rate and extent of the errors.

No-as-an-answer design is a design paradigm which follows the assumption that a required service might not be momentary available. Each component is

⁵Walker et al. talk about resilience in ecosystems and use it as a synonym for robustness. Although we use a different definition, the arguments they provide about resistance and its relationship to resilience are still useful.

to be designed such a way that it can cope with a temporary lack of a service – a “no” answer. This paradigm is advocated by Gribble [15] and Candea et al [6].

Components’ resilience to the temporary lack of service is vital for *recovery* (see Section 7). During the process of recovery, the affected component cannot deliver its service and the rest of the system must deal with this situation without breaking down.

5.1 Cognitive Resilience

Cognitive resilience is the aptitude of human users of a service to compensate for syntactic and semantic errors in the supplied service.

Cognitive resilience is an interplay of *information redundancy* (i-redundancy) of the service and *a priori knowledge* and *expectations* about the service’s semantic. The expectations can be based on the a priori knowledge about the service, but often it is based on experience, that is, an *a posteriori knowledge*. The i-redundancy is needed so that a user can develop such an a posteriori knowledge. The i-redundancy compensates for minor perturbations, such as those in the syntax. The rates of errors which cannot be compensated for by i-redundancy must be small so that a user can experience a period of correct service long enough to learn the patterns. This *learning phase* is not needed with the a priori knowledge.

Nowroth et al. [36] made use of a model of human cognitive resilience to improve transient fault tolerance of a JPEG compressor unit. They used a *psycho-visual* model of a human perception of images to identify critical areas in the JPEG compressor digital design where soft errors would cause a critical deviation from the required service, that is, image quality. These critical areas were then candidates for *hardening*. The cognitive resilience here stems from the vast i-redundancy of an image and natural human understanding of what is depicted (the image semantics).

5.2 Resilience Based on a priori Knowledge

The dynamics of a *real-time object* follow physical laws for which we might possess a priori knowledge – a model. An *actuator* or a controller in a real-time computer system may be instrumented with such an a priori knowledge and use the information to (1) check the correctness of sensor outputs, (2) use this information to compensate for a temporary lack of sensor data, and (3) for *state estimation* [29].

A designer can create models about various aspects of the designed system and its environment, he or she is not limited to real-time objects. Therefore we can talk in general terms: a component equipped with a priori knowledge about a required service can be made resilient to temporary degradation of such service by using this information to compensate for the degradation.

In compliance to HOT (see Section 2.2), there is a tradeoff between fragility and robustness. On one hand, the model relieves the component from problems caused by momentary lack of the required service, but on the other hand, the model creates extra assumptions about the environment. If those assumptions cease to hold, the system may break down.

5.3 Quantifying Resilience

In some cases, we can quantify the resilience of a component. Often, the provision of resilience is application specific but some solutions may share common patterns which can be analyzed with a rigor. Eslamnour et al. [5] presented an approach for quantifying resilience of a *resource management* systems (e.g., a scheduler) against perturbations which cause variation in a performance feature under question.

In this approach, the component's *critical feature* ϕ is quantified and its variation is required to be kept in a continuous range $\langle \beta_{min}, \beta_{max} \rangle$. Further, the set of perturbations Π must be identified. This set contains the possible perturbations π_i which are usually vectors and which have an influence on the variation of ϕ . For each perturbation vector π_i , we try to establish the resilience of the component against this perturbation given a resource allocation μ . The resilience $r_\mu(\phi, \pi_i)$ "is given as the smallest collective variation in the values of perturbation parameter that will cause the performance feature ϕ to violate its acceptable variation." [5] Mathematically, the r_μ is given as the smallest *distance* in a n-dimensional space from initial vector π_i^{init} to a vector π_i^* which caused an unacceptable variation of ϕ . Using, *Euclidean* distance is not satisfactory because the elements in the vector π_i likely have different degrees of *variation*. Also, the vector's elements might be correlated, therefore Eslamnour et al. opted for *Mahalanobis* distance. *Mahalanobis* distance can incorporate the desired probabilistic properties (i.e., variation and correlation) of the individual vector elements.

This approach of Eslamnour et al. is not limited to resource management systems but can be used whenever: (1) the critical property/feature can be quantified and its required values must be kept in a continuous range, (2) the perturbations, too, can be quantified and their effect on the variation of the critical property is known, and (3) we can estimate the variation and correlation of the elements of a perturbation vector.

5.4 Modeling Resilience of Composed Timed Systems

The research groups Fouchal et al. [12], Tarhini et al. [46], and Rollet et al. [39] focused on formal modeling of resilience in real-time component-based systems. The behavioral specification of a component is described by a variant of *timed automaton* called *timed labeled transition system* (TLTS) (for details about TLTS and the original timed automata see [46, 3] respectively). The authors work with two models specifying the component's (1) nominal behaviour, (2) and the degraded one. The nominal specification expresses the intended and error-free behaviour of the component whereas the degraded specification places the limits on the degradation of the behaviour in case of errors in the component's environment.

The resilience of a component is evaluated based on *hazard injection* during the components' execution and comparison of the resulting behavior against the specification of the degraded behaviour. The hazards are identified semi-automatically from the nominal specification. There are five kinds of hazards:

1. Replacing an input action – tests an response to an unexpected input for a component.

2. Changing the instant of an input action occurrence – tests perturbation in timing due to which a requested service (output) of another component may be delayed or received prematurely.
3. Exchanging two input actions – tests problems which arise from troubles in scheduling.
4. Adding unexpected transitions – a component receives additional unexpected input (this may be caused, for example, by some troubles with sensors).
5. Removing a transition – tests a loss of information in a system, for example due to problems with communication channels.

All possible hazards are automatically generated from the nominal specification and a designer assists the program to choose particular hazards for testing. From the set of chosen hazards the test-bench program automatically generates the test sequences. A component is said to be robust under a given set of test sequences if its observed behaviour did not violate its formal specification of the degraded behaviour.

6 Adaptability

Adaptability is the capability of a system to change its behavior or physical or logical structure in order to compensate for external or internal perturbations and changing requirements. Typical example is the Internet’s routing protocol which can cope with link and node failures by changing, in a distributed manner, the routing tables thus maintaining the network’s connectivity.⁶

The change of a system’s structure is also called *reconfiguration* and the new state of the system after the change is called a *configuration*. In the context of complex adaptive systems, we use the terminology of *chaos theory*. A configuration which is stable (resistant and resilient to perturbations) is then called an *attractor* (see for example [27]). Reconfiguration is the process of leaving one attractor and “falling” into another one. Adaptability is then the capability of a *dynamic* system to exchange an old attractor for a new one in which the system, under present conditions, achieves a better stability. More precisely, an attractor is “a point or an orbit in the phase space where different states of the system asymptotically converge”, where the *phase space* is “a multi-dimensional space that represent the dynamic of a system” (both quotes taken from [27]). Albeit very interesting and inspirational, the chaos theory and the related terminology are not, in our opinion, directly applicable in the context of complex computer system. It is because the phase space is often not known or it is not continuous. In the current state of the art, a computer system, in order to change its configuration, must undertake very specific actions rather than *evolve* through a phase space. The dynamism of the computer system is not in evolving through a phase space (e.g., a design space) but in the function of the system. Notable exception are self-organizing systems which we shall mention in Section 6.4.

⁶Here, the critical feature is connectivity and the perturbations are link and nodes failures.

During an adaptation⁷, parts of a system may encounter a degradation of a required service, hence the adaptability may require a resilience (see Section 5). Also, because of changing the system’s structure, some of the components may be removed from and others integrated into the system. The reintegration of component is supported by *recovery*, hence adaptability may require recovery (see Section 7).

In computer systems, adaptability covers a wide range of mechanisms often covered by the term *dynamic reconfiguration*. We have already mentioned the routing mechanism in Internet, another example is reconfiguration in communication schedules and dynamic resource management [17]. Kubisch et al. [32] pointed out that, unlike to natural systems, the adaptation in computer systems is reversible – they proposed an FPGA-based system with (1) a component which analyzed the system performance and suggested new configurations, (2) a repository which kept track of previous stable configurations of a system, and (3) a monitoring component which had the authority to decide about fitness of new and previous configurations and possibly to revert the system to an older and fitter configuration.

In the following, we shall overview the recent results of the research in principles of adaptation with stress on real-time systems (or complex computer systems in general). We start with a short discussion about diversity and heterogeneity and their influence on adaptation.

6.1 The Role of Diversity and Heterogeneity

In ecosystems, the *genetic* information diversity among species and phenotypical diversity among the individuals of the same species serve two main purposes. First, the diversity diminishes a *common mode failure* (CMF), for example, a highly homogeneous ecosystem lacking sufficient degree of genotypic diversity may be completely destroyed by a single virus. Second, the diversity “probes” the phase space of a system [23]. During a stable period of a system, the individuals in each new generation increase the overall diversity thanks to genetic cross-over and mutations and changing environmental conditions. The systems’ diversity creates and leaves open *strategic options* [19] into which the system can reconfigure in the case of strong perturbations (catastrophic events) such as floods, fires, viruses, etc. Because of the diversity, some individuals are more fit to survive a fire and some are more fit to survive a flood, together increasing the overall chances of the species to survive.

Naturally, the diversity in computer system (e.g., achieved through a 2-version programming) also decreases the chances of common mode failures. Some computer systems may also have, for example, several contingency mechanisms – which qualitatively differ from each other – to fulfill a given task. These mechanisms (e.g., algorithms) create then the open strategic options for the system and the system may choose the more fit of them to carry out the given task. We could regard it as a phase space probing, having on mind that no metaphor should be stressed too far – the diversity in computer systems is often static in contrast to ecosystems.

⁷For clarity, adaptability is the capability of a system to adapt, the adaptation is the actual action or process of adapting.

6.2 Modeling Adaptation

Trapp et al. [47, 48] focused on modeling a system’s runtime adaptation in the context of safety-critical systems. They propose a modeling language, similar to UML, which expresses a component’s adaptation in terms of required service quality and opting for an appropriate algorithms. The authors focus on behavioral adaptation rather than changing the system structure.⁸

The basic concepts in this modeling language are *modules* and *components*. Modules are the atomic blocks containing the actual specification of functionality. Components are containers consisting of other components or modules. Modules communicate with each other by signals (an analogy to a service in the terminology of this report). The signals are attributed by a signal value and a *quality*. The possible values of a signal quality and the semantics of it are defined by a *quality type* Q_t . A designer is responsible for identifying the various Q_t for signals. The identification itself is based on the requirements the signal value must meet.

A module has a pre-configured set of *behavioral variants* (configurations) and opts among them based on the availability of the signals and their qualities. The quality of an output signal of a module is determined by the used behavioral variant. Classes of the variants which induce the same output signal quality are called *modes*. A configuration of a module is defined by

1. the specification of the behaviour,
2. a guard (i.e., predicate) defining the conditions under which the configuration can be activated,
3. a priority,
4. function which determines the output signals qualities, and
5. a set of parameters which can be used to achieve a fine-grained adaptation of the behavior.

The *guard* is a function from a *quality domain*⁹ to the set {true, false}. When the guard evaluates to *true*, the configuration is said to be *enabled*. A module chooses a particular configuration from the set of enabled configurations based on the *priorities*.

The proposed modeling language supports formal verification. Namely, the guard predicates can be analyzed in order to determine that a system eventually takes up a particular configuration or that it will never enter certain configurations.

The adaptability modeled by the presented language is static in the sense that all the configurations are known during design time (compare, for example, with Internet routing or self-organization which is described in Section 6.4). However, in safety-critical system where verification and *certification* are crucial parts of the development, this kind of “static” adaptability cannot be ruled out. We present an example of an adaptation mechanism for which this modeling language would be a natural choice.

⁸If changing the structure of a system can be viewed as a behavior, then the proposed modeling language is applicable.

⁹A Cartesian product of sets of possible values of signals’ qualities.

6.2.1 Contingency Schedules – An Example of Static Adaptability

Kandasamy et al. [26] proposed a technique for an adaptation of task schedules in safety-critical embedded systems in order to compensate for transient faults caused, for example, by Heisenbugs. The basic principle of the proposed mechanism is a task re-execution, that is, a *time redundancy*. For each task, the system’s scheduler keeps one or more contingency schedules for the case that an error in the task execution is detected. If the error is detected, the scheduler executes the appropriate contingency schedule for the remainder of the affected cycle in which the task is re-executed. Contingency schedules are statically precomputed and they must preserve timing and precedence constraints. These constraints are the critical feature in this case and the perturbations are transient faults which affect tasks’ executions.

In the modeling language presented above, the scheduler can be seen as a module the input signals of which would be the tasks’ heartbeats or possibly an output from an error detection unit. The guards would watch for tasks’ failures and activate the precomputed schedules.

6.3 Adaptation in Communication

Gershenson et al. [14] discuss an option for components to “negotiate” communication semantics at run time. That is, the components lack a designed-in understanding of the message semantics but develop this understanding by playing *language games* among each others.

Such a communication paradigm supports the system’s extensibility across a wide time range. It eases the assumptions about backward compatibility of the system’s extensions. The critical property here, is the components’ compatibility and the extensibility of the architecture. The architecture is made robust against the perturbations stemming from ambiguity in a specification and its aging caused by inevitable demands for new functionality.

6.4 Self-organization

A system where a higher organization or a function *emerges* solely from local interactions of its elements without an explicit centralized control is called *self-organizing*. We talk about emergence whenever the organization or the function under discussion cannot be easily predicted (or even explained) by a *reductionist* approach.

It cannot be stressed enough that *organization* and *function* are system properties determined by an *observer* of the system or the user of the function [13] (we shall use the term observer to cover both the user and the observer). The observer determines whether a system shows an organization or not, and it is only the observer’s responsibility if the term self-organization will in the end cover *every* physical phenomenon, since it is the nature of physical phenomena to interact locally. We can paraphrase Holland (see Introduction) and say that we are not usually interested in self-organization of pile of rocks. Often in the studies of self-organizing systems [16], the organization or function is observed in the first place and later explained through the nature of local interactions (e.g., a fish shoal, a bird swarm, anthills, etc.). When designing a self-organizing system, we go the other way round: we have a *goal in mind* and we are trying to deduce

the appropriate rules for system elements' local interactions. Unfortunately, there is still lack of design guidelines how to do that.

If the system's function, which emerges through self-organization, is for us the *critical function*, then it is the nature of self-organizing systems that they can adapt to changing environmental and internal conditions and most importantly maintain this critical function. Gershenson, in his PhD thesis [13], successfully designed simple local rules for a city traffic lights such a way that the lights self-organized their phases and increased the average traffic throughput and decreased waiting times at crossroads. Such system was able to adapt to changing environmental conditions, namely failures of individual traffic lights and the fluctuations in traffic loads during the day, week, etc.

7 Recovery

Recovery of a component consists of *repairing* (i.e., healing in biological terminology) and *reintegration* of the component back to the system. The term recovery, as we see it in this report, is related to system's components, not the system's function, property, or the like.¹⁰ Our viewpoint here is structural; the main questions we aim to answer by studying recovery are

1. If a component is in an *erroneous* state, how can we detect the error and bring the component back to a correct state?
2. What are the correct states that support reintegration of the component and what are the states that rule it out?

The recovery is not well covered in the study of complex adaptive systems (e.g., ecosystems, economies, etc.) where the effects of healing an individual component are overweighted by the relatively high rate of creation and loss of components (*birth/death* rate). However, the healing process is often studied per se again as a complex adaptive system, for example *immune systems* of organisms, or *artificial immune systems* in computers – for the later see Forrest et al. [25] or our report on intrusion detections [35].

For the aforementioned reasons, we shall focus on recovery in the context of complex computer systems. In the rest of this section we shall use the term *node* and *component* interchangeably. Under the light of the arguments given in the introduction to this report, we could clearly see that computer components face increasingly high rates of various transient faults. It follows, that we must simply assume that components are bound to fail. *Were it not for recovery, each such a failure would be permanent.* The provision of resilience (see Section 5) copes only with *temporary* degradation or lack of a service, hence recovery is needed to bound the time periods of service shortages – recovery supports resilience. Also, any attempt to recover a component would be pointless if the rest of the system could not cope with the temporary unavailability of the very component being recovered, therefore the rest of the system must be resilient – the recovery requires resilience. The typical recovery is summarized in [29]:

The first step following the occurrence of a node failure is a self-test of the node hardware. If the self-test is successful, then, it can be

¹⁰At least not directly – the system's function is of course brought about by its components.

assumed that a transient fault led to the failure of the node, since a transient fault afflicts no permanent damage to the node hardware. The reintegration of the node can be started immediately.

Since now on, we shall tacitly assume that an erroneous state of a node was caused by a transient fault and that a recovery is possible. Also, instead of “repairing”, we shall directly use the term recovery.

7.1 Recursive Restartability

Restarts or *reboots* will always be effective methods to work around node failures caused by Heisenbugs, deadlocks, memory leaks, and other transient faults. A reboot brings the affected node to its initial state which is often the most studied and understood state of the node. The reboots allow for reclaiming stale resources and clean up corrupt state.

In their very inspirational paper [6], Candea and Fox coined the term *recursive restartability* (RR) which is the capability of a system to tolerate gracefully restarts on multiple levels of its hierarchy. We can also define an RR system recursively: The simplest, base-case RR system is a *restartable* atomic component (atomic in the sense of fixed level of abstraction chosen by a designer, for example, a micro component [40]); a general RR system is a composition of RR systems that obeys the guidelines described below in Section 7.1.1.¹¹ In their interesting paper [7], Candea et al. successfully applied the technique of recursive restartability to an Internet application.

An unannounced restart of a component is seen by the rest of the system as a temporary failure. Therefore, systems that are designed to tolerate such restarts¹² are inherently tolerant to all transient non-Byzantine failures [6]. The authors further pointed out, that the fine granularity of recursive restartability allows for a bounded portion of the system to be restarted upon a failure. This way, the restarts are “cheaper” because the full reboots are replaced with partial restarts. Partial restarts therefore increase availability. Moreover, the system has more strategic options with regards to the choice of level where it should perform a partial restart. The closer to the root component (i.e., the whole system) the more expensive (time demanding) the restart, but the higher the confidence that the restart will eventually help. In the terminology of [6], the partial restart create a *confidence continuum* for restarts. The system recovery mechanism must then tradeoff between the confidence in restart success and the costs.

7.1.1 Design guidelines for RR systems

Candea and Fox further proposed a set of design guidelines for building RR systems. The guidelines were originally proposed in the context of distributed soft-real-time, non-safety-critical application such as web application.

Accepting *No* for an answer First of all, the components should accept *No* for an answer, because the part of the affected system that is being restarted

¹¹The recursive definition is taken from [6] and slightly modified.

¹²Such systems are indeed discussed in Section 5 where we talk about resilience.

cease temporarily to deliver its function. We have covered this topic in resilience in Section 5.

Soft state and announce/listen protocols Next, the shared state among subsystems should be mostly *soft*. A shared state among subsystems is called soft, if the two subsystems are the only maintainers of the state and when they fail, the state is lost too. For the discussion of a soft state see, for example, [22]. Also, the component should adhere to announce/listen protocols which makes the default assumption that a component is unavailable unless it says otherwise.

Trading precision and consistency for availability A system should automatically trade precision or consistency for availability. In the words of [6]: “Inter-component ‘glue’ protocols should allow components to make dynamic decisions on trading consistency/precision for availability, based on both application-specific consistency/precision measures, and a consistency/precision utility function.”

Fine grain workloads An application should be structured around fine grain workloads such that the interactions among component are also fine grain (i.e., short-lived). Such design supports quick partial restarts and “leads directly to the simple replication and failover techniques found in large cluster-based Internet services.”[6]

Using orthogonal composition axes Independent subsystems that do not require an understanding of each other’s functionality are said to be orthogonal. Compositions of orthogonal subsystems exhibit high tolerance to component restarts. There is a strong connection between good modular structure and the ability to exploit orthogonal mechanisms; systems that exploit them well seem to go even further: their control flows are completely decoupled, influencing each other only indirectly through explicit message passing. Examples of orthogonal mechanisms include deadlock resolution in databases, software based fault isolation, as well as heartbeats and watchdogs used by process peers to monitor each other’s liveness.¹³

7.2 Ground State and its Monitoring

Ground state of distributed system, be it synchronous or asynchronous, was defined in [2]: a *ground state* is a global state in which all channels are empty. The definition of a global state is somewhat more complicated [2] (see below). It should be noted that in the model of a distributed system given in [2], a node changes its state only upon a reception or a sending of a message.

A *global state* is a set of states (1) one for each node, such that every message that a node receives before it reaches its state would have been sent by the sender of that message before it reached its state; (2) one for each channel, as a set (sequence for a FIFO channel) of messages sent along that channel before the sender node reached

¹³The whole paragraph is taken from [6].

its state less the messages received by the receiver node before it reached its state.

For clarity, the part (1) of the last definition says basically that a node’s state which can be fully determined only based on future states of other nodes must not be part of the global states. The advancement of the system is fully determined by its global state, that is, states of the components and the channels. Most importantly, the progress of a system which is in a ground state is fully determined by the states of the nodes solely. The ground state is therefore an ideal point of observation.

Kopetz [29] applied the definition of a ground state to a single component: “the ground state of a node in a distributed system at a given level of abstraction [is] a state where no task is active and where all communication channels are flushed, i.e., there are no messages in transit.” The node’s ground state is an ideal point of reintegration, it is therefore desirable that a component visits its ground state as often as possible, preferably at a priori known points in time. A node’s ground state may not exist, it is the designer’s responsibility to design for a ground state.

A ground state is an excellent reintegration point of a component because the state of the component is minimal – it is contained in the visible data structures and the program counter [29]. The ground state is a very suitable object for monitoring correctness of a component. Kopetz [30] proposed a ground state monitor (GSM) component which has the authority to monitor and decide about a correctness of a ground state which the monitored component periodically outputs in a message. The GSM component would be typically equipped with an application specific knowledge (i.e., an a priori knowledge about a behavior of the monitored component) and check for syntactic and semantic correctness of the externalized ground state. In the case of a detected component’s failure, the GSM is authorized to repair the component, for example by a reboot. If the ground state is empty, the only semantics is the very reception or absence of the ground state message; in this case the error detection and reintegration is trivial. However, if the ground state is not empty, the GSM has to store the last ground state and load it into the component upon a restart.

8 Conclusion

We have pointed out the reasons why the study of robustness is justified and more and more needed. Current complex computer systems are often fragile despite all the efforts put in to the “the-right-thing” design. The ultimate goal of the robustness research is to come up with a set of good “high-level design principles that guide the technical design” [22]. We have overviewed some of the well-established and well-tested ones such as modular structure and hour-glass model. The robust design should in the end unify these good practices and bring new ideas. In this report, we have identified resilience, adaptability, and recovery as three main pillars of robustness and discussed their roles and cornerstone principles.

Resilience, adaptability, and recovery is a self enforcing trinity of concepts. Resilience copes with temporary lack or degradation of service. The lack of service would not be temporary, were it not for recovery. Recovery is the capability to recover a component and reintegrate the component and its service

back to the system after an error is detected. Recovery of a node would not be of any use, if the rest of the system could not cope with the temporal shortage of the component's service, therefore recovery is best accompanied with resilience. Adaptability of a system ensures that a system can handle severe changes in the environment or its internal structure. The means of adaptability can cause a short-term lack of some services, thus adaptability requires resilience to compensate the service shortages. Furthermore, in order to adapt it might be necessary to disconnect some components from as well as integrate new ones into the system, therefore adaptation requires recovery because recovery entails component (re)integration.

A Remarks on Robustness in Real-Time Systems

In the introduction to this document, we have pointed out that the research of robustness occupies the niche between rigorous and uncertain (or undefined). Consider an adaptation, it is often a process of choosing a system's configuration which was not actually predicted; indeed, that is the case of ecosystems. In safety-critical real-time systems, however, we cannot usually afford even a slim level of uncertainty or unpredictability, due to possible catastrophic outcome in the case of a severe failure. Also, the certification is a part of designing such systems and the designer must be able to provide guarantees about the systems dependability. All these constraints make the research of robustness very interesting and challenging.

Robustness of Robust Design Principles In the section "Assumption, Constraints, and Models", we have pointed out that a robust design guidelines will necessarily abandon, or at least weaken, some typical assumptions often made about the environment, nature of faults (namely their independence), and others. In summary, the smaller the number of optimistic assumptions we make about the system and its environment, the smaller the number of unexpected events the system may face, and in consequence, the system is more robust. Consider, for example, a partially synchronous real-time system, where the timing assumptions are less stringent

But we have also pointed out, that designers make these assumptions in order to build simplified models about the environment or the system itself (e.g., by introducing *sparse time base* [29]). Therefore, there is a crucial tradeoff between *design complexity* and the *assumption extent*. Cognitive complexity of systems and designer's fallibility are causes of critical system failures. We argue, that it could be practical to also talk about a *robustness of design guidelines* which would express the resilience of the guidelines against designers' fallibility. The guidelines would be called robust if the resulting system is also robust despite overworked engineers, demands for short time to market, and the like.

Convergent Algorithms Here, we would like to argue for a usage of methods, or algorithms, which gradually *converge* to a result, rather than *solvers* which output a correct result just once when they finish their execution. We see two main reasons which support usage of convergent algorithms: (1) they

weaken timing assumptions (i.e., deadlines) and (2) they can possibly create shorter periods in which a component enters its ground state.

(1) The usage of convergent algorithms follows the Candea and Fox’s guideline (see Section 7.1.1) which calls for trading precision for availability. If a convergent algorithm was about to miss a deadline and be preempted, its incomplete, yet close-enough approximation of the result could still be used. After all, in robust design, we expect the rest of the components to be resilient to a certain degree of a service degradation. With solvers, no partial approximation of a result is available before the finish of the program. In the ideal case, a convergent algorithm would be proven to provide a precise result after N number of steps and its bounds on error would be also known for $M < N$ steps.

(2) Convergent algorithms operate in cycles and at the end of each cycle, the algorithm has in hand a better approximation of the final result. One evolution of the cycle can be seen as an *atomic operation*; once such atomic operation is completed, the component is in a ground state (given that other conditions of the ground state are also met – such as flushed communication channels). If it takes N atomic operations for a convergent algorithm to compute a result, then in comparison to a solver, the convergent algorithm creates N -times more opportunities for the component to enter a ground state. We could also say that convergent algorithms have a finer workloads, which is another recommendation of Candea and Fox.

An Experienced Actuator In the context of Section 5, we have mentioned that a cognitive resilience is an interplay of a priori knowledge and expectations. In its simplistic form, the expectations can be viewed as extrapolations of experiences i.e., predictions based on an a posteriori knowledge. A *surprise* is the degree to which the predictions differ from an observed reality. In this short remark, we propose a technique which enables a component to *learn* and to predict. We illustrate this technique on an actuator component.

An actuator could be equipped with a unit which would observe an input of the actuator and learn the input’s patterns. Such unit would not have the authority to decide about correctness of the input but it could provide the actuator with predictions during a shortage of the input, thus making the component resilient to a temporary lack of service. Such technique would require either fail-silent components with fast recovery or a satisfactory error detection such that the actuator could reliably decide when to opt for predictions. Nowadays, the lack of service is overcome by freezing the last position of actuator. We argue, that the proposed unit could decrease the “surprise” of an actuator once the service is again available and consequently increase the period of time during which an actuator lacks the required service and still tolerates it.

Recent results in the research of *artificial neural networks* showed some promising results. Leclercq et al. [33] invented an efficient autonomous learning algorithm for fully connected recurrent networks. This type of neural networks has a very good prediction capabilities for dynamic non-linear phenomena. Moreover, if started from zero initial conditions, the learning algorithm does not need any initialization. A unit equipped with a recurrent neural network and autonomous learning mechanism could provide an actuator with “experiences” and predictions as described above.

Acknowledgment Many thanks go to Christian El-Salloum for his review and valuable comments on the structure and contents of this work.

References

- [1] Santa Fe Institute.
<http://www.santafe.edu>.
- [2] M. Ahuja, A.D. Kshemkalyani, and T. Carlson. A basic unit of computation in distributed systems. *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 12–19, 28 May–1 Jun 1990.
- [3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [4] 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.
- [5] Eslamnour Behdis and Shoukat Ali. A probabilistic approach to measuring robustness in computing systems. *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 26–30 March 2007.
- [6] 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.
- [7] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando 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.
- [8] J. M. Carlson and John Doyle. Highly optimized tolerance: A mechanism for power laws in designed systems. *Phys. Rev. E*, 60(2):1412–1427, Aug 1999.
- [9] B. Carpenter. Architectural Principles of the Internet. RFC 1958 (Informational), June 1996. Updated by RFC 3439.
- [10] D. Clark. The design philosophy of the darpa internet protocols. *SIGCOMM Comput. Commun. Rev.*, 18(4):106–114, 1988.
- [11] Marie E. Csete and John C. Doyle. Reverse Engineering of Biological Complexity. *Science*, 295(5560):1664–1669, 2002.
- [12] Hacne Fouchal, Antoine Rollet, and Abbas Tarhini. *Lecture Notes in Computer Science*, chapter Robustness of Composed Timed Systems, pages 157–166. Springer Berlin / Heidelberg, 2005.
- [13] Carlos Gershenson. *Design and Control of Self-organizing Systems*. PhD thesis, Vrije Universiteit Brussel, 2007.

- [14] Carlos Gershenson and Francis Heylighen. Protocol requirements for self-organizing artifacts: Towards an ambient intelligence, 2004.
- [15] S.D. Gribble. Robustness in complex systems. *Hot Topics in Operating Systems*, pages 21–26, 20-22 May 2001.
- [16] Francis Heylighen. The science of self-organization and adaptivity, 1999.
- [17] Bernhard Huber. *Resource Management in an Integrated Time-Triggered Architecture*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2008.
- [18] Erica Jen, editor. *Robust Design*, chapter Introduction, pages 1–6. Oxford University Press, 2005.
- [19] Erica Jen, editor. *Robust Design*, chapter Stable or Robust? What’s the Difference?, pages 7–20. Oxford University Press, 2005.
- [20] Erica Jen, editor. *Robust Design*, chapter Cross-System Perspectives on the Ecology and Evolution of Resilience, pages 151–172. Oxford University Press, 2005.
- [21] Erica Jen, editor. *Robust Design*, chapter Robustness and the Internet: Theoretical Foundations, pages 273–285. Oxford University Press, 2005.
- [22] Erica Jen, editor. *Robust Design*, chapter Robustness and the Internet: Design and Evolution, pages 231–271. Oxford University Press, 2005.
- [23] Erica Jen, editor. *Robust Design*, chapter Robustness in Ecosystems, pages 173–190. Oxford University Press, 2005.
- [24] Erica Jen, editor. *Robust Design*, chapter Directing the Evolvable: Utilizing Robustness in Evolution, pages 105–134. Oxford University Press, 2005.
- [25] Erica Jen, editor. *Robust Design*, chapter Computation in the Wild, pages 207–230. Oxford University Press, 2005.
- [26] N. Kandasamy, J.P. Hayes, and B.T. Murray. Tolerating transient faults in statically scheduled safety-critical embedded systems. *Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on*, pages 212–221, 1999.
- [27] Hiroaki Kitano. Biological robustness. *Nature*, 5(11):826–837, Nov 2004.
- [28] 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.
- [29] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston: Kluwer Academic Publishers, 1997.
- [30] Hermann Kopetz. Reliable services in an imperfect world. DATE08 Conference, Keynote talk, 2008.

- [31] Hermann 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.
- [32] S. Kubisch, R. Hecht, R. Salomon, and D. Timmermann. Intrinsic flexibility and robustness in adaptive systems: A conceptual framework. *Adaptive and Learning Systems, 2006 IEEE Mountain Workshop on*, pages 98–103, 24-26 July 2006.
- [33] Edouard Leclercq, Fabrice Druaux, Dimitri Lefebvre, and Salem Zerkaoui. Autonomous learning algorithm for fully connected recurrent networks. *Neurocomputing, New Aspects in Neurocomputing, 11th European Symposium on Artificial Neural Networks*, 63:25–44, 2005.
- [34] Sharad Malik. A case for the runtime validation of hardware. IBM Verification Conference, Haifa, Keynote talk, 2005.
- [35] Vaclav Mikolasek. Intrusion detection systems - state of the art report. Research Report 21/2008, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2008.
- [36] Damian Nowroth, Ilia Polia, and Bernd Becker. A study of cognitive resilience in a JPEG compressor. To be published in DSN June, 2008.
- [37] M. Pflanz and H.T. Vierhaus. Online check and recovery techniques for dependable embedded processors. *Micro, IEEE*, 21(5):24–40, Sep/Oct 2001.
- [38] Karl R. Popper. *Unended Quest; An Intellectual Autobiography*. Routledge, 2002.
- [39] Antoine Rollet and Fares Saad-Khorchef. A formal approach to test the robustness of embedded systems using behaviour analysis. *Software Engineering Research, Management & Applications*, pages 667–674, 20-22 Aug. 2007.
- [40] Christian El Salloum, Roman Obermaisser, Bernhard Huber, Harald Paulitsch, and Hermann 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.
- [41] 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.
- [42] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks*, 2002.
- [43] Herbert A. Simon. *The sciences of the artificial (3rd ed.)*. MIT Press, Cambridge, MA, USA, 1996.
- [44] J. Sosnowski. Transient fault tolerance in digital systems. *Micro, IEEE*, 14(1):24–35, Feb 1994.

- [45] ARTEMIS strategic research agenda working group. Strategic research agenda reference, designs and architectures. <http://www.artemis-office.org>, 2006.
- [46] A. Tarhini and H. Fouchal. Robustness evaluation of real-time protocols. *Telecommunications, 2006. AICT-ICIW '06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 95–95, 19-25 Feb. 2006.
- [47] Mario Trapp, Rasmus Adler, Marc Förster, and Janosch Junger. Runtime adaptation in safety-critical automotive systems. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference*, pages 308–315, Anaheim, CA, USA, 2007. ACTA Press.
- [48] Mario Trapp and Bernd Schrmann. On the modeling of adaptive systems. <http://citeseer.ist.psu.edu/669688.html>.