Computing
DOI 10.1007/s00607-010-0107-y

A survey on self-healing systems: approaches and systems

Harald Psaier · Schahram Dustdar

Received: 21 July 2009 / Accepted: 11 July 2010

Abstract Present large-scale information technology environments are complex, heterogeneous compositions often affected by unpredictable behavior and poor manageability. This fostered substantial research on designs and techniques that enhance these systems with an autonomous behavior. In this survey, we focus on the self-healing branch of the research and give an overview of the current existing approaches. The survey is introduced by an outline of the origins of self-healing. Based on the principles of autonomic computing and self-adapting system research, we identify self-healing systems' fundamental principles. The extracted principles support our analysis of the collected approaches. In a final discussion, we summarize the approaches' common and individual characteristics. A comprehensive tabular overview of the researched material concludes the survey.

Keywords Autonomous behaving systems · Autonomic computing · Self-adaptive systems · Self-* properties · Self-healing principles · Self-healing approaches · Survey

Mathematics Subject Classification (2000) 00-02

Communicated by C.H. Cap.

H. Psaier (✉) · S. Dustdar
Distributed System Group, Institute of Information Systems 184/1,
Vienna University of Technology, Argentinierstrasse 8, 1040 Vienna, Austria
e-mail: hpsaier@infosys.tuwien.ac.at

S. Dustdar
e-mail: dustdar@infosys.tuwien.ac.at

Published online: 05 August 2010

1 Introduction

1.1 The complexity problem

Modern system design often results in humans overwhelmed by the effort to properly control the assembled collection. Designs comprise heterogeneous, tightly, and loosely coupled components. Likewise, dependencies of different application software are application contingent and often interfering. The dilemma became one of the main driving forces for research on autonomously behaving systems. As a result of the complexity, Ganek and Corbi [30] complain about the increasing costs in maintenance. Huebscher and McCann [42] describe it as the increasing human effort required to keep systems operational. Salehie and Tahvildari [71] blame it on the heterogeneity, dynamism, and interconnectivity in software applications, services and networks. Finally, Paul [65] brings it to the point and calls it THE obstacle: the increasing complexity.

1.2 Autonomous behavior

Recently, the trend of assembling heterogeneous parts to a purposefully collaborating, and foremost profitable system unfortunately results often in poor security, dependability, and maintenance along with many other difficulties. These challenges motivated visions of self-aware systems described by several research papers: Most prominently by the *autonomic computing* vision of IBM introduced by Kephart and Chess [50]. Autonomic computing's naming derives from the research on nature's autonomic nervous system. It tries to follow the concept of decoupling the high level reasoning from an independent maintenance system underneath. The research focuses on possible solutions for autonomous maintenance in current computing infrastructures. With overlapping intentions the research on *self-adaptive systems* has evolved. Salehie and Tahvildari [72] see one distinction in the fact that self-adaptive systems try to focus on challenges at a more general level. Most of their contributions cover higher level functionality such as the autonomous management, control, evaluation, maintenance, and organization of a whole systems.

Nevertheless, there is one common underlying idea: introducing an autonomous behavior to handle an otherwise complex and unmaintainable system. This autonomous behavior must independently take decisions at runtime and manage the assigned system. Management actions (e.g., configure, adapt, recover) are goal dependent, however, must result in a consistent system. The success of an adequate autonomous behavior depends on an accurate system knowledge. Both, autonomic computing and self-adaptive system research [72,75] agree, only the knowledge of internal state (self aware) and external situation (context aware) allows for proper adaptation. This combined awareness is gained by filtered data from sensing and feedback from effecting interfaces aligned to the context resulting in an accurate system overview. To keep a current view and meet contingent time constraints a closed loop system is assumed. Figure 1 outlines the layer dependencies and data-flow between the required data.

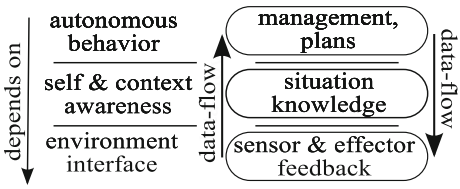

Fig. 1 Required dependencies and data-flow for systems with autonomous behavior

1.3 Self-healing research origins

The most cited cornerstone of the mentioned research areas is probably the autonomic computing initiative by IBM currently comprising several papers and research directions.[1] Ganek and Corbi [30] claim that current systems lack comprehensibility and require an extension for autonomous behavior, by adapting at runtime to unpredictable system changes. They emphasize the need by presenting numbers of the increasing amount spend on system maintenance over the last years. In a nutshell, they cast the fundamentals of their vision for autonomic computing onto four self properties tied to a self-managing system, including:

self-configuring: The ability to readjust itself "on-the fly"
self-healing: Discover, diagnose, and react to disruptions
self-optimization: Maximize resource utilization to meet end-user needs
self-protection: Anticipate, detect, identify, and protect itself from attacks.

Over the years this list of properties has been extended and is currently covered by the research on self-*, self-X respectively, properties (refer to glossary in [75] for an extended overview). Self-healing research, the focus of this survey, has not only become an integral part of the autonomic computing vision but generally of the research on autonomously behaving systems. Thus, Salehie and Tahvildari [72] include self-healing in self-adaptive research and describe it as a combination of self-diagnosing and self-repairing with the capabilities to diagnose and recover from malfunctions. Valid for both directions, Kephart and Chess [50] define the common objectives of self-healing as to maximize system's availability, survivability, maintainability, and reliability.

Research on self-healing systems has its origin in fault-tolerant and self-stabilizing systems research. Fault-tolerant systems handle transient and mask permanent failures in order to return to a valid state [67]. Self-stabilizing systems [25] are considered a non fault masking approach for fault-tolerant systems. These systems have two distinct properties. Arora and Gouda [9] refer to them as (i) the system is guaranteed to return to a legal state in a finite amount of time regardless of interferences (*convergence*) and (ii) once in legal state it tries to remain in the same (*closure*). More recent research in self-stabilizing computer systems seizes on the challenges of autonomic computing and self-healing research. Most notably the research of Dolev and Schiller [26] on group communication on arbitrary networks with self-stabilizing protocols.

Contemporary large-scale networked systems have become highly distributed and allow new levels and structures of management and organization. However, such

[1] http://www.research.ibm.com/autonomic/.

restructuring is accompanied by new risks and loss of comprehensive control. Indeed, arbitrary behaving systems are contrary to the convergence and closure properties of self-stabilization. Survivable systems [55] are designed to sustain the unexpected. Their main approach is to classify the parts of the system according to their overall essence. On a malicious influence the system focuses on maintaining the essential services and recovers non-essential services after intrusions have been dealt with. Finally, Ghosh et al. [31] note that exceptional situations might also require human intervention to support self-healing systems.

In short, a system with self-healing properties can be identified as a system that comprises *fault-tolerant, self-stabilizing, and survivable system* capabilities and, if needed, must be *human supported*.

The rest of this survey is organized as follows: Section 2 presents the principles and requirements for a self-healing enhanced system. Section 3 describes the areas of application, and some prominent representatives and solutions to the problems. A conclusion and comparison of the research is given in Sect. 4 and a summary and outlook in Sect. 5 concludes the survey.

2 Self-healing principles

This section describes the main principles of self-healing systems. It will help to understand the design decisions of the researched approaches and their underlying structures. Starting with a current definition of self-healing systems, we identify the important parts of a self-healing system to give a detailed insight into their purpose, composition, and functionality.

2.1 What is self-healing?

Included by IBM [30] as one of the main four properties defining an autonomic system, Ghosh et al. [31] provide a most recent definition of self-healing systems:

> "…a self-healing system should recover from the abnormal (or "unhealthy") state and return to the normative ("healthy") state, and function as it was prior to disruption."

One might argue that this is very general and highly similar to what is expected of the well established fault-tolerant or recent survivable systems. Fault-tolerant systems comprise stabilization techniques and replication strategies as essential methods for recovery. Therefore, Ghosh et al. [31] admit that self-healing systems in some cases are seen as subordinate to fault-tolerant systems. Survivable systems handle malicious behavior by containing failing components and securing the "essential services" representing a minimal but functioning system configuration [27,55,57]. Generally, the focus of self-healing research is on recovery as an elaborate process. This comprises both, methods for stabilizing, replacing, securing and isolating, but more essentially, strategies to repair and prevent faults. [31] identify the key aspect of self-healing systems as *recovery oriented computing*. This might also be a reason, why some of the

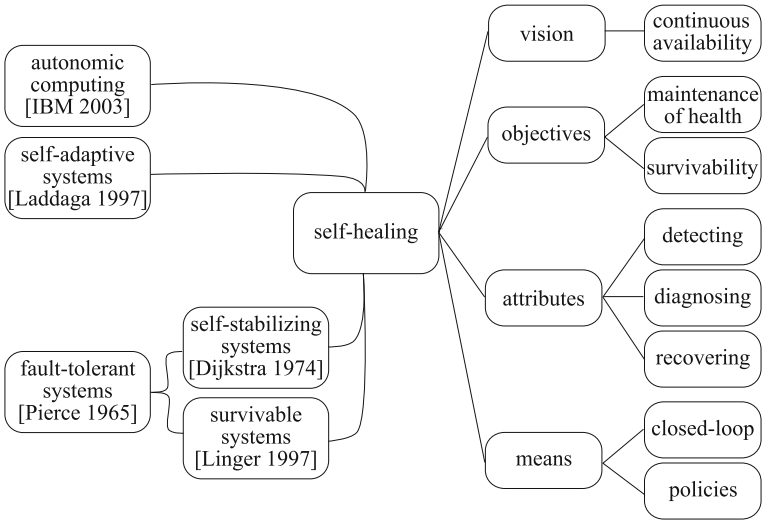

Fig. 2 Relations and properties of self-healing research

researched approaches outline self-healing only as an enhanced recovery method (e.g., [3,21]). Sterritt [75] sees the efforts of all autonomic computing as an evolutionary, well elaborated path to achieve the goals. Ganek and Corbi [30] further detail self-healing applications' operation mode as an organized process of detecting and isolating a faulty component, taking it off line, fixing the failed component, and reintroducing the fixed or replacement component into the system without any apparent disruption. For Ganek and Corbi [30] the objective of self-healing properties is to support system's reliability by minimizing the outages. Additionally, self-healing systems should be able to anticipate conflicts trying to prevent possible failures.

To summarize, the reason for enhancing a system with self-healing properties is to achieve *continuous availability*. Compensating the dynamics of a running system, self-healing techniques momentarily are in charge of the *maintenance of health*. *Enduring continuity* includes resilience against intended, necessary adaptations and unintentional, arbitrary behavior. Self-healing implementations work by *detecting* disruptions, *diagnosing* failure root cause and deriving a remedy, and *recovering* with a sound strategy. Additionally, to the accuracy of the essential sensor and actuator infrastructure, the success depends on timely detection of system misbehavior. This is only possible by continuously analyzing the sensed data as well as observing the results of necessary adaptation actions. The system design leads to a *control loop* similar assembly. An environment dependent and preferably adaptable set of *policies* support remedy decisions. Possible policies include simple sets of event dependent instructions but also extended AI estimations supporting the resolution of previously unknown faults. A conspectus of the research on self-healing properties is given in Fig. 2. At the bottom, the origins of the self-healing ideas are illustrated. On the top some research based on self-healing research is depicted. The properties of self-healing are listed on the right.

2.2 Self-healing loop

The main design element of autonomic computing is the autonomic element [42,44, 50]. It is kept very abstract to fit the internals of all the autonomic properties. The element comprises a manager that holds five distinct functions with individual tasks.

monitor: The monitor gathers status information from the system through sensors and pre-processes it for the analyze task.

analyze: This entity determines whether the received monitored information must follow a designated action. This is generally done by comparing status information to system specific thresholds.

plan: A running system often is full of situation specific dynamics. Therefore, an accurate, sound, and planed deployment of the actions demanded by analyze is required.

execute: Presents the entity that executes the parts of previously conceived plans on the managed element.

knowledge: This represents the knowledge base consumed and produced by all four previously mentioned tasks.

The collaboration of the five tasks assembles the work of the manager. More precisely, the subtask of a task is to process the input and filter the output for further processing. It becomes obvious that there is a data-flow in the form of a loop among the tasks. This was called the autonomic control loop sometimes referred as MAPE or MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop [44].

The idea of a continuous multi-state processing loop fits best the operation mode of self-healing approaches. In self-healing literature, the five autonomic processes are usually reduced and included into three main stages in a loop. Kephart and Chess [50] identify them as detection, diagnosis, and repair. Salehie and Tahvildari [72] call it a sum of self-diagnosing and self-repairing with discovery, diagnosing, and reacting stages. Parashar and Hariri [63] only consider detect and recover as the stages. Huebscher and McCann [42] see the three in detect, diagnose, and fix actions. Considering the researched work, it is safe to say that for the first stage detection is the most adequate definition. Originating from the Latin *detectio*, the act of uncovering or revealing an alternating of the normal behavior describes the stage's task the most accurate. Analysis and planning functions are comprised by the diagnosis in self-healing implementations. A set of rules or adaptable policies support diagnosis in planning. The most appropriate denotation for the final stage is recovery. Recovery is considered extended, however, not always entirely successful, and what differs self-healing from related remedy techniques. Figure 3 depicts the formation of the self-healing loop with the data-flow among the three stages and the environmental interfaces.

detecting: Filters any suspicious status information received from samples and reports detected degradations to diagnosis.

diagnosing: Includes root cause analysis and calculates an appropriate recovery plan with the help of a policy base.

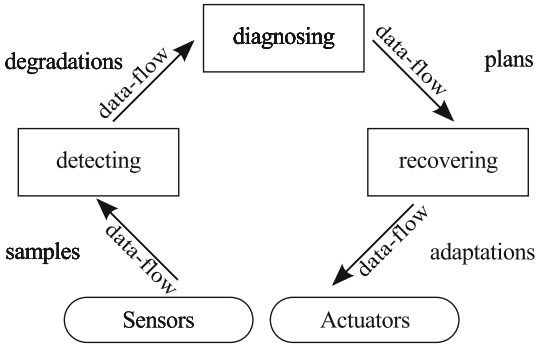

Fig. 3 Staged loop of self-healing

recovery: Carefully applies the planned adaptations meeting the constraints of the system capabilities and avoids any unpredictable side effects.

2.3 Self-healing states

The success of self-healing extensions depends on the distinction between system's intentional states and degraded, unacceptable states. The operating environment of self-healing extensions large-scale, unreliable systems, hold various error sources, possibly varying over time. The robustness of the self-healing alignment must not depend on a single element but the system as a whole should be able to recover from failures [81]. Thus, single element failures should have only minor impact on the whole system. In many cases there is no fine line, clearly separating acceptable from an unacceptable state. Instead, there is a momentary transmission zone in between.

The most recent model presented by Ghosh et al. [31], in particular, features a fuzzy transition zone with an unclear "Degraded State". This state reflects the fact that the adverse conditions of a systems cause self-healing systems to drift in a still acceptable state, however, closer to failure. This concept regards the fact that large, unpredictable systems usually do not suddenly quit operations when smaller portions fail, but continue operation with possibly considerable loss on performance. This provides recovery techniques with additional time for actions and can bring the system back on track without complete disruption. The described model is depicted in Fig. 4.

Another problem observed by Clarke and Grumberg [20] is the state explosion problem of large systems with many concurrent processes. Their observation reveals that the number of processes may cause the number of possible states to grow exponentially. The proposed solution to handle all the possible states is to identify common properties. In the case of Alpern and Schneider [6] states are aggregated according to patterns in the execution history or in Clarke and Grumberg [20] according to equivalence classes for the running processes.

2.4 Self-healing policies

Influenced by AI research on human behavior [66,74], Norman et al. [62] propose a three level model based on reaction, routine, and reflection. In this model, the three

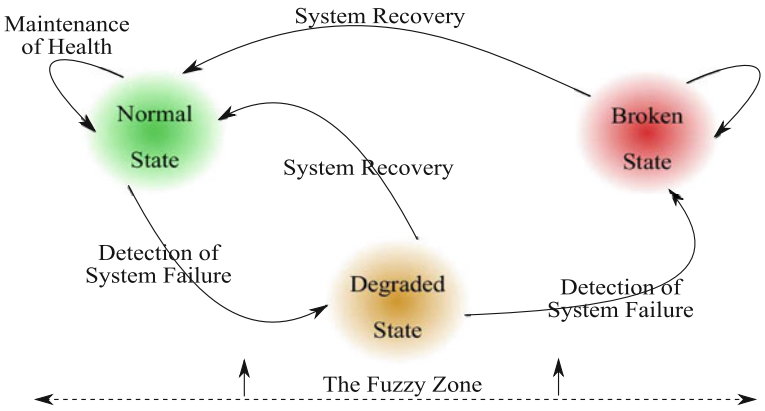

Fig. 4 State diagram of self-healing (taken from [31])

levels differ in depth of processing involved between evaluation of surrounding world (affect) and interpretation of world (cognition). Later, Kephart and Walsh [49] define three different types of policies: Action, Goal and Utility Function with increasing behavioral specification that correspond the three previously presented levels. The policies related to the corresponding model level are the following:

Action Policies (Reaction): This is a type of policy that dictates an action to be taken on a certain occurrence, similar to an IF(Condition)THEN(Action) statement. Likewise, the reaction level is defined as one where no learning occurs and immediate response is expected.

Goal Policies (Routine): These policies define a desired state, respectively, a set of states. This implies that the system must calculate a situation depending set of actions to make a transition from the current to the desired state. Akin to this the routine level is defined as one, where largely routine evaluation and planning behaviors takes place.

Utility Function Policies (Reflection): As a generalization of the goal policies, utility function policies connect a value to each possible state that is adjusted at runtime, depending on the current state. The reflection level is described as self-aware. It deduces the results for problem solving from information of its history, system capabilities, current system state, and current environment state.

A prototype evaluation presented by White et al. [81] observes that goal-driven and utility function policies can be key elements to achieve a degree of self-management. Self-healing research considers recovery as a solid planned process. Simple reactive behavior might not be sufficient for the scope. Instead, to recover and also maintain the system several possible options must be balanced. The result of self-healing policies is a directly or indirectly caused set of actions moving the system towards a safe state.

2.5 Failure classification

Failure classification and root cause analysis is a challenging task in computer networks with a complex compositions. Only a classification and identification of the

failure allows to deploy adequate recovery strategies. Failures can affect single units or whole portions of the systems and the two types can provoke each other because of the dependencies. However, general classifications of failures are available in self-healing related research. The occurrence of a failure is generally defined as an event at runtime where the current system behavior deviates from the intended. Ghosh [32] provides a comprehensive fault classification for fault-tolerant systems. Coulouris et al. [22] provide a classification of faults in regard to distributed systems. Table 1 represents a summary of the identified classes relevant to self-healing research.

Another, more general, fault classification is provided by Kopetz [53] which partitions failures into dependent on value or timing by nature. A failure can be recognized (failure perception) consistently by all affected parties or in the worst case only inconsistently. The second type is also named the two-faced, malicious or Byzantine failure type and can only be recognized by $3k + 1$ components (k representing the number of tolerated failures). A system can deal with the effects of a benign failure, whilst a malign failure exceeds the recovery capabilities and may cause total failure. Finally, a failure can be identified by the number of occurrence in a given time interval. Permanent failures occur only once and remain in faulty state until repair. Transient failures recover themselves and can appear repeatedly.

It becomes clear that especially in large, arbitrary systems failure detection and immediate classification in most cases is not a straight forward process. A crash failure, e.g, might be classified as an omission failure because local detection is not available or affected by the failure. Thus, a detected failure might be the result of another. The columns *Possible Detection* and *Possible Resolution* in Table 1 can only list some well known and possible methodologies for classification and resolution. However, because of the many interdependencies in large systems and possible false detection and recovery strategy, self-healing technologies rely on the state model presented in Sect. 2.3. Remaining in a fuzzy state with degraded functionality, self-healing applications select among several strategies depending on the current state retrieved by the feedback loop and history information until healthy state is achieved.

2.6 System fitness and evolution

Designing a complex system for long-term service can prove to be challenging. Firstly, many requirements are not known a-priori but expose over time. As part of the unexpected behavior, self-healing systems also need to support evolutionary contingent, necessary and intentional adaptations. Secondly, a variety of malicious occurrences might restrict system's functionality to the essential services without the capability of the self to return to normal performance. Additionally, resource usage might exceed the limits by dynamic requirements. Therefore, Kephart and Chess [50] retreat from a fully autonomous vision and consider human assistance to the adaptation loop in the evolution of autonomic computing environments. Ghosh et al. [31] point out that part of the self-healing research also recognizes human intervention as a means to adjust and restore. They coin the term assisted-healing to describe the situation. Salehie and Tahvildari [72] explain the possible options and trade-offs. The limits and portions of intentional external influence without interfering the autonomously

Table 1 Failure classes

Class	Affects	Description	Possible detection	Possible resolution
Crash failure	Process	Externally undetectable interruption of a process execution	Local detection methods	State recovery and restart
Fail-stop	Process	Execution is deliberately inhibited on a failure and detected by other processes	Halt on failure property	Stable storage status reconstruction and partition of remaining work
Omission	Process or channel	Message loss generally caused by lack of buffer space (e.g. send-omission or receive-omission), intervening gateway strategies, network transmission errors	Timeout, checksum	Re-route, retransmission
Transient	Process or channel	The instantaneous transparent presence of various self recovering faults disturbing other parts of the system	Only side effects	Recovery of side effects
Timing and Performance	Process or channel	Constrained distributed synchronous execution of tasks by a specific amount of time	Timeout (QoS)	Re-assignment of task
Security	Process or channel	The system is compromised by adversary implied malicious behaviour	Behavior dependent	Behavior dependent
Arbitrary (Byzantine)	Process or channel	Any type of failure may occur A process confuses the neighbors by providing constantly individual consistent but contradicting information A communication channel may deliver corrupted or duplicate messages	Process: redundant communication and voting ($3k + 1$), Channel: checksum and sequence numbers	Reconstruction, resend and ignore

adapting mechanism must be estimated. Next, the possible how and where of the interaction interfaces must be discussed. Interfaces must permit an externally launched adaptation to be carefully planned, controllable, and soundly deployed.

3 Implemented approaches and applications

The early years of autonomic computing started a substantial amount of research effort [75]. As a part of the same, a number of self-healing concepts and techniques have

been developed in different application areas. In the following sections we present the features of implemented self-healing approaches classified by areas of research. A short introduction explains the environment with predominant disruptions and emphasizes the motivation for a self-healing implementation. Thereafter, representatives of approaches and their solutions are described in detail.

3.1 Survey index and guide

Table 2 provides an overview of the collected work detailed in the following sections. It is structured in an index with section reference and a short summary of the sections' content.

3.2 Embedded systems

Embedded systems operate in special, constrained environments. Because of these constraints they are usually limited in hardware resource, power consumption, processor speed, and memory size. They must interact continuously with a dynamic environment and are expected to function for extended periods. Another requirement is an appropriate and reliable response to changing environment conditions. Therefore, embedded systems are equipped with sensors and effectors [46]. These characteristics are closely related to the requirements of self-healing systems. However, embedded-systems' constrained nature allows only restricted modes of influence. Thus, at design time, considerable effort is spent to resolve all reliability issues. Reliability considers both, timely reaction to usual events but also detection and recovery from exceptional incidents. The term reactive system has become an alternative term to describe an embedded systems [15]. Critical time constraints reduce the options in recovery and diagnosis. Hence, the common reaction to failures with these types is redundancy by replacing broken parts with standby duplicate spare parts. Consequently, to provide an appropriate amount of reliability, a certain overhead of extra components is required. Self-healing approaches in this area point out that this can lead to suboptimal design and other constraints caused by, e.g., higher production costs, exceeding weight, and higher power consumption.

In their analysis Glass et al. [33] criticize that previous redundancy models with duplicate parts had exactly one resource type dedicated for one specific task. Their solution is to consider multiple mappings between resources and tasks. This means on a network of resources more instances of the same task are running simultaneously, some of them as idle. These are ready to instantly take over in the event of a fault. Later, their self-healing description of a point-to-point network in [34] influenced by the research in [51] explains the recovery strategy as a combination of *self-reconfiguration* and *self-routing*. An idle shadow task receives status updates by the regular tasks in a mechanism called checkpointing. Failure detection is provided by keep alive messages exchanged by task and shadow-task. Thus, the shadow task is ready to take over once messages disappear and a reconfiguration of the network is initiated. This also leads to a necessary re-routing of the task's communication path supported by the combined routing knowledge of the individual nodes.

Table 2 Overview and index

Research area	Summary
Embedded systems (Sect. 3.2)	Demanded by their design and purpose, embedded systems have a long tradition in fast recovery strategies Novel self-healing extensions try to define recovery actions including the whole system instead of just focusing on the recovery of single components
Operating systems (Sect. 3.3)	The dependencies of the various running OS-services can interfere not only on startup but also at runtime. New ideas include a self-healing manager which notes service failures and re-runs stalled services
Architecture based (Sect. 3.4)	With the help of architectural models expressed by description languages differences between runtime composition and healthy model of composition are extracted. Moreover, recovery strategies are deduced by the type of difference
Cross/multi-layer-based (Sect. 3.5)	Approaches in this category organize their resources into layers. The interpretation of the layer concept can be different. One approach uses the layers to avoid conflicts in resource assignment others define the recovery boundaries and priority of a resource by layers
Multi Agent-based (Sect. 3.6)	Agent-based environments provide new opportunities for self-healing extensions. Agents can act autonomously and host self-healing capabilities that guard the assigned system part. Other approaches consider the collaboration capabilities of agents
Reflective-middleware (Sect. 3.7)	The self-reflective property of reflective systems offered new ideas for self-healing techniques on the middleware layer. The approaches in this section highlight managers that transparently combine the properties of reflective-middleware to a self-healing enhancement for applications on top
Legacy application and AOP (Sect. 3.8)	This section presents work that tries to enhance new and legacy applications with self-healing techniques The common idea is to find and provide ideal checkpoints and hooks for monitoring and recovery actions. These features are either provided by the linker or runtime for legacy applications or implemented by the AOP paradigm
Discovery systems (Sect. 3.9)	Self-healing strategies in this category of systems focus on failure resilience. Instead of deploying recovery strategies to the collection of registered resources they try despite of failing resources to keep request latency to a minimum and the query responses consistent
Web services and QoS-based (Sect. 3.10)	The self-healing approaches focus on the process structure or the maintenance of the involved services. In the first type of approaches the monitoring and recovery capabilities of process languages are combined to a self-healing loop. The other type detects degradations by monitoring the communication between the services Recovery tries to balance the service load

A method of exploiting the capacity of all redundant resources on a network is also shared by initial work in [80]. In the regular case, all resources are standby for additional tasks, possibly required during runtime. As in the previous work, recovery in this implementation considers isolating the faulty component by bypassing it. A more extended approach can be found in [3]. In this FPGA approach, instead of only having independent embedded nodes with multiple redundant function units, a *self-configuring master-slave architecture* transparent to applications, relocates failing slaves. Nodes with reduced redundancy are connected to a central manager that maintains a table with an overview of nodes and related assigned tasks. In a self-optimization algorithm new tasks are dispatched equitably to nodes. Once a single node runs out of part's redundancy, it reports the problem to the central manager. Recovery gathers the task's last state from the failing node and assigns the unfinished task to a spare node.

3.3 Operating systems

Soft faults and hard faults are the two categories of faults affecting a running operating system. Soft faults are considered application crashes that can be recovered. Opposed to that, a complete system restart is required on hard faults caused by broken hardware or faulty drivers often resulting in blocking I/O exceptions. According to Tanenbaum et al. [77] the main objective of self-healing in operating systems is, aside from failure resilience, to avoid faults requiring a system restart. These include mainly two possible methods: (i) release the supervising kernel from dependencies and (ii) free allocated resources and rerun the failing applications. This second concept is also applied in recovery-oriented computing and an influential approach is found in Candea et al. [16]. The recovery by rerun scheme is described as recursive microreboots that reboot units recursively according to the dependencies hold in the referencing reboot tree.

The first solution to minimize and stabilize the operating system kernel is presented in the example of the Minix3 Operating System.[2] The precaution taken in this system design is an isolation of all system drivers from the kernel. External access to the kernel data structures is granted in read-only mode. In Herder et al. [38] two special server (services) *reincarnation and data server*, handle recovery. The reincarnation server is a parent process to all other processes and notices through this relation if a child thread fails. The data server holds configuration and status information. Recovery policies are stored for every component and the common replaces the failing application with a fresh copy.

The other prominent idea is a predictive self-healing mechanism described in Shapiro [73] and implemented in Solaris-10.[3] Proactive in this implementation means a clever diagnosis that can anticipate major degradations. At the core, an operating system service called the *fault manager*, polls the messages provided by the system logging facilities. It then dispatches the messages to the matching *diagnosis engines* for the several fault classes, e.g., CPU, memory, I/O, and applications faults. Each diagnosis engine attempts to anticipate and diagnose a possible problem and trigger

[2] http://www.minix3.org/.

[3] http://www.sun.com/software/solaris/.

an automated recovery response. A recovery response manages the coordinated rerun of the affected applications. Coordination is essential, because operating system service providing applications, such as daemons might have interdependencies with other applications and daemons. A plain restart of a misbehaving one can result in others failing. Therefore, this approach handles dependencies as contracts. These are managed by a dedicated service, the *service manager*. The service manager is consulted by recovery for a correct order of possibly multiple restarts.

3.4 Architecture-based

A common diagnosis method in self-healing compares current values of critical system properties to pre-estimated "healthy" value constraints. This fact was recognized by system architecture-based research and expressed by the idea of keeping an updated notion of the guarded system captured in an architectural model. The real environment is mapped to a model of resources and dependencies and kept consistent. The complexity of reality is usually reduced by abstracting only the substantial properties (e.g., elements, interaction patterns, performance expectations) of the environment and expressed in an architectural description language (ADL). Monitoring and recovery facilities are assumed to be supported by a properly equipped middleware underneath (possibly in conjunction with, e.g., reflective middleware see Sect. 3.7). At runtime, these approaches need to handle three major tasks. Firstly, the sensed data must be translated for comparison to the last known model. Next, possible violations of ranges caused by changes must be recognized using the model. Finally, on a violation, diagnosis deduces the according correction model, translated back to required recovery actions.

The approach in Dashofy et al. [24] describes the system under consideration in an XML-based architecture description language (xADL 2.0). During diagnosis, the *ArchDiff* tool takes current and desired architecture as input and outputs the discrepancies in an *architectural diff*. This diff is mapped to a recovery description. Dedicated *design critics* collaborate and pre-estimate any violation to integrity by checking the recovery description. Only correct descriptions are applied. Cheng et al. [18] illustrate an adaptation framework, following the model of the MAPE loop. The state recognition in this approach is handled by a service called *Architectural Manager*. It comprises tools for abstraction of the monitored feedback and relation to the properties of the architectural model. An evaluation determines if the system is still in acceptable ranges. Finally, a repair handler adapts the model and propagates the changes to the running system.

The model adaptation concept is also shared by Cheng et al. [17,19]. These exploit the advantages of a quite reliable and complete view of the underlying environment. They not only adapt for recovery, but support system reconfiguration and evolution at runtime by policy adaptation. In the layered structure in Cheng et al. [17] the layer above the *Model Layer* can set overall system objectives regarding, i.e., the quality of service. In Cheng et al. [19], the authors extend the architecture-based self-adaptation of the Rainbow framework with an adaptive learning approach. The system records and remembers (learns) human administrator decisions in critical situations. Such derived action scripts are called tactics and related to self-healing policies.

3.5 Cross/multi-layer-based

Adaptation focused on more than one resource is a strength of self-healing techniques. The approaches in this section additionally consider coordinated adaptations on different system layers. The most prominent areas of application described in [2,83] are multimedia applications and unreliable networks [47,68,82]. A starting point is separation of concerns. Minor functionalities and responsibilities are partitioned into the layers. An overview layer, manages the required adaptations and keeps an updated view of the status on the different layers' resources. Therefore, this layer must be informed of the latest status in addition to all adaptation features of the subordinated layers.

In the GRACE project [2,83], a *resource coordinator* assigns spare resources (network and hardware) to the requesting applications. Once reserved, the application regulates on its own the allocation and reallocation of the assigned resource. The reasons for a request can be conflicting and contradicting and are resolved at the coordinator. In a predictive manner the coordinator tries to avoid conflicts and must balances between cost and overall utility, with the aim of optimizing utility and lowering cost.

A different understanding of multi-layer adaptation is found in wireless and ad-hoc networks. The layers referred to in these works are the layer of the ISO/OSI-7-Layer model. The concept in Kant and Chen [47] presents a self-healing multi-layer method with policies containing *hierarchical survivability requirements*. Therefore, the dependencies of critical and prioritized services are restored first at all layers. In a following work, Kant and Chen [48] realize that the same multi-layer recovery result can be achieved with a combination of different adaptation strategies on several layers. This approach selects the adequate one for each layer depending on the delay expectations of the affected parties.

3.6 Multi agent-based

Agent-based systems, similar to the previously presented embedded systems, provide robustness by redundancy. The difference is that embedded systems are usually encountered in closed environments with simple behavioral patterns. The strength of agent-based approaches is that by design they can handle unexpected situations in environments with unpredictable behavior. An agent is responsible for its entrusted environment and must be self aware and adapt to support overall objectives [45]. Furthermore, to meet the objectives, agent-based systems consist of multiple cooperative, persistent agents [40].

Corsava and Getov [21], *intelligent agents* or "intelliagents", are advised to diagnose the behavior of different classes of resources in a cluster landscape. In particular, different intelliagents monitor hardware, operating systems, and network resources. An agent's structure has five major parts. Apart from the common ones including monitoring, diagnosing, and repair, intelliagents collaborate with each other by communication. They log their activities and try to self-maintain. The design of an agent is faithful to the initial ideas of autonomic computing. Intelliagents are mainly deployed to support the human administrator in daily tasks. Thus, detection results are always

logged and the recovery attempt is simple. Recovery stalls the execution on affected resources, tries to find a fail-over unit and restarts execution with the saved state. This has the advantage that external influence and help by the human administrator does not confuse the self-healing system.

But agent robustness not only derives from agent redundancy. Equipped with the same algorithms, the agents are contained with the same flaws and most probably fail successively on the same defect. Instead, one important notion in agent-based software and self-healing, is robustness via heterogeneous implementations.

An interesting path is followed by the ideas presented in Huhns et al. [43]. This paper considers its contribution to self-healing in the aforementioned robust redundancy. In contrary to the redundancy by identical parts, the outlined concept means both, *redundancy by diversity* of implementation, and *redundancy by voting* for a correct result. Furthermore, the voting method can be applied as a pre-selection of the most suitable implementations, a selection of the most accurate result at the end; or a combination of both. Two architectural approaches are considered. In the centralized one, only one agent collects the capabilities of all other agents and decides which are the most suitable for an incoming task. At the end it decides on the best result. The distributed approach is more complex and includes a gossip communication between the agents. Again, a pre and post process voting is possible.

A centralized approach can be found in Tesauro et al. [79]. The developed software architecture called Unity, aims at *self-management by component interaction*. Components are autonomic elements appointed to control the resources. The presented self-healing method involves only elements that serve as policy repositories. The approach allows human administrators to set the system objectives as policies. The hierarchy comprises a bootstrapping arbiter on top that deploys several redundant registries containing the policies into a cluster of resources. Deployed registries provide the elements managing the cluster resources with policy updates. Once a registry fails, the arbiter will temporarily reassign all activities of the failing to a still running registry. Meanwhile, the arbiter determines an ideal host in the cluster to deploy the replacement repository. In this scenario, the arbiter represents the voter and the new host selection the diversity.

An elaborated agent approach is available by IBM's APLE, a Java based agent framework [12]. This framework builds on the idea of blank base agents that can be extended with functionality and bound together to a new agent, thereby enabling task splitting among the sub-agents and divers task handling by different extension composition. The base of the framework is formed by the *AbleAgent* composed by components called *AbleBeans*. A collection of interconnected beans define the agents specific function. A component library is an archive for the different types of beans. Three types of beans are distinguished. *Data beans* access and transform data. *Learning beans* contain learning algorithms and data maintaining capabilities. *Rule beans* define sets of rules in a specific rule language. On the highest level the agent platform comprises services that provide life-cycle management (create, suspend, resume, quit) for agents as well as inter agent communication. In a final section, the paper presents the layout of the *Autonomic agent*. This is a composed agent platform that mimics an autonomic manager by combining required functions with a collection of dedicated agents. As a special contribution three levels of reaction, reflexive, instinctive, and

learned behavior are added to the autonomic agent as separate agents (*AbleSubsumptionAgents*). These apply to the three types of policies of Sect. 2.4.

3.7 Reflective-middleware

Self-representation is the main feature of a reflective system. In detail, this describes an interface with two main features. It enables analysis by queries on structure and on system states, and adaptation by reconfiguration actions. This requires a stringent connection between representation and the system itself. Maes [56] specifies the connector of a reflective system causal, meaning that once internal structures change, they must also change in the reflective representing domain and vice versa. Starting with the adoption of architectural reflection in programming languages, reflection lately was also introduced into middleware and includes such projects as DynamicTAO [52], OpenCORBA [54], OpenORB [13], to name a few.

Blair et al. [14] explain the relation between OpenORB self-healing techniques. The main idea is to resign to one of the main features of middleware, the transparency. Instead, openness provides authorized applications, via meta-models (*resource meta-models and interception*) access to configuration means to control structure and behavior. On the one hand, OpenORB provides structural reflection, including interface meta-models for external representation and architectural models for internal representation of a component. These are a representation of the underlying system structure and support adaptation by the knowledge of components and their interconnections. On the other hand, behavioral reflection, composed by interception meta-models, enables dynamic inspection of data-flow monitoring and manipulation, and resources meta-models offer access to resources and their management. The presented approach considers self-healing comprised by monitoring and adaptation *management components*. These are dynamically introduced into various meta-space models. They provide an interpreter for timed automata to control at one side, and interfaces to the other components on the other. Predefined policies in the form of timed automata, guide the management components in event registration and handling, and, if necessary, in adaptation.

A complete self-healing loop for reflective middleware is outlined in Hong et al. [41]. This work considers self-healing as one of the computation units atop of the reflective middleware. The outlined system comprises, in the notion of computation units, the runtime as basic computation level monitored by the reflective computation level. Above that, a self-healing computation analyses and plans changes. The required adaptations are then propagated through the reflective computation down to the basic computation.

3.8 Legacy application and AOP

This section considers self-healing approaches that enhance existing and new application software with recovery structures. The goal is to capture any possible exceptions, to root analyze the fault, and to overcome the interruption, supporting application continuance.

The first presented representative tries to support established legacy applications. The challenge is to find a way to access and control, however only soundly or transparently interfere, with the regular mode of operation. In contrast to the previously presented middleware approaches, sensor and actuator interfaces are not available from the start. Instead, ideal access points must be located. The presented solutions add checkpoints or hooks to the application's linking mechanism or runtime environment to monitor and control the program-flow. The work in Griffith and Kaiser [35] is an initial idea of how this method could work in a .NET runtime. The Just In Time (JIT) compiler provides an entry point to possible execution flow adaptation. Overwriting the execution pointer address before or after a method call could redirect the flow to an external repair engine in the case of a failure. The Java based approach in Fuad et al. [28] injects hooks at strategic points enabling runtime interaction with the managed code. Injection means *introspect and retrofit* existing byte codes with Javassist[4] bytecode manipulation at strategic checkpoints. Additionally, every method is finalized with an extra try-catch block, catching any runtime exceptions. On an exception, diagnosis estimates the root cause. Recovery tries to reinitialize the code at the failing point and execution can continue at the last known state.

The event-based extensible framework in Abbas et al. [1] provides program-flow interception and redirection of linked applications, by hooking into the dynamic linker and altering jump addresses if necessary. The approach extends a common linker with dedicated hooks. Hooks monitor the symbol look-up at the linker and trigger events in a callback to a service called *Dynamic Dynamic Linker (DDL)*. This service has an API for external extension and provides a redirection library that can dynamically adapt the application linking.

Aspect-oriented programming (AOP) provides access points to monitor and alter behavior of a running application already at design time. AOP is based on joint points, which similar to the previous hooks and defines a point in the program flow, like a method invocation. Joint points associated advices can then execute defined methods before, during, and after an identified situation.

As an example, the approach in Haydarlou et al. [37] considers AOP as a method for self-healing. Detection comprises two stages. Sensors are instrumented in all strategic components (objects) and boundaries (interaction points) of the application. Sensors are passively waiting for failure alarms. A status model is actively gathered storing system structure and changing states. Analysis uses information on events, method invocations, and state changes, to recognize failure patterns. Recovery planning in this implementation covers three stages comprising all three types of policies presented in Sect. 2.4. Well known failure patterns are easily recognized and handled in reflexive manner by the *Reflexive Planner*. If this type of planner fails, a deeper analysis estimates the root cause for the *Instinctive Planner* that selects a more complex recovery plan form a repository. Finally, if the planner also fails to select a strategy, a *Cognitive Planner* equipped with human interaction interface and machine learning algorithms creates a new complex recovery plan that is executed and saved to the repository.

[4] http://www.csg.is.titech.ac.jp/~chiba/javassist/.

Finally, the work in Alonso et al. [5] considers supporting self-healing with a fine-grain monitoring framework architecture. This AOP-Monitoring Framework is composed by sensors and the *Monitor Manager*. The sensors represent aspects and deliver the manager with collected information. The monitor analyzes the data and determine an appropriate action policy. It anticipates failures by using data mining or forecasting methods using the sensor collected data. In detail, it can dynamically activate and fine tune sensors at a suspicious spot to precisely investigate the situation.

3.9 Discovery systems

Discovery systems must provide a consistent view of distributed components. They are designed to support the discovery of different resources with a variety of distributed applications on nodes with distinct requirements specified by individual semantics. Such systems must be highly available, scale, and be up to date on the network's status. Failure sources include the unreliable behavior of the resources that suddenly disappear, crash, recover, and then possibly later reappear. Hence, discovery systems and self-healing systems handle environments with similar behavior.

Dabrowski and Mills [23] discuss this fact. Similar to self-healing's aim for continuous availability, service discovery primarily focuses on consistency maintenance of discovered resources and answer client's resource queries at any time. To succeed, updates to service descriptions are fetched either by active polling of the resource or by subscription to notifications on changes. Failure discovery is described as combination of two stages. At a first stage, heartbeat messages in the form of scheduled resource announcements are monitored. If these disappear, in a second stage, the monitor actively polls the suspicious resource waiting for an acknowledgment. On a recognized resource failure the recovery strategies base on persistence at two levels. The first, *soft-state persistence* includes preliminary discarding knowledge from the registry, considering a temporal failure. The second is *application-level persistence* and counts on the resilience of individual applications using a failing service.

Noticeable, this work contains no description of any recovery methods deployed by the registry. Rather, because of the system's complexity and heterogeneity, there cannot be one defined remedy method for all resources. To scale, this cannot be the concern of the discovery service of the system. On the contrary, the healing methods must be applied to the individual resources and it is their responsibility to announce availability after recovering.

Hence, the main effort in discovery systems is failure resilience, meaning that request latency is kept to a minimum and lookups remain consistent. In wide-area unreliable networks, the trend has gone towards tree lookup in form of distributed hash tables (DHT). DHT manage their knowledge in structured graphs. Neighbor nodes share partly the same information about resources. A node failure is recovered by firstly broadcasting the fact to all concerning neighbors and than selecting the best fitting substitute [69].

This method for self-healing discovery systems is also used by the approaches in Albrecht et al. [4] and Rilling [70]. Both base the success of their self-healing on the recovery strategies on DHTs. The first describes a resource discovery service for wide-area distributed systems named SWORD. The results confirm the claim

that services managed by DHT automatically inherit the DHT's *self-configuration, self-healing, and scalability properties*. The Vigne self-healing architecture for grid operating systems presents an environment for distributed decentralized control of applications. It accomplishes recovery and fault transparency relying on the underlying DHT routing mechanism.

The self-healing network (SHN) [8] for runtime environments uses the notion of neighbors for failure recovery. SHN belongs to the family of discovery and resource management systems and aims for high scalability. Recovery is similarly enabled by nodes organized as neighbors in a tree structure, informing each other on detected changes. It bases on the authors' previously specified scalable and fault-tolerant protocol (SFTP) [7], that use the neighbor nodes to reroute the messages in the failure cases.

3.10 Web services and QoS-based

Services have gained importance not only in research (Grid-Computing, Workflow engines) but also in business applications (Google Webservice,[5] Amazon Web Services[6]). As the W3C Working Group Node[7] on QoS for Web Services explains, because of the dynamic and unpredictable characteristics of Web services, one of the major challenges of service providers is to guarantee a certain quality of service (QoS) towards the expectations of their clients. Out of the desired properties of WS QoS listed by the W3C performance, reliability, robustness, exception handling, integrity, availability are the most related to self-healing. On the one side, a contract (e.g., SLA) binds the provider to a certain degree of quality promised to the requester, on the other side, the provider aims to optimize profit from the service by any means available, namely exploiting the best all available resources. While the optimization side is more subject to other self-* research, the maintenance of a certain degree of QoS is an entry point for the presented self-healing approaches in this subsection. Web services are failure affected in a similar way as discovery services. Moreover, the failure of just a single service can cause a significant degradation in performance because of the service interdependencies. This leaves WS recovery with two possible options. Either a reconfiguration is done internally on the composition of the WS workflow, or a reconfiguration requires architectural changes of the services' landscape itself.

BPEL engines, providing the common workflow engines of today, comprise three handlers as interfaces for recovery. These are fault, compensation, and event handlers. Fault and event handler are activated during execution and provide status information which can be monitored and diagnosed. For recovery, the compensation handler can be invoked after execution of the faulty activity. However, these handlers are only interfaces and it is up to the designer to combine the methods for a successful recovery.

This lack is recognized by the work in Modafferi and Conforti [58]. It presents a non-intrusive approach, which instead of implementing a new engine or extending the existing one, extents the design language of BPEL workflows with proper

[5] http://code.google.com/apis/ajaxsearch/.

[6] http://aws.amazon.com/.

[7] http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/.

annotations. A complementary pre-processor converts the definitions back to BPEL workflows extended with optional recovery operations. The imagined recovery actions are in detail, the modification of process variables, the specification of deadlines for tasks, the ability of re-doing a nested portion of the flow (task or scope), the possibility to reroute the flow path, and the return to a safe point in the process.

Modafferi et al. [59] extend a standard BPEL engine (BPEL4WS 1.1 compatible) with management access composing it to a Self-healing BPEL (SH-BPEL) engine. An additional *Process Manager* hosts the BPEL engine. The manager controls the engine through two interfaces. The Process Management API supports operations on the process instances and the Engine API allows direct influence of the engine. In addition, the manager can intercept the message exchange between BPEL engine and Web services and, transparently to the BPEL engine, influence and replace the Web services. This already relates the second option of the WS recovery methods. Finally, the manager stores action policies that react to notifications and provides itself a management API that allows external applications to trigger policies.

Another extension of an existing engine is presented in Subramanian et al. [76]. The SelfHealBPEL engine interfaces with the ActiveBPEL engine. The extension provides a recovery policy definition sh-policy for any activity. Detection assures that pre- and post-condition and non-functional properties of a tracked activity are satisfied and then on execution monitors unexpected exceptions. Any deficient activity is reported to diagnosis. This module suspends process execution. Diagnosis' recovery commits the plan to process creation and management of the ActiveBPEL engine.

ActiveBPEL is also subject of extension in the work presented in [10,11]. This self-healing solution named Dynamo (Dynamic Monitoring) bases on reusable sets of *supervision rules*. These rules are organized in monitoring locations in the process, supervision parameter as detection rules, monitoring expressions stating diagnosis pre- and post-conditions, and reaction strategies for recovery. Priority levels associated with supervision rules guarantee that from the bunch of rules connected to a location only those with a certain priority are executed. Recovery strategies are composed of atomic actions called strategy steps. Strategy steps can be combined and executed alternatively (by an "or" statement) or together (by an "and" statement). This self-healing technique is applied by an AOP-extended ActiveBPEL with a Java based evaluator and rule processor.

A service related approach of a self-healing infrastructure in the work of Moo-Mena et al. [60]. They consider non-intrusive communication flow QoS monitoring for service degradation detection. An interceptor measures the response times between requester and provider. Three types of information sources help the diagnosis in decisions. The first one, the *Parameter Repository* serves as a log for the QoS measurement. The second, the *Service Level Parameters* contains thresholds for the QoS of the single interactions. *WS Table Access* keeps current service transaction state information. By monitoring agreement degradations (e.g., via SLAs) combining the information of current QoS measurement in the first source and expected in the second, the diagnosis decides whether to initiate recovery. The main recovery method of this approach is to instantiate a new service of the affected service's type. Depending on performance degradation or service malfunction the new service rule is additional service or replacement. In both cases, the interceptor reroutes the request to the new service.

A combination of reactive and proactive self-healing is described in the work of Halima et al. [36]. The work presents a self-healing middleware called QoS-Oriented Self-Healing (QOSH), that enhances SOAP messages with QoS metadata to monitor QoS degradations. This is an example of an application non-intrusive monitoring using, as frequently used, the response time to rate QoS. In this implementation analysis is split into diagnosis and prognosis. Whilst diagnosis is reactive to alerts, diagnosis tries to predict service deficiency by analyzing the history of QoS measurements. In either case, the reaction is a reconfiguration of the services. To balance possible local adaptation over-reactions, the middleware is also equipped with a global diagnosis and can identify the source of degradation more precisely, inhibit useless adaptations, and optimize the overall healing effort.

Another services related adaptation approach can be found in the paper by Moser et al. [61]. The Vienna Dynamic Adaptation and Monitoring Extension (VieDAME) focuses on BPEL controlled service invocations with a high demand on availability. VieDAME is split into two parts: the core and the engine adapters. The core comprises monitoring, service selection and message transformation. The engine adapter interacts with the BPEL engine. An adaptation cycle includes the monitoring (SOAP calls) and evaluation of various QoS attributes of services towards constraints. Then in the service selection process the best fitting among "partner" services for the same task is selected. Finally, a message transformation adapts the service invocation messages to the selected service interface. Implemented with AOP methods, the system represents a non-intrusive approach separating the BPEL engine from the adaptation engine.

4 Discussion of approaches

The principles of self-healing systems were described in Sect. 2. On the base of these, the following sections extract the common and distinct characteristics of the researched work. First, common characteristics are discussed. In line with the structure of Sect. 2.2 the behavior in the three dominant stages is examined. A summarizing table in Sect. 4.7 gives a comprehensive overview and concludes the discussions.

4.1 Separation of concerns

As explained in the principles (cf. Sect. 1.1), most self-healing approaches recognize their main task in resolving the system's complexity. The well researched *separation of concerns* concept of the software engineering community [64,78] is considered a viable method for providing some flexibility and comprehensibility.

In the researched works, flexibility is usually gained by introducing a layered or master–slave structure. A higher, global view, gains an updated, current view of the system hold in tables or models. A local layer acts as an interface to the context and comprises sensor and effector units. All three stages of self-healing implementations depend on an appropriate current view. Detection requires a reference to distinguish between the self-healing states. Diagnosis selects the correct recovery policy according to the current requirements. Recovery supervises the adaptation process with feedback from the context. The most important aspect of an accurate view is that it allows to

involve the whole system into the adaptation process. Recovery does not only depend on local recovery methods but strategies can consider all known system capabilities. Adaptation might shift the failing task from a destabilized to a stable portion of the system. Increasing load can be balanced fair among all available resources.

An important result of the separation is that it augments the quality of the recovery. Quality consideration include a better traceability and more precise identification of faults as well as a widespread, prospective fault recovery. Failure recovery not only depends on simple strategies but aims for a global resolution and a clever reconfiguration to avoid future disruptions. A user is provided with more reliability and reliance. Finally, a separation of concerns lowers also the probability of an error-prone impact of evolutionary changes. Hence, system's maintenance is facilitated; reuse and evolution guaranteed. Minor parts, added, exchanged, or removed, issue only negligible impacts.

4.2 Intrusive versus non-intrusive

Self-healing approaches often extend existing, well established systems. Therefore, the requirements for an integration are delicate and aim for a smooth alignment. Two modes of integration are considered appropriate.

In an intrusive manner the original system must be adapted to support self-healing extensions. Methods for gaining system awareness and allow adaptations need to be recognized, elaborated, and implemented. The constructed solution might interfere with the original design and alter the course and timing of data exchange and logic decisions. Different, possibly faulty, results must be taken into account and compensated. However, apart from these disadvantages, a thoroughly planed and strict coupling with the guarded systems can guarantee an optimal integration in the time domain. Detection becomes more accurate and adaptations are timely and precise.

A non-intrusive alignment of self-healing techniques respects the guarded system as a complete unit. Adaptation and monitoring are limited to the optional interfaces possibly provided by the system. If no interfaces given, diagnosis can only evaluate the interaction between the system and the environment. A recovery strategy includes only adaptations of the system's surroundings. Non-intrusive might be the preferred way of integration, however the less applied. Its efficiency depends fully on the capabilities and characteristics of the supported system.

4.3 Closed versus open

As apparent from the principle section, stabilization is one of the main goals of self-healing systems. However, guaranteeing stability is an almost impossible task in systems with unpredictable behavior. Therefore, some of the self-healing designs try at least to avoid all a-priori known failure sources. The concept of a strictly closed-loop presented in Sect. 2.2 is adopted by most of the reviewed self-healing approaches. The result of this straightforward concept is a clearer understanding of the operation mode. As a drawback, little configuration options limit the system's adaptations for future requirements. However, self-healing techniques are usually aligned to systems to enhance the long term use. Thus, some of the researched

works introduce at least indirect influence by allowing dynamic handling of policies. Fewer consider direct influence via loop connected interfaces (e.g., UI with control unit).

4.4 Detecting and reporting suspicious behavior

Fault detection in self-healing systems is accomplished by recognizing degradations. In the examined work, different types of approaches are applied. A first general distinction separates approaches that actively search for inconsistencies in the sensed data and others that are triggered by inappropriate behavior. Both take advantage of the layer or hierarchical structure of the system. A lower level monitors and prefilters the data-flow. Detection analyses the event and on suspicion alerts diagnosis. All approaches support detection by an exclusive system knowledge. Knowledge comprises, e.g., system models, component dependencies, special notifications, log information, monitoring data enhancements.

4.5 Diagnosing and policy selection

Self-healing systems are recovery oriented. An elaborate recovery strategy requires an extended diagnosis. Only one of the studied approaches relies on action policies. In accordance with the evaluation in White et al. [81] mentioned in Sect. 2.4, the rest of the approaches considers at least goal driven polices, minor rely on utility functions. In practice, a combination of more than one policy type is often observed. Comparable to the separation of concerns, most approaches equip their external parts with reactive action policies as a first fault containment. Supported by detailed system status information a centralized planning additionally elaborates goal or utility oriented strategies.

4.6 Recovery techniques

An appropriate amount of redundancy affecting both hardware and software allows a combination of different types of recovery. While hardware redundancy assists self-healing implementations only with spare parts as duplicates or additional resources, software and application redundancy can also include implementation diversity or relocation of services. A detailed list of the redundancy types discovered in the researched approaches is the following:

replacement: The faulty hardware part is replaced with a duplicate spare one. For application software this means a rerun by killing of the faulty instance, freeing possible still allocated resources, and starting a fresh new application instance.

balancing: The degradation is caused by load. In this situation recovery can temporarily include extra application instances or spare hardware parts to sustain the load, and when safe, free them again.

isolation: Cuts of a failing part of the system to assure that its malicious behavior does not infect the other parts.

persistence: Assumes that an occurring defect causes no further degradation and is carried by all parts. This can be seen as a “passive” recovery method. The failing

part is ignored and has to provide its own recovery actions if it wants to joint the system again.

redirection: Changes the task-flow on a failure to a recovery routine and then back to the original flow.

relocation: Moves an application or task from a failing to a different host re-directing also the possibly affected communication and data flow.

diversity: Switches to a different approach (algorithm) to solve the task once, one is considered to fail.

4.7 Overview

Table 3 represents a comprehensive overview of the examined areas of self-healing research. A reference links to the appertaining approach. Principle and NI/IF (non-intrusive and self-healing loop interface) introduce the approach. The values in NI/FI refer to the explanations of Sects. 4.2 and 4.3, respectively. Detection criteria lists the sources expected to reveal degradations. Monitoring is described in the mode column. Diagnosis is split in support, Pol./prog. (Policy and prognosis). Support denotes the basis of diagnosis considerations. Policy refers to the policy types explained in Sect. 2.4. Prognosis distinguishes those approaches that consider prognosis a property of their adaptation strategies. The recovery technique column aligns the approaches to the implemented recovery strategies presented in the previous section. A "-" indicates that there was no evident information on the related examination column available. Note, for example, that to the global view introduced in Sect. 4.1 most of the examined approaches seem non-intrusive. For architecture-based approaches the subject is even out of scope and non of the work-flow approaches dares to alter the internals of the work-flow engine. The reader is advised, however, that the final overview does not include any presumptions on the extracted categories and indicates, e.g., only those approaches as non-intrusive which explicitly refer to the subject in their work.

5 Summary and outlook

The number of available approaches elucidate that the research on self-healing is very active. The fundamental ideas trace back to the area of fault-tolerant systems. As a particular contribution, self-healing focuses on an elaborated recovery process. The notion of the self and the context as well as a hierarchical separation of concerns provide a reliable base for reasoning and a balanced delegation of the required adaptation actions. Different recovery strategies are considered, combined, selected, and deployed according to accepted side effects and current system status.

This survey gives an insight into a major selection of current and past self-healing approaches. Origins, principles, and theories of self-healing are explained. An introducing paragraph describes the research environment and the motivation for a self-healing approach in the area. An examination of related approaches comprehensively explains the concepts in the line with the stated principles. In a final discussion

Table 3 Comparison of approaches

Overview		Approach		Detection		Diagnosis		Recovery
Area	Reference	Principle	NI/IF	Criteria	Mode	Support	Pol./prog.	Technique
Embedded systems	[34]	Network of coordinating nodes	–/*c*	Node failures	Keep alive messages	Routing tables	*a*, *g*/–	*repl*, *rel*
	[3]	Coordinating master	–/*c*	Node failures	Central manager invocation	Node-task table	*a*, *g*/yes	*repl*, *rel*
Operating systems	[38]	Reincarnation	–/*c*	Child failure	Thread messaging	Data server	*a*/–	*rep*, *per*
	[73]	Dedicated OS-services	yes/*c*	Log monitoring	Log examination	Diagnosis eng., dependency mgr.	*g*/yes	*rep*
Architecture-based	[18]	Architecture mgr.	–/*c*	Model comparison	model diff	Model capturing resources and dependencies	*g*/–	–
	[19]	Architecture mgr.	–/*o*	Model comparison	model diff	Model capturing resources and dependencies	*u*/–	–
Cross/multi-layer	[2,83]	Resource coordination	–/*o*	Resource requirement	Allocation request	Resource allocation overview	*g*/yes	*per*, *rel*
	[47]	Network management system	–/*c*	Alarm model	Alarms	Survivability requirements	*g*/–	*per*, *rel*
Agent-based	[21]	Agent collaboration	–/*o*	Predefined subjects	Resource monitoring	Dedicated knowledge agent	*a*/–	*per*, *rel*
	[79]	Replaceable agents	–/*o*	Application, network failures	Sentinel notification	Central arbiter	*g*/–	*per*, *rel*
	[12]	Extensible agents	–/–	–	–	Autonomic agent	*a*, *g*, *u*/–	–
Reflective-middleware	[14]	Reflection	–/*c*	Meta-models	Configured by timed automata	Mng. components with timed automata	*g*/yes	–
Legacy appl. and AOP	[28]	Checkpoint setting	–/*c*	Exceptions, data-flow	Hooks	Program-flow mappings, exceptions	*a*, *g*/–	*red*
	[1]	Checkpoint setting	–/*o*	Exceptions, data-flow	Hooks	Program-flow mappings, exceptions	*a*, *g*/–	*red*
	[37]	Joint points and advices	–/*o*	Aspects	Failure alarms	AOP	*a*, *g*, *u*/–	*red*
	[5]	Joint points and advices	–/*c*	Aspects	Finetune sensors	AOP	–/yes	–

Table 3 continued

Overview		Approach		Detection		Diagnosis		Recovery
Area	Reference	Principle	NI/IF	Criteria	Mode	Support	Pol./prog.	Technique
Discovery systems	[23]	Shared redundant location information	–/*o*	Resource failure	Polling or notification	TTL	–/–	*per*
	[4,70]	Shared redundant location information	–/*o*	Resource failure	–	DHT recovery	*g*/–	*rel*
	[8]	Shared redundant location information	–/*o*	Resource failure	–	SFTP	*g*/–	*rel*
Web service and QoS-based	[59]	BPEL compensation handler	–/*o*	QoS and SLA violations	Message exchange, engine API	Process manager	*g*/yes	*red*
	[76]		–/*c*		Condition tracking	Sh-policy	*g*/–	*red*
	[10]	Supervision framework	–/*c*	Rule violation	Location monitoring	Supervision rules	*g*/–	*red*
	[60]	QoS and SLA violations	yes/*c*	Agreement degradations	Interceptor for response times	Reference values for measurements	*g*/yes	*red rel*
	[36]		yes/*c*	SOAP enhancement			*g*/yes	*red rel*
	[61]	QoS and high availability	yes/*o*	SOAP invoke calls	Aggregated invocation performance	Various QoS attributes	*g*/yes	*red rel*, *bal*

In column NI/IF (non-intrusive/interface) *o*pen, *c*lose; in column Pol./prog. (Policy/prognosis) *a*ction, *g*oal, *u*tility; in column Technique *rep*lacement, *rel*ocation, *per*sistence, *red*irection, *bal*ancing

differences and similarities of the researched material are summarized and an overview table allows for approach comparison and evaluations for future improvement.

Large-scale and unreliable system accumulations have become reality and a means to handle the unpleasant repercussions are considered important. The spectrum of solutions is wide and still not completely exhausted. Paramount, fault-tolerant system research has established a fixed position on the subject with the concepts of failure transparency and self-stabilization algorithms. Survivable systems research recognizes that for malicious intrusions with unpredictable impacts exist no simple fault recovering formulas. Self-healing research inherits from both and combines its capabilities to isolate, recover from faults, or at least, sustain essential parts of the guarded system. A whole new research seems to open in the direction of self-sustaining systems [29,39]. This research considers the requirements for adaptive systems antagonistic to the limiting requirements of perfect system design. Their motivation for future system design is to find a satisfying trade-off between the vision of a perfectly functioning environment and the real requirements of an adaptive but at times not-fully-operational runtime.

Acknowledgments This survey has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

1. Abbas N, Palankar M, Tambe S, Cook JE (2004) Infrastructure for making legacy systems self-managed. In: 2004 Workshop on Self-Managing Systems, Newport Beach, CA, USA, October 31–November 1
2. Adve S, Harris A, Hughes C, Jones D, Kravets R, Nahrstedt K, Sachs D, Sasanka R, Srinivasan J, Yuan W (2002) The Illinois GRACE Project: global resource adaptation through cooperation. In: Proceedings of the workshop on self-healing, adaptive, and self-managed systems (SHAMAN)
3. Akoglu A, Sreeramareddy A, Josiah J (2009) FPGA based distributed self healing architecture for reusable systems. Cluster Comput 12(3):269–284
4. Albrecht J, Oppenheimer D, Vahdat A, Patterson DA (2008) Design and implementation trade-offs for wide-area resource discovery. ACM Trans Internet Technol 8(4):1–44
5. Alonso J, Torres J, Moura Silva L, Griffith R, Kaiser G (2008) Towards self-adaptable monitoring framework for self-healing. Tech. Rep. TR-0150, Institute on Architectural issues: scalability, dependability, adaptability, CoreGRID-Network of Excellence
6. Alpern B, Schneider FB (1989) Verifying temporal properties without temporal logic. ACM Trans Program Lang Syst 11(1):147–167
7. Angskun T, Fagg GE, Bosilca G, Pjesivac-Grbovic J, Dongarra JJ (2006a) Scalable fault tolerant protocol for parallel runtime environments. In: 2006 Euro PVM/MPI
8. Angskun T, Fagg GE, Bosilca G, Pjesivac-Grbovic J, Dongarra JJ (2006b) Self-healing network for scalable fault tolerant runtime environments. In: Proceedings of 6th Austrian-Hungarian workshop on distributed and parallel systems. Springer, Innsbruck
9. Arora A, Gouda M (1993) Closure and convergence: a foundation of fault-tolerant computing. IEEE Trans Softw Eng 19(11):1015–1027
10. Baresi L, Guinea S (2007) Dynamo and self-healing BPEL compositions. In: 29th International conference on software engineering-companion, ICSE 2007 Companion, pp 69–70
11. Baresi L, Guinea S, Pasquale L (2007) Self-healing BPEL processes with Dynamo and the JBoss rule engine. In: ESSPE '07: International workshop on Engineering of software services for pervasive environments. ACM, New York, pp 11–20
12. Bigus JP, Schlosnagle DA, Pilgrim JR, Mills IWN, Diao Y (2002) Able: a toolkit for building multiagent autonomic systems. IBM Syst J 41(3):350–371
13. Blair GS, Coulson G, Andersen A, Blair L, Clarke M, Costa F, Duran-Limon H, Fitzpatrick T, Johnston L, Moreira R, Parlavantzas N, Saikoski K (2001) The design and implementation of Open ORB 2. IEEE Distributed Syst Online 2(6)

14. Blair GS, Coulson G, Blair L, Duran-Limon H, Grace P, Moreira R, Parlavantzas N (2002) Reflection, self-awareness and self-healing in OpenORB. In: WOSS '02: Proceedings of the first workshop on Self-healing systems. ACM, New York, pp 9–14
15. Broy M (1997) Requirements engineering for embedded systems. In: Proceedings of the first workshop formal design of safety critical embedded systems (FemSys)
16. Candea G, Cutler J, Fox A (2002) Improving availability with recursive micro-reboots: a soft-state system case study. In: Dependable systems and networks: performance and dependability symposium (DNS-PDS)
17. Cheng SW, Garlan D, Schmerl B, Steenkiste P, Hu N (2002) Software architecture-based adaptation for grid computing. In: HPDC '02: Proceedings of the 11th IEEE international symposium on high performance distributed computing. IEEE Computer Society, Washington, DC, p 389
18. Cheng SW, Garlan D, Schmerl BR, Sousa JP, Spitnagel B, Steenkiste P (2002) Using architectural style as a basis for system self-repair. In: WICSA 3: Proceedings of the IFIP 17th world computer congress-TC2 Stream/3rd IEEE/IFIP conference on software architecture. Kluwer, The Netherlands, pp 45–59
19. Cheng SW, Garlan D, Schmerl B (2006) Architecture-based self-adaptation in the presence of multiple objectives. In: SEAMS '06: Proceedings of the 2006 international workshop on self-adaptation and self-managing systems. ACM, New York, pp 2–8
20. Clarke EM, Grumberg O (1987) Avoiding the state explosion problem in temporal logic model checking. In: PODC Proceedings of the sixth annual ACM Symposium on Principles of distributed computing. ACM, New York, pp 294–303
21. Corsava S, Getov V (2003) Intelligent architecture for automatic resource allocation in computer clusters. In: IPDPS '03: Proceedings of the 17th international symposium on parallel and distributed processing. IEEE Computer Society, Washington, DC, p 201.1
22. Coulouris G, Dollimore J, Kindberg T (1994) Distributed systems: concepts and design. Addison-Wesley Longman Publishing Co., Inc., Boston
23. Dabrowski C, Mills K (2002) Understanding self-healing in service-discovery systems. In: WOSS '02: Proceedings of the first workshop on self-healing systems. ACM, New York, pp 15–20
24. Dashofy EM, van der Hoek A, Taylor RN (2002) Towards architecture-based self-healing systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems. ACM, New York, pp 21–26
25. Dijkstra EW (1974) Self-stabilizing systems in spite of distributed control. Commun ACM 17(11): 643–644
26. Dolev S, Schiller E (2004) Self-stabilizing group communication in directed networks. Acta Informatica 40(9):609–636
27. Ellison R, Fisher D, Linger R, Lipson H, Longstaff T, Mead N (1999) Survivability: protecting your critical systems. Internet Comput IEEE 3(6):55–63
28. Fuad MM, Deb D, Oudshoorn MJ (2006) Adding self-healing capabilities into legacy object oriented application. In: ICAS '06: Proceedings of the international conference on autonomic and autonomous systems. IEEE Computer Society, Washington, p 51
29. Gabriel R (2008) On sustaining self. In: Self-sustaining systems: first workshop, S3 2008 Potsdam, Germany, May 15–16, 2008 Proceedings. Springer, Berlin, pp 51–53
30. Ganek AG, Corbi TA (2003) The dawning of the autonomic computing era. IBM Syst J 42(1):5–18
31. Ghosh D, Sharman R, Raghav Rao H, Upadhyaya S (2007) Self-healing systems—survey and synthesis. Decis Support Syst 42(4):2164–2185
32. Ghosh S (2006) Distributed systems: an algorithmic approach. Chapman & Hall/CRC, Boca Raton
33. Glass M, Lukasiewycz M, Streichert T, Haubelt C, Teich J (2007) Reliability-aware system synthesis. Design, Automation and Test in Europe Conference & Exhibition, pp 1–6
34. Glass M, Lukasiewycz M, Reimann F, Haubelt C, Teich J (2008) Symbolic Reliability Analysis of Self-healing Networked Embedded Systems. In: SAFECOMP '08: Proceedings of the 27th international conference on computer safety, reliability, and security. Springer, Berlin, pp 139–152
35. Griffith R, Kaiser G (2005) Manipulating managed execution runtimes to support self-healing systems. SIGSOFT Softw Eng Notes 30(4):1–7
36. Halima RB, Drira K, Jmaiel M (2008) A QoS-oriented reconfigurable middleware for self-healing web services. In: ICWS '08: Proceedings of the 2008 IEEE international conference on web services. IEEE Computer Society, Washington, pp 104–111
37. Haydarlou A, Overeinder B, Brazier F (2005) A self-healing approach for object-oriented applications. In: Proceedings of the sixteenth international workshop on database and expert systems applications, pp 191–195

38. Herder JN, Bos H, Gras B, Homburg P, Tanenbaum AS (2006) MINIX 3: a highly reliable, self-repairing operating system. SIGOPS Oper Syst Rev 40(3):80–89
39. Hirschfeld R, Rose K (2008) Self-sustaining systems: first workshop, S3 2008 Potsdam, Germany, May 15–16, 2008. Revised Selected Papers. Springer, Berlin
40. Holderfield V, Huhns M (2003) A foundational analysis of software robustness using redundant agent collaboration. Lecture notes in computer science, pp 355–369
41. Hong M, Huang G, Tsai W (2005) Towards self-healing systems via dependable architecture and reflective middleware. In: Proceedings: 10th IEEE international workshop on object-oriented real-time dependable systems, WORDS 2005, 2–4 February 2005, Sedona, Arizona. IEEE Computer Society, Washington, DC, p 337
42. Huebscher MC, McCann JA (2008) A survey of autonomic computing—degrees, models, and applications. ACM Comput Surv 40(3):1–28
43. Huhns MN, Holderfield VT, Gutierrez RLZ (2003) Robust software via agent-based redundancy. In: AAMAS '03: Proceedings of the second international joint conference on autonomous agents and multiagent systems. ACM, New York, pp 1018–1019
44. IBM (2005) An architectural blueprint for autonomic computing. IBM
45. Jennings N (2000) On agent-based software engineering. Artif Intell 117(2):277–296
46. Kaelbling LP (1993) Learning in embedded systems. The MIT Press, Cambridge
47. Kant L, Chen W (2004) Alarm model specification and dynamic multi-layer self-healing mechanisms for commercial and ad-hoc wireless networks. In: 15th IEEE international symposium on personal, indoor and mobile radio communications, PIMRC 2004, vol 2, pp 959–963
48. Kant L, Chen W (2005) Service survivability in wireless networks via multi-layer self-healing. Wireless communications and networking conference, IEEE, vol 4, pp 2446–2452
49. Kephart J, Walsh W (2004) An artificial intelligence perspective on autonomic computing policies. In: Proceedings fifth IEEE international workshop on policies for distributed systems and networks, POLICY 2004, pp 3–12
50. Kephart JO, Chess DM (2003) The vision of autonomic computing. Comput IEEE Comput Soc Press 36(1):41–50
51. Koch D, Streichert T, Dittrich S, Strengert C, Haubelt C, Teich J (2006) An operating system infrastructure for fault-tolerant reconfigurable networks. In: Proceedings of the 19th international conference on architecture of computing systems (ARCS 2006), Frankfurt/Main, Germany. Springer, Frankfurt, pp 202–216
52. Kon F, Rom'an M, Liu P, Mao J, Yamane T, Magalha C, Campbell RH (2000) Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In: Middleware '00: IFIP/ACM international conference on distributed systems platforms. Springer, Secaucus, pp 121–143
53. Kopetz H (1997) Real-time systems: design principles for distributed embedded applications. Springer, Berlin
54. Ledoux T (1999) OpenCorba: a reflektive open broker. In: Reflection '99: proceedings of the second international conference on meta-level architectures and reflection. Springer, London, pp 197–214
55. Linger R, Mead N, Lipson H (1998) Requirements definition for survivable network systems. In: Proceedings of the 1998 international conference on requirements engineering (ICRE'98), pp 6–10
56. Maes P (1987) Concepts and experiments in computational reflection. ACM Sigplan Notices 22(12):147–155
57. Merideth M (2003) Enhancing survivability with proactive fault-containment. In: DSN Student Forum, Citeseer
58. Modafferi S, Conforti E (2006) Methods for enabling recovery actions in Ws-BPEL. Lect Notes in Comput Sci 4275:219
59. Modafferi S, Mussi E, Pernici B (2006) SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In: MW4SOC '06: Proceedings of the 1st workshop on middleware for service oriented computing (MW4SOC 2006). ACM, New York, pp 48–53
60. Moo-Mena F, Garcilazo-Ortiz J, Basto-D'ıaz L, Curi-Quintal F, Alonzo-Canul F (2008) Defining a self-healing QoS-based infrastructure for web services applications. In: CSEWORKSHOPS '08: Proceedings of the 2008 11th IEEE international conference on computational science and engineering-workshops. IEEE Computer Society, Washington, DC, pp 215–220
61. Moser O, Rosenberg F, Dustdar S (2008) Non-intrusive monitoring and service adaptation for Ws-BPEL. In: WWW '08: Proceeding of the 17th international conference on World Wide Web. ACM, New York, pp 815–824

62. Norman DA, Ortony A, Russell DM (2003) Affect and machine design: lessons for the development of autonomous machines. IBM Syst J 42:38–44
63. Parashar M, Hariri S (2005) Autonomic computing: an overview. In: Unconventional programming paradigms. Springer, Berlin, pp 247–259
64. Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058
65. Paul H (2001) Autonomic computing: IBM's Perspective on the State of Information Technology. International Business Machines Corporation, http://www.research.ibm.com/autonomic/
66. Picard RW (1997) Affective computing. The MIT Press, Cambridge
67. Pierce W (1965) Failure-tolerant computer design. Academic Press, New York
68. Razzaque MA, Dobson S, Nixon P (2007) Cross-layer architectures for autonomic communications. J Netw Syst Manage 15(1):13–27
69. Rhea S, Geels D, Roscoe T, Kubiatowicz J (2004) Handling churn in a DHT. In: ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference, USENIX Association, Berkeley, CA, USA, pp 10–10
70. Rilling L (2006) Vigne: towards a self-healing grid operating system. In: Proceedings of Euro-Par 2006. Lecture notes in computer science, vol 4128. Springer, Dresden, pp 437–447
71. Salehie M, Tahvildari L (2005) Autonomic computing: emerging trends and open problems. SIGSOFT Softw Eng Notes 30(4):1–7
72. Salehie M, Tahvildari L (2009) Self-adaptive software: landscape and research challenges. ACM Trans Auton Adapt Syst 4(2):1–42
73. Shapiro MW (2005) Self-healing in modern operating systems. Queue 2(9):66–75
74. Sloman A, Croucher M (1981) Why robots will have emotions. In: Proceedings IJCAI
75. Sterritt R (2005) Autonomic computing. Innov Syst Softw Eng 1(1):79–88
76. Subramanian S, Thiran P, Narendra NC, Mostefaoui GK, Maamar Z (2008) On the enhancement of BPEL engines for self-healing composite web services. In: SAINT '08: Proceedings of the 2008 international symposium on applications and the internet. IEEE Computer Society, Washington, DC, pp 33–39
77. Tanenbaum A, Herder J, Bos H (2006) Can we make operating systems reliable and secure? Computer 39(5):44–51
78. Tarr P, Ossher H, Harrison W, Sutton SM Jr (1999) N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st international conference on software engineering. ACM, New York, pp 107–119
79. Tesauro G, Chess DM, Walsh WE, Das R, Segal A, Whalley I, Kephart JO, White SR (2004) A multi-agent systems approach to autonomic computing. In: AAMAS '04: Proceedings of the third international joint conference on autonomous agents and multiagent systems. IEEE Computer Society, Washington, pp 464–471
80. Venishetti SK, Akoglu A, Kalra R (2007) Hierarchical built-in self-testing and fpga based healing methodology for system-on-a-chip. In: AHS '07: Proceedings of the second NASA/ESA conference on adaptive hardware and systems. IEEE Computer Society, Washington, DC, pp 717–724
81. White S, Hanson J, Whalley I, Chess D, Kephart J (2004) An architectural approach to autonomic computing. In: Proceedings international conference on autonomic computing, pp 2–9
82. Winter R, Schiller J, Nikaein N, Bonnet C (2006) Crosstalk: cross-layer decision support based on global knowledge. Commun Mag IEEE 44(1):93–99
83. Yuan W, Senior Nahrstedt K, Adve SV, Jones DL, Kravets RH (2006) GRACE-1: cross-layer adaptation for multimedia quality and battery energy. IEEE Trans Mobile Comput 5(7):799–815