Handling Software Faults with Redundancy*

Antonio Carzaniga[1], Alessandra Gorla[1], and Mauro Pezzè[1,2]

[1] University of Lugano, Faculty of Informatics,
via Buffi 13, 6900, Lugano, Switzerland
antonio.carzaniga@usi.ch, alessandra.gorla@usi.ch
[2] University of Milano-Bicocca, Dipartimento di Informatica, Sistemistica e Comunicazione, Via Bicocca degli Arcimboldi 8, 20126, Milano, Italy
mauro.pezze@usi.ch

Abstract. Software engineering methods can increase the dependability of software systems, and yet some faults escape even the most rigorous and methodical development process. Therefore, to guarantee high levels of reliability in the presence of faults, software systems must be designed to reduce the impact of the failures caused by such faults, for example by deploying techniques to detect and compensate for erroneous runtime conditions. In this chapter, we focus on software techniques to handle software faults, and we survey several such techniques developed in the area of fault tolerance and more recently in the area of autonomic computing. Since practically all techniques exploit some form of redundancy, we consider the impact of redundancy on the software architecture, and we propose a taxonomy centered on the nature and use of redundancy in software systems. The primary utility of this taxonomy is to classify and compare techniques to handle software faults.

1 Introduction

This work addresses the engineering of software systems that are used in the presence of faults. Arguably, despite mature design and development methods, despite rigorous testing procedures, efficient verification algorithms, and many other software engineering techniques, the majority of non-trivial software systems are deployed with faults. Also, in practice, computing systems cannot exist in isolation as purely mathematical objects, and therefore are inevitably affected by faults. For these reasons, we accept the assumption that many systems can not be completely rid of faults, and that the reliability of such systems can be improved by allowing them to *prevent or alleviate the effects of faults*, and perhaps even to correct the faults at runtime. These are essentially the goals of much research in the area of fault tolerance [1,2] and more recently in autonomic computing [3,4].

There are important differences between the approaches to reliability found in the fault tolerance and autonomic computing literature, respectively. First, at

* This work was supported by the Swiss National Science Foundation under the *PerSeoS* and *WASH* projects.

R. de Lemos et al. (Eds.): Architecting Dependable Systems VI, LNCS 5835, pp. 148–171, 2009.

a high level, fault tolerance is a more focused area, while autonomic computing covers a larger set of objectives. In fact, the term autonomic computing refers to the general ability of a system to respond to various conditions such as performance degradation or changes in the configuration of the environment, many of which may not be caused nor affected by faults. In this chapter we will limit ourselves to a sub-area of autonomic computing, typically called self-healing, whose specific objective is to achieve an automatic reaction to, or prevention of functional faults and failures.

Another difference is in the nature of the intended application domain. fault tolerance research has been driven primarily by highly specialized and safety-critical systems, whereas autonomic computing—specifically, self-healing—is targeted towards general purpose components or loosely coupled systems where the effects of failures are less destructive. These two application domains also have significant differences in the levels of costs and ultimately in the type of designs that are considered acceptable.

Yet another difference is that fault tolerance research explicitly addresses both hardware and software faults with different techniques that may be hardware or software, while self-healing research does not distinguish different classes of faults and has so far studied mostly software techniques. Finally, classic fault tolerance approaches have a stronger architectural implications than many recent self-healing approaches.

Their differences not withstanding, fault tolerance and self-healing in autonomic computing share the same goal of rendering software systems immune or at least partially resilient to faults. Therefore, in this chapter we propose to unify the contributions of these two somewhat separate research areas in a coherent classification. In particular, we propose to focus on what we see as the central, common element of most of the techniques developed in both communities, namely *redundancy*. We will pay particular attention to techniques that handle software faults, and to their architectural implications.

A system is redundant when it is capable of executing the same, logically unique functionality in multiple ways or in multiple instances. The availability of alternative execution paths or alternative execution environments is the primary ingredient of practically all systems capable of avoiding or tolerating failures. For example, a fairly obvious approach to overcome non-deterministic faults, such as hardware faults, is to run multiple replicas of the system, and then simply switch to a functioning replica when a catastrophic failure compromises one of the replicas. In fact, redundant hardware has been developed since the early sixties to tolerate development faults as well as manufacturing faults in circuits [5,6,7]. An analogous form of redundancy is at the core of many widely studied replication techniques used to increase the availability of data management systems [8]. Similarly, redundancy has been used extensively to tolerate software faults [1]. This paper focuses primarily on techniques that exploit software redundancy for tolerating such faults.

Software poses some special challenges and also provides new opportunities to exploit redundancy. For example, while simple replication of components can

handle some classes of production faults typical of hardware design, it cannot deal with many failures that derive from development and integration problems that occur often in software systems. On the other hand, software systems are amenable to various forms of redundancy generally not found in hardware systems. A form of redundancy for software systems that is analogous to basic replication techniques used for non-deterministic hardware faults is *N-version programming*. In this case, replicas run independently-developed versions of the same system, and therefore may also be able to tolerate deterministic faults [9]. As another example, consider a self-healing system that, in order to overcome deterministic faults, detects the faulty component and redirects every request for that faulty component to another, more or less equivalent component [10,11].

The availability of such a replacement component is a form of redundancy, and is also generally applicable to hardware systems. However, the nature of software components and their interactions may make this technique much more effective, in terms of time and costs, for software rather than hardware components. Yet another example is that of *micro-reboots*, which exploit another form of redundancy rooted in the execution environment rather than in the code. In this case, the system re-executes some of its initialization procedures to obtain a fresh execution environment that might in turn make the system less prone to failures [12,13].

Having centered our taxonomy on redundancy, we proceed with a high-level classification by examining the following four questions. We first ask what is the intent of redundancy. We ask whether a fault tolerance mechanism relies on redundancy that is deliberately added to the system, or whether the mechanism opportunistically exploits redundancy that occurs naturally in the system. For example, N-version programming is clearly a deliberate form of redundancy, while micro-reboots are opportunistic in nature, since they do not require the design of redundant code. This question of intent is correlated with the costs and effectiveness of a technique: Generally speaking, redundancy established by design is more effective but also more costly. This question is also important because it may highlight useful forms of latent redundancy, that is, forms of redundancy that, even though not intentionally designed within a system, may be exploited to increase reliability.

Then, we ask what type of redundancy is employed in a particular system. This question in effect identifies the elements of the execution of the system that are made redundant. The three high-level categories we distinguish are *code*, *data*, and *environment*, which follow quite closely the taxonomy proposed by Ammar et al., who introduce the concepts of *spatial*, *information* and *temporal* redundancy [14]. For example, micro-reboot employs redundancy in the execution environment, whereas N-version programming amounts to code redundancy.

Third, we look at how redundancy is used. Specifically, we ask whether it is used preventively, to avoid failures, or reactively to mask or otherwise compensate for the effects of faults. For example, the recovery blocks mechanism is reactive, while software rejuvenation is preventive [15]. In the case of methods that use redundancy reactively, we also explore the nature of the failure detectors

needed to trigger the reaction, and the corrective measures used in the reaction. We examine these two aspects especially in relation to redundancy.

The fourth question differentiates mechanisms according to the nature of the faults they are most effective with. We distinguish two large classes of faults: those that affect a system *deterministically* when given the same input vector, and those that are *non-deterministic* in nature, for instance because of some inherent non-determinism of the system or most likely its environment. Faults of these two classes are often referred to as *Bohrbugs* and *Heisenbugs*, respectively [16,17].

Fault tolerance and autonomic computing are well-established research areas. The former is arguably more mature, but the latter has received more attention in recent years, and both have been surveyed extensively. Most surveys of these two areas either focus on specific techniques and applications, or adopt general conceptual models and a historical perspective. In any case, existing surveys consider only one of these two areas. For example, Huebscher and McCanne review many techniques developed in the context of of self-managed systems [18] but do not relate them to fault tolerance. Specifically, they organize their survey around a foundational but rather high-level model of autonomic computing. At the other end of the spectrum, De Florio and Blondia compile an extensive survey of software fault tolerance techniques [19]. In particular, they discuss some techniques related to redundancy (for instance, N-version programming) but primarily they review domain-specific languages and other linguistic techniques to enhance the reliability of software systems at the application level. Another related survey is one by Littlewood and Strigini [20], who examine the benefits of redundancy—specifically *diversity*—to increase system reliability in the presence of faults that pose security vulnerabilities. Yet another example of a focused survey is the widely cited work by Elnozahy et al. on rollback-recovery protocols [21].

Particularly interesting in the scope of this paper are the taxonomies by Ammar et al. [14] and Avizienis et al [2]. Ammar et al. propose an extensive survey of the different aspects of fault tolerance techniques, and, in this context, distinguish *spatial*, *information* and *temporal* redundancy. This taxonomy focuses on the dimensions of redundancy, and matches well the differences of redundant techniques for handling hardware as well as software faults. The classification in terms of *code*, *data* and *environment* redundancy adapts better to different types of redundancy introduced in the context of fault tolerance as well as self-healing research to handle software faults. Avizienis et al. propose a fault taxonomy that has become a de-facto standard. In this chapter, we refer to Avizienis' terminology and taxonomy limited to the aspects relevant to software faults.

Our goal with this survey is rather different. First, at a high level, we take a broad perspective on this field and consider a wide range of techniques. Therefore we do not attempt to examine every technical aspect of each technique in great detail. Nevertheless, we make an effort to maintain a strictly technical approach in our classification, distinguishing methods and systems on the basis of their technical features and merits, and regardless of the conceptual model used to

describe them or the application domain for which they were developed. Second, we intend to unify the two areas of fault tolerance and self-healing under a single taxonomy, ignoring the historical path that lead to one technique or another, and instead basing our taxonomy on what we see as the essential, common ideas behind both fault tolerance and self-healing. Finally, we focus on techniques for handling software faults, and we discuss the different aspects as well as the overall architectural implications.

In Section 2 we discuss the main architectural implications of redundancy, and we identify few simple design patterns. In Section 3 we detail the classification method we used in our survey. We then proceed with the analysis of several systems, first in Section 4 where we discuss systems based on the deliberate use redundancy, and then in Section 5 where we discuss opportunistic redundancy.

2 Architectural Implications

Although implicitly discussed in different papers, the architectural implications of fault tolerance mechanisms have been explicitly addressed only recently. Gacek and de Lemos suggest that dependability should be designed at the architecture level, and define the requirements for embedding fault tolerant approaches into architectural description languages [22]. Feiler and Rugina enrich the architectural description language AADL with error annexes to model dependability at the architectural level. Harrison and Avgeriou discuss the implementability of fault tolerance tactics within different architectural patterns [23]. Hanmer presents an extensive set of patterns for error and fault handling [24]. In this section, we shortly illustrate the conceptual impacts of redundancy on software architectures.

From the software architecture viewpoint, redundancy can be introduced either at intra- or at inter-component level. When introduced at intra-component level, redundancy does not alter the structure of the connections between components, but only the single components. This is the case for example of wrappers that filter component interactions, robust data structures that introduce redundancy at the level of data structure to handle data related faults, and automatic workarounds that exploit redundancy that is implicitly present in the program.

When introduced at inter-component level, we can recognize few recurrent patterns that are shown in Figure 1. In the *parallel evaluation* pattern, an adjudicator evaluates the results of several alternative implementations that are executed in parallel. The adjudicator is often a simple voting mechanism that identifies failures. This is the case for example of N-version programming that executes N different versions with the same input configuration, and of data diversity and process replicas that execute identical copies with different input configurations. In the *parallel selection* pattern, the adjudicator is active at the end of each program executed in parallel to validate the result and disable failing components. The pattern is implemented in self-checking programming. In the *sequential alternative* pattern, alternative programs are activated when adjudicators detect failures. This pattern is implemented by recovery blocks,

self-optimizing applications, registry-based recovery, data diversity and service substitution approaches. We will discuss the different approaches in more details in the next sections when introducing a taxonomy for redundancy.

3 A Taxonomy for Redundancy

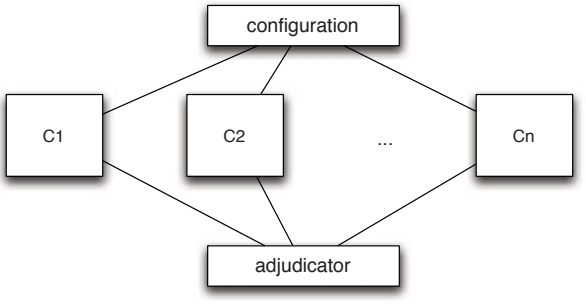
As discussed in the introduction, both fault tolerance and autonomic computing are well established research areas, and both have been surveyed extensively. Particularly interesting for this paper are the surveys by Ammar et al. [14], Elnozahy et al. [21], Littlewood and Strigini [20], and De Florio and Blondia [19], who survey approaches to fault tolerance and the use of redundancy in fault tolerance from different perspectives, and the work by Huebscher and McCanne who survey approaches to self-managed systems [18]. All these surveys focus on either fault tolerance or self-healing systems, and none of them suites well both research areas. The taxonomy proposed in this paper aims to provide a unifying framework for the use of redundancy for handling software faults both in fault tolerant and self-healing systems.

To survey and compare the different ways redundancy has been exploited in the software domain, we identify some key dimensions upon which we define a taxonomy of fault tolerance and self-healing techniques. These are the *intention* of redundancy, the *type* of redundancy, the *nature of triggers and adjudicators* that can activate redundant mechanisms and use their results, and lastly the *class of faults* addressed by the redundancy mechanisms. Table 1 summarizes this high-level classification scheme.

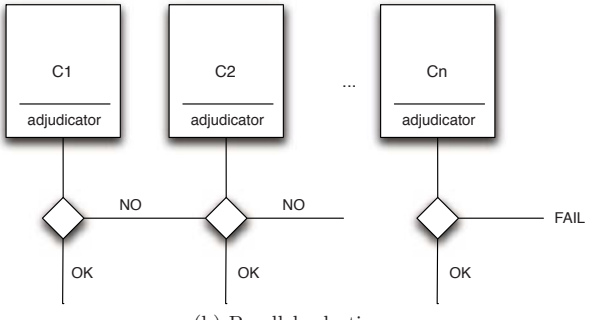
Table 1. Taxonomy for redundancy based mechanisms

Intention:	deliberate opportunistic
Type:	code data environment
Triggers and adjudicators:	preventive (implicit adjudicator) reactive: implicit adjudicator explicit adjudicator
Faults addressed by redundancy:	interaction - malicious development: Bohrbugs Heisenbugs

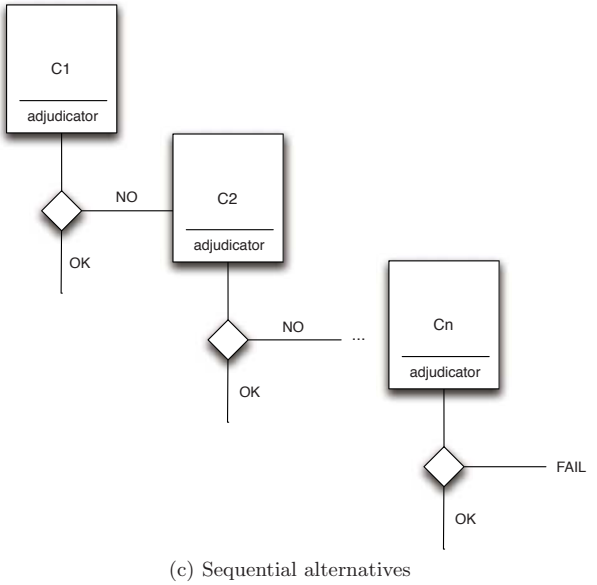
Intention. Redundancy can be either *deliberately* introduced in the design or implicitly present in the system and *opportunistically* exploited for fault handling. Some approaches deliberately add redundancy to the system to handle faults. This is the case, for example, of N-version programming that replicates the design process to produce redundant functionality to mask failures in single modules [9]. Other approaches opportunistically exploit redundancy latent in the



(a) Parallel evaluation



(b) Parallel selection



(c) Sequential alternatives

Fig. 1. Main architectural patterns for inter-component redundancy

system. This is the case, for example, of automatic workarounds that rely on the equivalence of different compositions of the internal functionality [25]. Although approaches from both categories can be applied to different classes of systems, deliberately adding redundancy impacts on development costs, and is thus exploited more often in safety critical applications, while opportunistic redundancy has been explored more often in research on autonomic and self-healing systems.

Type. A system is redundant when some elements of its *code*, its input *data*, or its execution *environment* (including the execution processes themselves) are partially or completely replicated. Some approaches rely on redundant computation that replicates the functionality of the system to detect and heal a faulty computation. For example, N-version programming compares the results of equivalent computations to produce a correct result. This is a case of *code* redundancy. Other approaches rely on redundancy in the data handled during the computation. For example, so-called data diversity relies on redundancy in the data used for the computation, and not on the computation itself, which is based on the same code [26]. Yet other approaches exploit environmental conditions that influence the computation. For example environment perturbation techniques rely on redundancy that derive from different reactions of the environment [27]. Different types of redundancy apply to different types of systems and different classes of faults. As indicated in the introduction, the classification based on the type of replicated elements is similar to Ammar's classification in spatial, information and temporal redundancy [14] that applies better to the more general kind of redundancy that can be found when considering techniques to handle both hardware and software faults.

Triggers and adjudicators. Redundant components can be either triggered *preventively* to avoid failures, or exploited *reactively* in response to failures. In the first case, the system must decide when and where act to maximize the chance of avoiding failures. Examples of the preventive use of redundancy are rejuvenation techniques that reboot the system before failures occur [15]. In the second case, the system must at a minimum detect a failure, and therefore decide how to exploit the redundancy of the system in order to cope with the failure. We refer to the component of the system that makes these decisions as the *adjudicator*. We further classify a system by distinguishing adjudicators that are either *implicitly* built into the redundant mechanisms, or *explicitly* designed by the software engineers for a specific application. For example, N-version programming reveal errors automatically by comparing the results of executing redundant equivalent code fragments; the result is chosen with a majority vote between the different executions, and therefore amounts to an implicit adjudicator. On the other hand, recovery-blocks require explicit adjudicators that check for the correctness of the results to trigger alternative computations [28].

Faults. Redundancy may be more or less effective depending on the types of faults present in the system. In our taxonomy, we indicate the primary class of

faults addressed by each mechanism. In particular, we refer to the fault taxonomy proposed by Avizienis et al. and revised by Florio et al. [2,19]. Avizienis et al. distinguish three main classes of faults: *physical*, *development*, and *interaction* faults. Physical faults are hardware faults caused either by natural phenomena or human actions. Examples of physical faults are unexpected lacks of power, or physical hardware damages. Development faults are faults introduced during the design of the system, for example incorrect algorithms or design bottlenecks. Interaction faults are faults that derive from the incorrect interaction between the system and the environment, for example, incorrect settings that cause bad interactions with the system or malicious actions that aim to violate the system security. In this chapter, we focus on software faults, and thus we consider development and interaction faults, and not physical faults that are related to hardware problems. We further distinguish development faults that consistently manifest under well defined conditions (Bohrbugs) form development faults that cause software to exhibit non-deterministic behavior (Heisenbugs) [17,16], and we refer to interaction faults introduced with malicious objectives [2].

Table 2 summarizes the main exploitations of redundancy in fault tolerance and self-healing systems, characterized according to the categories introduced in this section. In the following sections, we discuss the taxonomy illustrated in Table 2. Here we identify the techniques listed in the table with a quick description and a reference to the main architectural implications. N-version programming compares the results of executing different versions of the program to identify errors (parallel evaluation pattern). Recovery blocks check the results of executing a program version and switch to a different version if the current execution fails (sequential alternatives pattern). Self-checking programming parallelizes the execution of recovery blocks (parallel selection pattern). Self-optimizing code changes the executing components to recovery from performance degradation (sequential alternatives pattern). Exception handling activates handlers to manage unplanned behaviors (sequential alternatives pattern). Rule engines code failure handlers that are activated through registries (sequential alternatives pattern). Wrappers intercept interactions and fix them when possible (intra-component). Robust data structures and software audits augment data structures with integrity checks (intra-component). Data diversity executes the same code with perturbed input data (either parallel selection or sequential alternatives pattern). Environment perturbation changes the execution environment and re-executes the failing code. Rejuvenation preventively reboots the system to avoid software aging problems. Process replicas execute the same process in different memory spaces to detect malicious attacks. Dynamic service substitution links to alternative services to overcome failures (sequential alternatives pattern). Genetic programming for fault fixing applies genetic algorithms to fix faults (intra-component). Automatic workarounds exploit the intrinsic redundancy of software systems to find alternative executions (intra-component). Checkpoint-recovery rebuilds a consistent state and re-executes the program. Reboot and micro-reboot restart the system to recovery from Heisenbugs.

Table 2. A taxonomy of redundancy for fault tolerance and self-managed systems

	Intention	Type	Adjudicator	Faults
N-version programming [9,29,30,31]	deliberate	code	reactive implicit	development
Recovery blocks [28,29]	deliberate	code	reactive explicit	development
Self-checking programming [32,29,33]	deliberate	code	reactive expl./impl.	development
Self-optimizing code [34,35]	deliberate	code	reactive explicit	development
Exception handling, rule engines [36,37,38]	deliberate	code	reactive explicit	development
Wrappers [39,40,41,42]	deliberate	code	preventive	Bohrbugs malicious
Robust data structures, audits [43,44]	deliberate	data	reactive implicit	development
Data diversity [26]	deliberate	data	reactive expl./impl.	development
Data diversity for security [45]	deliberate	data	reactive implicit	malicious
Rejuvenation [46,15,17]	deliberate	environment	preventive	Heisenbugs
Environment perturbation [27]	deliberate	environment	reactive explicit	development
Process replicas [47,48]	deliberate	environment	reactive implicit	malicious
Dynamic service substitution [10,49,11,50]	opportunistic	code	reactive explicit	development
Fault fixing, genetic programming [51,52]	opportunistic	code	reactive explicit	Bohrbugs
Automatic workarounds [53,25]	opportunistic	code	reactive explicit	development
Checkpoint-recovery [21]	opportunistic	environment	reactive explicit	Heisenbugs
Reboot and micro-reboot [12,13]	opportunistic	environment	reactive explicit	Heisenbugs

4 Deliberate Redundancy

Deliberately adding redundancy is common practice in the design of computer systems at every level, from single registers in a processor to entire components in a computer, to entire computers in a data center. In this section, we survey software fault tolerance and self-healing techniques that deliberately introduce redundancy into software systems at the code, data and environment levels.

4.1 Deliberate Code Redundancy

Deliberate software redundancy has been widely exploited at the code level. Classic approaches explored the use of N-version programming and recovery-blocks to

tolerate software faults. Later approaches introduced the concepts of self-checking and self-optimizing programming to overcome a wider variety of faults as well as performance issues. Recently some approaches proposed various forms of registries to identify healing procedures, mostly in the context of BPEL processes. A different form of deliberate code redundancy, defined in various contexts, is represented by wrappers. Wrappers add redundant code to detect and correct interaction problems such as incompatibilities of formats or protocols between software components.

N-version programming. The approach was originally proposed by Avizienis et al., and is one of the classic approaches to the design of fault tolerant software systems [9]. N-version programming relies on several programs that are designed independently and executed in parallel. The results are compared to identify and correct wrong outputs. The multiple versions must differ as much as possible in the use of design and implementation techniques, approaches, and tools. A general voting algorithm compares the results, and selects the final one based on the output of the majority of the programs. Since the final output needs a majority quorum, the number of programs determines the number of tolerable failures: a three-versions system can tolerate at most one faulty result, a five-versions system can tolerate up to two faulty results, and so on. In general, in order to tolerate k failures, a system must consists of $2k + 1$ versions.

The original N-version mechanism has been extended to different domains, in particular recently to the design of web- and service-based applications. Looker et al. define *WS-FTM*, a mechanism that supports the parallel execution of several independently-designed services. The different services implement the same functionality, and their results are validated on the basis of a quorum agreement [30]. Dobson implements N-version programming in WS-BPEL, by implementing the parallel execution of services with a voting algorithms on the obtained responses [29]. Gashi et al. describe and evaluate another typical application of N-version programming to SQL servers [31]. In this case, N-version programming is particularly advantageous since the interface of an SQL database is well defined, and several independent implementations are already available. However, reconciling the output and the state of multiple, heterogeneous servers may not be trivial, due to concurrent scheduling and other sources of non-determinism.

N-version programming is a relevant instance of *deliberate code-level redundancy*, since it requires the design of different versions of the same program. The approach relies on an general built-in consensus mechanism, and does not require explicit adjudicators: The voting mechanism detects the effects of faults by comparing the results of the program variants, and thus acts as a *reactive, implicit* adjudicator. N-version programming has been investigated to tolerate development faults, but, if used with distinct hardware for the different variants, it can tolerate also some classes of physical faults. This makes the approach particularly appealing in domains like service-oriented applications, where services are executed on different servers and may become unavailable due to server or network problems.

Recovery-blocks. The approach was originally proposed by Randell, and relies on the independent design of multiple versions of the same components [28]. Contrary to N-version programming, in this case the various versions are executed sequentially instead of in parallel. When the running component fails, the technique executes an alternate (redundant) component. If the alternate component fails as well, the technique selects a new one, and in the case of repeated failures, this process continues as long as alternate components are available. The recovery-blocks mechanism detects failures by running suitable acceptance tests, and relies on a rollback mechanism to bring the system back to a consistent state before retrying with an alternate components.

As for N-version programming, the core ideas behind recovery blocks have been extended to different domains, and in particular to web- and service-based applications. In their work that extends N-version programming to WS-BPEL, Dobson exploits also the BPEL *retry* command to execute an alternate service when the current one fails [29]. As in the classic recovery-block approach, alternate services are statically provided at design time.

The recovery block approach is another classic implementation of *deliberate code-level redundancy*, since it relies on redundant designs and implementations of the same functionality. However, recovery blocks differ from N-version programming in that they rely on *reactive, explicit* adjudicators to detect failures and trigger recovery actions. In fact, recovery blocks detect component failures by executing explicitly-designed acceptance tests. Like N-version programming, recovery-blocks target development faults, but, unlike N-version programming, they are less than ideal for physical faults, as they do not exploit parallel execution.

Self-checking programming. Further extending the main ideas of N-version programming and recovery blocks, Laprie et al. proposed self-checking programming, which is a hybrid approach that augments programs with code that checks its dynamic behavior at runtime [32]. A self-checking component can be either a software component with a built-in acceptance test suite, or a pair of independently designed components with a final comparison. Each functionality is implemented by at least two self-checking components that are designed independently and executed in parallel. If the main self-checking component fails, the program automatically checks the results produced by the alternative component to produce a correct result. At runtime, they distinguish between "acting" components that are in charge of the computation, and "hot spare" components that are executed in parallel to tolerate faults of the acting components. An acting components that fails is discarded and replaced by the hot spare. This way, self-checking programming does not require any rollback mechanism, which is essential with recovery blocks. The core idea of self-checking software goes back to 1975, when Yau et al. suggested software redundancy to check for the correctness of system behavior in order to improve system reliability [33].

Similarly to previous approaches, Dobson applies also the self-checking programming approach to service oriented applications, by calling multiple services in parallel and considering the results produced by the hot spare services only in case of failures of the acting one [29].

Self-checking programming is yet another example of *deliberate code-level redundancy*, since it is based on redundant implementations of the same functionalities. Self-checking programming uses *reactive* adjudicators that can be *implicit* or *explicit* depending on the design of the self-checking components. Components with a built-in acceptance test suite implement reactive, explicit adjudicators, while components with a final comparison of parallel results implement reactive, implicit adjudicators. Similarly to N-version programming and recovery blocks, self-checking programming has been introduced to tolerate *development faults*.

Self-optimizing code. Development faults may affect non-functional properties such as performance. The term self-optimization, used within the larger study of self-managed systems, refers to an automatic reaction of a system that would allow it to compensate for and recover from performance problems. Some approaches to self-optimization rely on redundancy. Diaconescu et al. suggest implementing the same functionalities with several components optimized for different runtime conditions. Applications can adapt to different performance requirements and execution conditions at runtime by selecting and activating suitable implementations for the current contexts [34].

Naccache et al. exploit a similar idea in the Web services domain [35]. They enhance web service applications with mechanisms that choose different implementations of the same service interfaces depending on the required quality of service. To maintain the required performance characteristics in web services applications, the framework automatically selects a suitable implementation among the available ones.

These self-optimizing approaches *deliberately* include *code* redundancy. In fact the presence of different components and web services at design time is required to allow these frameworks to work at runtime. The adjudicators are *reactive and explicit*, since the frameworks monitor the execution and when the quality of service offered by the application overcomes a given threshold then another component or service is selected.

Exception handling and rule engines (Registries). Exception handling is a classic mechanism that catches pre-defined classes of errors and activates recovery procedures (exception handlers) explicitly provided at design time [54]. Rule engines extend classic exception handling mechanisms by augmenting service-based applications with a *registry* of rule-based recovery actions. The registry is filled by developers at design time, and contains a list of failures each one with corresponding recovery actions to be executed at runtime. Both Baresi et al. [36] and Pernici et al. [37] propose registry-based approaches. They enhance BPEL processes with rules and recovery actions. In both cases, failures are detected at runtime by observing violations of some predetermined safety conditions, although the two approaches differ in the way they define rules and actions.

Mechanisms that rely on exception handlers and registries add redundant code deliberately, and rely on explicit adjudicators, which are managed as exceptions. Recovery actions address development faults.

Wrappers. The term *wrapper* indicates elements that mediate interactions between components to solve integration problems. Wrappers have been proposed in many contexts. Popov et al. propose wrappers in the context of the design of systems that integrate COTS components to cope with integration problems that derive from incomplete specifications [39]. Incompletely specified COTS components may be used incorrectly or in contexts that differ from the ones they have been designed for. The wrappers proposed by Popov et al. detect classic mismatches and trigger appropriate recovery actions, for example they switch to alternative redundant components. Chang et al. require developers to release components together with sets of healing mechanisms that can deal with failures caused by common misuses of the components [40]. Failure detectors and so-called healers are designed as exceptions that, when raised, automatically execute the recovery actions provided by the developers.

Salles et al. propose wrappers for off-the-shelf components for operating systems. With wrappers, Salles et al. improve the dependability of OS kernels that integrate COTS components with different dependability levels [41]. Fetzer et al. introduce "healers" to prevent some classes of malicious faults [42]. Fetzer's healers are wrappers that embed all function calls to the C library that write to the heap, and perform suitable boundary checks to prevent buffer overflows. Wrappers deliberately insert redundant code to prevent failures. They have been proposed to deal both with Bohrbugs and malicious attacks.

Costs and efficacy of code redundancy. As our survey shows, deliberate code redundancy has been exploited primarily to cope with development faults, and has been recently extended to cope with performance and security faults. Different approaches try to mitigate the additional design and execution costs, by trading recovery and adjudicator design costs for execution costs. N-version programming comes with high design and execution costs, but works with inexpensive and reliable implicit adjudicators. Recovery blocks reduce execution costs, but increase the cost of designing adjudicators. Self-checking components support a flexible choice between the two approaches at the price of complex execution frameworks. Software execution progressively consumes the initial explicit redundancy, since failing elements are discard and substituted with redundant ones. The efficacy of explicit redundancy is controversial. Supporters of explicit redundancy acknowledge the increased reliability of properly designed redundant systems [55]. Detractors provide experimental evidence of the limited improvements of the reliability of redundant over non redundant systems. For example, Brilliant et al. indicate that, in N-version programs, the amount of input errors increases unexpectedly, and the correlation is higher than predicted, thus reducing the expected reliability gain [56].

4.2 Deliberate Data Redundancy

Although with less emphasis, redundancy has been deliberately added to both data and, more recently, to the runtime environment. Deliberate data redundancy has been proposed to increase the dependability of data structures, to

reduce the occurrences of failures caused by specific input-dependent conditions (e.g., corner cases in data structures), and very recently to cope with some classes of security problems.

Robust data structures and software audits. Connet et al. introduced a preliminary form of data redundancy in the early seventies [44]. They augment systems with so called software audits that check for the integrity of the system itself at runtime. Taylor et al. exploited a form of deliberate data redundancy to improve the reliability of data structures [43]. Taylor et al. propose data redundancy consisting of additional code to trace the amount of nodes in data structures and of additional node identifiers and references to make data structures more robust. They use the redundant information to identify and correct faulty references. These approach exploits *data* redundancy that is *deliberately* added to the programs to tolerate *development* faults. The redundant information implicitly enables failures detection, thus adjudicators are *reactive and implicit.*

Data diversity. Knight et al. apply deliberate data redundancy to cope with failures that depend on specific input conditions [26]. Knight's approach is applicable to software that contains faults that result in failures with particular input values, but that can be avoided with slight modifications of the input. The approach relies on data "re-expressions" that can generate logically equivalent data sets. Re-expressions are *exact* if they express the same input in a different way, thus producing the expected output; They are *approximate* if they change the input and thus produce a different output but within an accepted range. Data diversity is implemented in the form of either "retry blocks" that borrow from the idea of recovery blocks, or "N-copy programming" that redefine N-version programming for data. Therefore, data diversity uses both *reactive* and *implicit* adjudicators. As for recovery blocks and N-version programming, data diversity addresses development faults.

Data diversity for security. Recently Knight et al. extended the conceptual framework of data diversity to cope with security problems [45]. They apply data diversity in the form of N-variant systems to provide high-assurance conjectures against a class of data corruption attacks. Data are transformed into variants with the property that identical concrete data values have different interpretations. In this way attackers would need to alter the corresponding data in each variant in a different way while sending the same inputs to all variants. The only available implementation is run in parallel on the different data sets, and executions are compared. Thus this approach relies on *data* redundancy *deliberately* added to tolerate *malicious* faults. Since the approach relies on the parallel execution and the comparison of results, the adjudicator is *implicit.*

Despite early attempts trace back almost 30 years, deliberate data redundancy has not been exploited as thoroughly as code redundancy. Most approaches focus on development faults. Recent work indicates space for applications to non-functional faults.

4.3 Deliberate Environment Redundancy

Deliberate environment redundancy is the most basic form of redundancy, and has been used extensively to increase reliability in the face of purely hardware faults, for example in the case of database replication. Deliberate environment redundancy consists of deliberately changing the environment conditions and re-execute the software system under the new conditions. Thus, this form of redundancy impacts on the program execution rather then on the program structure. We only mention this well-known and widely studied application of environment redundancy in passing here because we intend to focus specifically on software faults. Therefore we describe in detail only some more recent uses of environment redundancy that are more significant for this class of faults.

Rejuvenation. The first notable attempt to deliberately modify the environment conditions to work around failures tracks back to the nineties, when Wang et al. proposed software rejuvenation. Rejuvenation is a technique that works with environment diversity, and relies on the observation that some software systems fail due to "age," and that proper system reinitializations can avoid such failures [57,17]. Wang et al. focused on memory-management faults, such as memory leaks, memory caching, and weak memory reuse, that can cause the premature termination of the program execution. Rejuvenation amounts to cleaning the volatile state of the system periodically, whenever it does not contain useful information. The same research group improved software rejuvenation by combining it with checkpoints: By rejuvenating the program every N checkpoints, they can minimize the completion time of a program execution [46].

Software rejuvenation approaches are *deliberate* redundant changes to the environment, since the memory state is cleared intentionally by re-executing some global initialization procedures, thereby presenting a new environment to the system. Rejuvenation acts independently from the occurrence of failures, thus it can be both reactive or preventive. However it does not rely on an adjudicator that explicitly identifies a failure, and thus we classify it as preventive from the adjudicator viewpoint. These approaches work well for Heisenbugs.

Environment perturbation. Using the analogy of allergies in humans and other animals, and specifically of the treatment of such allergies, Qin et al. suggested a rollback mechanisms, called RX, that partially re-executes failing programs under modified environment conditions to recover from deterministic as well as non deterministic faults [27]. The mechanism is based on environment changes that include different memory management strategies, shuffled message orders, modified process priority, and reduced user requests. These changes in the execution environment can prevent failures such as buffer overflows, deadlocks and other concurrency problems, and can avoid interaction faults often exploited by malicious requests.

Similarly to software rejuvenation, RX is based on *deliberate* environment redundancy, since the applied changes explicitly create different environments where the programs can be re-executed successfully. However, contrary to

rejuvenation, RX relies on *reactive and explicit* adjudicators to start proper recovery actions. In particular, the environment changes are triggered by exceptions or by sensors that monitor the system execution. This technique works mainly with *Heisenbugs*, but can be effective also with some *Bohrbugs* and *malicious* faults.

Process replicas. The main concepts of N-version programming have been extended to environment changes to cope with malicious faults. Cox et al. proposed executing N-variants of the same code under separate environment conditions and compare their behavior to detect malicious attacks [47]. The aim is to complicate the attackers' task by requiring malicious users to simultaneously compromise all system variants with the same input to actually succeed in the attack. The framework provided by Cox starts from the original program, and automatically creates different variants by partitioning the address space, and by tagging the instructions. Partitioning the address space can prevent memory attacks that involve direct reference to absolute addresses, while tagging the instructions (that is, prepending a variant-specific tag to all instructions) can detect code injection. Bruschi et al. improved Cox' process replicas with a new mechanism that also detects attacks that attempt to overwrite memory addresses [48].

Approaches based on process replicas *deliberately* add redundancy to the execution *environment*, since the variants are obtained through explicit, though automatic, changes. Cox' tagging mechanism acts on the program, and thus also creates redundancy in the code. Process replicas do not require explicit adjudicators, but instead rely on *reactive, implicit* mechanisms in the same way that N-version programming derives a single output value, by executing the variants in parallel, and then by comparing execution results at runtime. Process replicas target *malicious faults*, and do not seem well suited to deal with other types of faults.

In general, deliberate redundancy in environment execution conditions has been exploited only recently, and seems well suited to deal with Heisenbugs and some classes of interaction faults, especially malicious faults, that are particularly difficult to detect and remove.

5 Opportunistic Redundancy

While deliberate redundancy has been exploited since the seventies and in many contexts, implicit redundancy has been explored only recently, with some promising results. Implicit redundancy at the code level usually stems from the complexity of system internals, which in turn result in a partial overlap of functionality within different program elements. Implicit redundancy at the environment level comes from the complexity of the execution environment, which typically does not behave deterministically to all requests, and therefore may allow for different but functionally equivalent behaviors.

5.1 Opportunistic Code Redundancy

Implicit redundancy at code level has been exploited both in specific application domains, mostly in the context of service oriented applications and more general of dynamically bound components, and with specific technologies, namely genetic programming.

Dynamic service substitution. Popular services are often available in multiple implementations, each one designed and operated independently, each one also possibly offering various levels of services, but with every one complying to an equivalent, common interface. In fact, this is more or less the vision of service-oriented computing. Some researchers propose to take advantage of the available, independent implementations of the same or similar service to increase the reliability of service-oriented applications, especially for failures that may be caused by malfunctioning services or unforeseen changes in the functionalities offered by the current implementation. Subramanian et al. enhance BPEL with constructs to find alternative service implementations of the same interfaces in order to overcome unpredicted response or availability problems [10]. Taher et al. enhance runtime service substitution by extending the search to services implementing similar interfaces, and by introducing suitable converters to use services that, although different, are sufficiently similar to admit to a simple adaptation [49]. Sadjadi et al. further simplify the substitution of similar service implementations by proposing transparent shaping to weave alternative services invocation at runtime, thus avoiding manual modification of the original code [11]. Mosincat et al. define an infrastructure to handle dynamic binding of alternative services that can handle both stateless and stateful Web services [50]. In summary, service substitution amounts to exploiting available redundant code in an opportunistic manner, it is triggered in reaction to faults thanks to explicit adjudicators, and allows systems to tolerate both development faults and physical faults.

Fault fixing using genetic programming. Recently both Weimer et al. and Arcuri et al. investigated genetic programming as a way to automatically fix software faults [51,52]. Both approaches assume the availability of a set of test cases to be used as adjudicator. When the software system fails, the runtime framework automatically generates a population of variants of the original faulty program. Genetic algorithms evolve the initial population guided by the results of the test cases that select a new "correct" version of the program.

Genetic programming does not require the deliberate development of redundant functionality, but exploit the *implicit code* redundancy *opportunistically* to produce variants of the original programs and select a "correct" variant. Genetic approaches react to failures detected by test suites, and thus rely on *reactive and explicit* adjudicators to identify and correct *Bohrbugs*.

Automatic workarounds. In some recent work, we investigated the possibility of automatically identifying workarounds by opportunistically exploiting the implicit redundancy present in the code [53,25]. We observed that many complex software systems provide the same functionality through different combinations

of elementary operations. So, in order to respond to a failing sequence of operations, we proposed a technique to automatically identify alternative execution sequences that are expected to have the same *intended* effect of failing sequence, but that are not affected by failures, and therefore that can be used as workarounds. Our technique relies on other mechanisms to detect failures, and to bring the system back to a consistent state immediately after the failure. When the system fails, the technique automatically examines the failing sequence, and on the bases of a specification of the system or its interface, generates alternate execution sequences that serve as potential workarounds. The generated sequences are sorted according to the likelihood of success and are then executed up to a correct execution. This technique explores opportunistic redundancy in a new way, to some extent mimicking what a real user would do in the attempt to work around emerging faulty behaviors. Being completely automatic, the technique can also explore redundancy not exploitable by end users, thus with better chances to find a useful solution to the problem.

The approach of automatic workarounds is therefore *opportunistic* in the way it explores intrinsic *code* redundancy. It is triggered by an *explicit, reactive* adjudicator, and is useful primarily to avoid the effects of *development* faults.

Considerations on the opportunistic exploitation of code redundancy. It is only recently that researchers have started to investigate the possibility of exploiting code redundancy opportunistically. These studies rely on characteristics of emergent application domains, like service-based applications, and programming techniques like genetic programming. The results are encouraging and suggest that implicit code redundancy can be exploited also in classic application domains to increase reliability with little additional costs. However, we should emphasize that making good use of redundancy in an opportunistic manner is an issue of availability and cost. Contrary to the approaches discussed in the previous section in which redundancy is added by design, opportunistic techniques such as automatic fault fixing and automatic workarounds do not require the development of any redundant code. Service substitution does require multiple implementations, but their development cost is presumably amortized over several systems sharing the same pool of service implementations. So, the main advantage of the opportunistic approach is that it does not incur serious additional development costs. However, it should be clear that the same approach still relies on code redundancy, although it does so *implicitly*. So, there is always a question of whether such latent redundancy really exists. And furthermore, even when redundant code exists, it typically requires runtime search and adaptation.

5.2 Opportunistic Environment Redundancy

Approaches that opportunistically exploit environment redundancy extend and formalize the common experience of non-deterministic behavior of the execution environment: Simple system reboots are often used as last resource to overcome unexpected failures.

Checkpoint-recovery. Checkpoint and recovery techniques exploit environment redundancy by periodically saving consistent states to be used as safe roll backs [21]. When the system fails, it is brought back to a consistent state and re-executed, to solve temporary problems that may have been caused by accidental, transient conditions in the environment.

These approaches *opportunistically* exploit *environment* redundancy, since a system would re-execute the same code without trying to modify the environment, but instead relying on spontaneous changes in the environment to avoid the conditions that created the failure. Notice that this is different from other explicit approaches, such as that of the RX method by Qin et al. that deliberately changes the environment before re-executing the code [27]). Checkpoint-and-recovery requires *reactive and explicit* adjudicators to determine if the system has failed, and therefore to roll back to a consistent state. These techniques are effective in dealing with *Heisenbugs* that depend on temporary execution conditions, but do not work well for *Bohrbugs* that persist in the code and in the execution environment.

Reboot and micro-reboot. The classic brute force but surprisingly effective approach that consists in simply rebooting the systems has been refined by Candea et al., who propose local micro-reboots to avoid the high cost of complete reboots [12]. The same approach was extended to service-based applications by Zhang et al. [13]. Although intuitively simple, micro-reboot approaches require careful modular design of the systems as well as adequate runtime support for reboot operations that do not affecting the overall execution. These approaches are *opportunistic*, and exploit redundant behavior of the execution environment to overcome *Heisenbugs*. As in the other cases, they operate when triggered by *reactive, explicit* adjudicators, since they react to system failures explicitly notified by the adjudicators.

6 Conclusions

Both the fault tolerance and the self-healing communities work on techniques to reduce the runtime effects of faults during software execution, to guarantee software reliability also in the presence of faults. The many techniques investigated so far tackle several problems, work under different assumptions, impact in various forms on the development and execution costs, address various application domains, and may affect software architectural issues. The known attempts to frame the various approaches under a unifying view have focused either on fault tolerance or self-healing techniques, but do not unify well the results of both communities. Thus, they miss the important relations between the work in the two areas that are strictly intertwined. In this paper, we identify few general architectural patterns that are implemented by different techniques, and we propose a unifying framework to classify techniques that reduce the effects of faults at runtime, and we compare the main approaches to fault tolerance and self-healing. The framework elaborates on the different ways the techniques proposed so far exploit redundancy and reveals open areas of investigation.

References

1. Pullum, L.L.: Software Fault Tolerance Techniques and Implementation. Artech House, Inc., Norwood (2001)
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable Secure Computing 1(1), 11–33 (2004)
3. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
4. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington (2007)
5. Lyons, R., Vanderkulk, W.: The use of triple-modular redundancy to improve computer reliability. IBM Journal of Research and Development 6(2), 200–209 (1962)
6. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (RAID). SIGMOD Record 17(3), 109–116 (1988)
7. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI 2004: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, Berkeley, CA, USA (2004)
8. El Abbadi, A., Skeen, D., Cristian, F.: An efficient, fault-tolerant protocol for replicated data management. In: PODS 1985: Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems, pp. 215–229. ACM, New York (1985)
9. Avizienis, A.: The N-version approach to fault-tolerant software. IEEE Transactions on Software Engineering 11(12), 1491–1501 (1985)
10. Subramanian, S., Thiran, P., Narendra, N.C., Mostefaoui, G.K., Maamar, Z.: On the enhancement of BPEL engines for self-healing composite web services. In: SAINT 2008: Proceedings of the 2008 International Symposium on Applications and the Internet, pp. 33–39. IEEE Computer Society, Washington (2008)
11. Sadjadi, S.M., McKinley, P.K.: Using transparent shaping and web services to support self-management of composite systems. In: ICAC 2005: Proceedings of the Second International Conference on Automatic Computing, pp. 76–87. IEEE Computer Society, Washington (2005)
12. Candea, G., Kiciman, E., Zhang, S., Keyani, P., Fox, A.: JAGR: An autonomous self-recovering application server. In: Active Middleware Services, pp. 168–178. IEEE Computer Society, Los Alamitos (2003)
13. Zhang, R.: Modeling autonomic recovery in web services with multi-tier reboots. In: ICWS 2007: Proceedings of the IEEE International Conference on Web Services (2007)
14. Ammar, H.H., Cukic, B., Mili, A., Fuhrman, C.: A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. Annals Software Engineering 10(1-4), 103–150 (2000)
15. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.D.: Software rejuvenation: Analysis, module and applications. In: FTCS 1995: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, vol. 381. IEEE Computer Society, Washington (1995)
16. Gray, J.: Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems, pp. 3–12 (1986)
17. Grottke, M., Trivedi, K.: Fighting bugs: Remove, retry, replicate, and rejuvenate. Computer 40(2), 107–109 (2007)

18. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing—degrees, models, and applications. ACM Computing Surveys 40(3), 1–28 (2008)
19. Florio, V.D., Blondia, C.: A survey of linguistic structures for application-level fault tolerance. ACM Computing Surveys 40(2), 1–37 (2008)
20. Littlewood, B., Strigini, L.: Redundancy and diversity in security. In: Samarati, P., Ryan, P.Y.A., Gollmann, D., Molva, R. (eds.) ESORICS 2004. LNCS, vol. 3193, pp. 423–438. Springer, Heidelberg (2004)
21. Elnozahy, M., Alvisi, L., min Wang, Y., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys 34(3), 375–408 (2002)
22. Gacek, C., de Lemos, R.: Architectural description of dependable software systems. In: Besnard, D., Gacek, C., Jones, C. (eds.) Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective, pp. 127–142. Springer, Heidelberg (2006)
23. Harrison, N.B., Avgeriou, P.: Incorporating fault tolerance tactics in software architecture patterns. In: SERENE 2008: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, pp. 9–18. ACM, New York (2008)
24. Hanmer, R.: Patterns for Fault Tolerant Software. Wiley Publishing, Chichester (2007)
25. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: SEAMS 2008: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 17–24. ACM, New York (2008)
26. Ammann, P.E., Knight, J.C.: Data diversity: An approach to software fault tolerance. IEEE Transactions on Computers 37(4), 418–425 (1988)
27. Qin, F., Tucek, J., Zhou, Y., Sundaresan, J.: Rx: Treating bugs as allergies—a safe method to survive software failures. ACM Transactions on Computer Systems 25(3), 7 (2007)
28. Randell, B.: System structure for software fault tolerance. In: Proceedings of the international conference on Reliable software, pp. 437–449. ACM, New York (1975)
29. Dobson, G.: Using WS-BPEL to implement software fault tolerance for web services. In: EUROMICRO 2006: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 126–133. IEEE Computer Society, Washington (2006)
30. Looker, N., Munro, M., Xu, J.: Increasing web service dependability through consensus voting. In: COMPSAC 2005: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), vol. 2, pp. 66–69. IEEE Computer Society, Washington (2005)
31. Gashi, I., Popov, P., Stankovic, V., Strigini, L.: On designing dependable services with diverse off-the-shelf SQL servers. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems II. LNCS, vol. 3069, pp. 191–214. Springer, Heidelberg (2004)
32. Laprie, J.C., Béounes, C., Kanoun, K.: Definition and analysis of hardware- and software-fault-tolerant architectures. Computer 23(7), 39–51 (1990)
33. Yau, S.S., Cheung, R.C.: Design of self-checking software. In: Proceedings of the International Conference on Reliable software, pp. 450–455. ACM, New York (1975)
34. Diaconescu, A., Mos, A., Murphy, J.: Automatic performance management in component based software systems. In: ICAC 2004: Proceedings of the First International Conference on Autonomic Computing, pp. 214–221. IEEE Computer Society, Washington (2004)

35. Naccache, H., Gannod, G.: A self-healing framework for web services. In: ICWS 2007: Proceedings of the 2007 IEEE International Conference on Web Services, pp. 398–345 (2007)
36. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with dynamo and the JBoss rule engine. In: ESSPE 2007: International workshop on Engineering of software services for pervasive environments, pp. 11–20. ACM, New York (2007)
37. Modafferi, S., Mussi, E., Pernici, B.: SH-BPEL: a self-healing plug-in for WS-BPEL engines. In: MW4SOC 2006: Proceedings of the 1st workshop on Middleware for Service Oriented Computing, pp. 48–53. ACM, New York (2006)
38. Fugini, M.G., Mussi, E.: Recovery of Faulty Web Applications through Service Discovery. In: SMR 2006: 1st International Workshop on Semantic Matchmaking and Resource Retrieval: Issues and Perspectives, Seoul, Korea (2006)
39. Popov, P., Riddle, S., Romanovsky, A., Strigini, L.: On systematic design of protectors for employing OTS items. In: Euromicro 2001: in Proceedings of the 27th Euromicro Conference, pp. 22–29 (2001)
40. Chang, H., Mariani, L., Pezzè, M.: In-field healing of integration problems with COTS components. In: ICSE 2009: Proceeding of the 31st International Conference on Software Engineering, pp. 166–176 (2009)
41. Salles, F., Rodrguez, M., Fabre, J.C., Arlat, J.: Metakernels and fault containment wrappers. In: International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, Los Alamitos (1999)
42. Fetzer, C., Xiao, Z.: Detecting heap smashing attacks through fault containment wrappers. In: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems, pp. 80–89 (2001)
43. Taylor, D.J., Morgan, D.E., Black, J.P.: Redundancy in data structures: Improving software fault tolerance. IEEE Transactions on Software Engineering 6(6), 585–594 (1980)
44. Connet, J.R., Pasternak, E.J., Wagner, B.D.: Software defenses in real-time control systems. In: Proceedings of the International Symposium on Fault-Tolerant Computing, pp. 94–99 (1972)
45. Nguyen-Tuong, A., Evans, D., Knight, J.C., Cox, B., Davidson, J.W.: Security through redundant data diversity. In: DSN 2008: IEEE International Conference on Dependable Systems and Networks, pp. 187–196 (2008)
46. Garg, S., Huang, Y., Kintala, C., Trivedi, K.S.: Minimizing completion time of a program by checkpointing and rejuvenation. SIGMETRICS Performance Evaluation Review 24(1), 252–261 (1996)
47. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: a secretless framework for security through diversity. In: USENIX SS 2006: Proceedings of the 15th conference on USENIX Security Symposium. USENIX Association, Berkeley (2006)
48. Bruschi, D., Cavallaro, L., Lanzi, A.: Diversified process replicae for defeating memory error exploits. In: WIA 2007:3rd International Workshop on Information Assurance. IEEE Computer Society, Los Alamitos (2007)
49. Taher, Y., Benslimane, D., Fauvet, M.C., Maamar, Z.: Towards an approach for web services substitution. In: IDEAS 2006: Proceedings of the 10th International Database Engineering and Applications Symposium, pp. 166–173. IEEE Computer Society, Washington (2006)
50. Mosincat, A., Binder, W.: Transparent runtime adaptability for BPEL processes. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 241–255. Springer, Heidelberg (2008)

51. Weimer, W., ThanVu Nguyen, a.C.L.G., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE 2009: Proceeding of the 31st International Conference on Software Engineering, pp. 364–374 (2009)
52. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: CEC 2008: Proceeding of IEEE Congress on Evolutionary Computation (2008)
53. Carzaniga, A., Gorla, A., Pezzè, M.: Healing web applications through automatic workarounds. International Journal on Software Tools for Technology Transfer 10(6), 493–502 (2008)
54. Goodenough, J.B.: Exception handling: issues and a proposed notation. Communications of the ACM 18(12), 683–696 (1975)
55. Hatton, L.: N-version design versus one good version. IEEE Software 14(6), 71–76 (1997)
56. Brilliant, S.S., Knight, J.C., Leveson, N.G.: Analysis of faults in an N-version software experiment. IEEE Transactions on Software Engineering 16(2), 238–247 (1990)
57. Wang, Y.M., Huang, Y., Vo, K.P., Chung, P.Y., Kintala, C.: Checkpointing and its applications. In: FTCS 1995: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, pp. 22–31. IEEE Computer Society, Washington (1995)