

Self-healing systems — survey and synthesis

Debanjan Ghosh ^a, Raj Sharman ^{b,*}, H. Raghav Rao ^{a,b}, Shambhu Upadhyaya ^a

^a Department of CSE, SUNY, Buffalo, United States

^b Department of Management Science and Systems, School of Management, SUNY, Buffalo, NY 14260, United States

Received 25 April 2005; received in revised form 9 March 2006; accepted 7 June 2006

Available online 17 August 2006

Abstract

As modern software-based systems and applications gain in versatility and functionality, the ability to manage inconsistent resources and service disparate user requirements becomes increasingly imperative. Furthermore, as systems increase in complexity, rectification of system faults and recovery from malicious attacks become more difficult, labor-intensive, expensive, and error-prone. These factors have actuated research dealing with the concept of self-healing systems. Self-healing systems attempt to “heal” themselves in the sense of recovering from faults and regaining normative performance levels independently the concept derives from the manner in which a biological system heals a wound. Such systems employ models, whether external or internal, to monitor system behavior and use inputs obtaining therefore to adapt themselves to the run-time environment. Researchers have approached this concept from several different angles this paper surveys research in this field and proposes a strategy of synthesis and classification. © 2006 Elsevier B.V. All rights reserved.

Keywords: Software engineering designing; Software architecture; Fault tolerance; Self-healing; Decision support systems; Distributed systems; Adaptive systems; Survivable systems

1. Introduction

An increasingly significant requisite for software-based systems is the ability to handle resource variability, ever-changing user needs and system faults. Certain standard programming practices, such as capacitating extensive error handling capabilities through exception catching schemes, do contribute towards rendering systems fault-tolerant or self-adaptive,¹ however, these methods are tightly coupled with software codes and are highly application-specific. Designs that enable software sys-

tems to heal themselves of system faults and to survive malicious attacks would radically improve the reliability and consistency of technology in the field. The endeavor to secure these benefits has originated the concept of self-healing systems. Self-healing can be defined as the property that enables a system to perceive that it is not operating correctly and, without (or with) human intervention, make the necessary adjustments to restore itself to normalcy. Healing systems that require human intervention or intervention of an agent external to the system can be categorized as assisted-healing systems. The key focus or contrasting idea as compared to dependable²

* Corresponding author.

E-mail address: rsharman@buffalo.edu (R. Sharman).

¹ A system in which the programs will evolve to optimize an extrinsic fitness function imposed on their environment. Typically these programs will only be estimated for their fitness, and those with the best health will be chosen to survive and propagate. For more on adaptive systems refer to [22].

² According to Avizienis et al. [3] dependability is an integrating concept that encompasses availability (readiness for correct service), reliability (continuity of correct service), safety (absence of catastrophic consequences on the users and the environment), integrity (absence of improper systems alterations) and maintainability (ability to undergo modifications and repairs).

systems is that 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. A recent instance where this definition has been deemed normative is in the work of Lu et al. [36], which deals with electronic devices where a feedback control algorithm for dynamic frequency/voltage scaling is used for portable multimedia applications.

Some scholars treat self-healing systems³ as an independent area of study while others view them as a subclass of traditional fault tolerant computing⁴ systems. Fault tolerant computing can be defined as the ability of a system to respond seamlessly [with minimal disruption] to an unexpected hardware or software failure. The term “Fault-tolerant” indicates that a computer system or component thereof has been designed such that in the event a component fails, a backup component or procedure can immediately take its place with no loss of service. Many fault-tolerant computer systems ‘mirror’ all their operations — that is, every operation is performed on two or more duplicate systems, so that if one system fails, another can take over. ‘Survivable systems’⁵ also fall under the rubric of fault tolerant computing systems; these are systems that are able to continue discharging their normal operation despite the presence of malicious or arbitrary faults. However, neither of these two systems comprehends recovery oriented computing, which is a key aspect of self-healing systems. Healing systems are more concerned about post-fault or post-attack states; they are focused to a greater degree upon the healing process and upon bringing the system back to “normal”. Most current research concentrates on the offline healing of components, being fault-diagnosis, recovery and re-induction of repaired element into the system.

Self-healing systems can support decision making in a large way for managerial and organizational situations. As these systems are typically recovery oriented, they can take their own decision on how to come back to normalcy from a broken state. Many of the decision-support systems (DSS) offer passive forms of decision support, where the decision-making process depends upon the user’s initiative. But self-healing systems can

offer an active form of decision support, without human intervention they can detect the fault and recover from the fault. Also, with intelligent architectural models, a self-healing system can select the proper repair plan to deploy the broken component (or client), if there is more than one component that needs to be healed, they can prioritize a fault component over the others, etc. Such active involvement is especially needed in complex decision-making environments.

This paper peruses literature in the area of self-healing systems and attempts a classification based on similarities or relationships. Section 2 outlines topical issues in the field of self-healing and provides a glimpse of the scope of this subject. Section 3 is devoted to an in-depth analysis of research in self-healing, with particular emphasis on the approach adopted. Section 4 details the applications of self-healing systems. Section 5 comprises a discussion on conclusion and promising new directions for research in this field.

2. Self healing strategies

2.1. Self healing and biology

The term “self-healing” is drawn from the natural (biological) paradigm. Through billions of years nature has created extraordinary mechanisms to perform robustness and self-healing. The “biological program” is essentially coded in DNA. Various biological features have inspired development of computational models to create problem-solving techniques.

As biological systems exhibit adaptation, healing and robustness in the face of continuing, changing environmental behavior, this paradigm has attracted numerous computer science researchers. Before discussing the approaches mimicking biology in self-healing or adaptive systems, we will try to examine how immune systems of biology have inspired researchers to model computer systems. Artificial immunology is an important field, which has drawn noteworthy attention lately. Forrest et al. [15] point out that the role of natural immune systems for protecting animals from foreign pathogens (for example viruses, bacteria, toxins) is very much analogous to the self-healing systems of computers.

Properties like multilayered protection, distributed detections, matching strategies, selection, etc. of natural immune systems are remarkable and highly compelling, and at the same time may be tracked to improve the designs of self-healing systems.

Table 1 below addresses recent research in the field of artificial immune systems (AIS) and self-healing systems, inspired by the biology paradigm.

³ Interest in self-healing systems has recently increased enormously. A recent workshop on Self-Healing Systems (WOSS’02, November 18–19, 2002 Charleston, SC, USA) exposed a diverse range of researcher perspectives on self-healing systems.

⁴ Nelson [44] pointed out that the goal of fault-tolerant systems is to improve dependability by enabling a system to perform its intended function in the presence of a given number of faults.

⁵ Survivable systems may need to be both fault-tolerant and secure, for example, detect a malicious fault, and remove the same from the system and attempt to secure the system [37].

Table 1
Artificial immune systems (AIS) approaches

Models	Descriptions	References
Immune network theory	B-cells population is created by a raw training set of data that is used for generating the network.	[11]
	A different set of training data is used for antigen training items.	[11]
	Randomly selected antigens are presented in the B-cell network for binding. For successful binding, B-cells can be cloned and mutable.	[11]
Negative selection mechanism	High probability that some T-cells will detect self-peptides.	[15]
	Non-functional T-cells are sent to the thymus for a maturing process that goes through various stages (like a censoring process).	[15]
	T-cells failing to bind with self-proteins are allowed to leave the thymus and be part of the immune system, a process called negative selection.	[15]
Barrier defense analogy	The cell membrane acts as a barrier separating the external and internal environments of a cell.	[28]
	This is similar to an exterior router, firewalls or any IDS.	[28]
Barrier transmission analogy	Cells communicate via electrical signals through open membrane channels or holes (network firewalls).	[28]
	Gap-junctions of cells allow the passage of small molecules among cells (VPN's).	[28]
Internal organization analogy	Through porous perinuclear space, cytoplasm and nucleus communicates.	[28]
	Network DMZ (buffer zones).	[28]
Internet routing analogy	Endocytosis and exocytosis facilitate the communication and messaging between organelles.	[28]
	Similar to IPv6 where security and privacy values are added in packet header fields.	[28]
Self-assembly of components	Self-assembling complex structures.	[41]
	Interaction of similar programmed modules.	[41]
Cell division and its application	Transition in system states mimicking cell division.	[18]
	Environmental awareness of components.	[18]
	Signaling other components of the system.	[18]

2.2. Self and non-self states of a system

In this section, we first detail the differences between “healthy” and “unhealthy” systems. This section also includes a discussion on the typical characteristics of self-healing systems.

Before a system can make adjustments necessary to restore itself to normalcy, it needs to be cognizant of what constitutes “normalcy” in its operations and be attentive to deviations there from. The specification to which a system has been built is usually not fully known to those who maintain it; this hinders both maintenance and recovery of the system. Furthermore, as Shaw [50] has pointed out, the standards of what constitutes acceptable behavior in a system may vary both with time and from one user to another. Key properties of a system, such as its modularity, fidelity, performance in different environments, and even its proposed utility are often incompletely defined. The designer’s perception of both the properties of the system and the requirements of the user is usually incomplete; as a result, systems are typically vulnerable and error-prone even when newly commissioned. Therefore, it is important to define the criteria for a “healthy” system and identify thresholds indicative of the need to initiate a healing process. These criteria are often dynamic, and change as the user revises the use to which the system is put, as also with variation in the conditions under which it operates. Given that our

knowledge of the specifications of a system is undependable, we may need to revise our conclusions about the performance of a system as further knowledge of the specifications of that system becomes available. Thus, the distinction between “healthy” and “broken” is often indistinct and fuzzy, and often there is a gradual transition between these two states; a system often does not break down recognizably but deteriorates over time. Thus we can say there is a fuzzy zone, a degraded state, separating acceptable and unacceptable behavior of a system, which again depends on user preferences and environmental changes. Fig. 1 indicates that the system usually transitions from a normal state to a degraded state as attacks become successful or faults begin to take effect.

In summary, it is difficult to draw a discrete difference between “healthy” and “unhealthy” states of a system as the transition in between the two states is not abrupt. What generally obtains is a gradual transition from one state to another, this deterioration forming a “fuzzy zone” of behavior. It is important that a self-healing system discern this progressive decline, as also identify a definite threshold to initiate rectification at, and thereby maintain its health. Detection of self and non-self both at the component level and at the system level is a hard task.

An important question with regards to healing is whether the healing is achieved with or without human intervention. Raz et al. [46] have pointed out that human

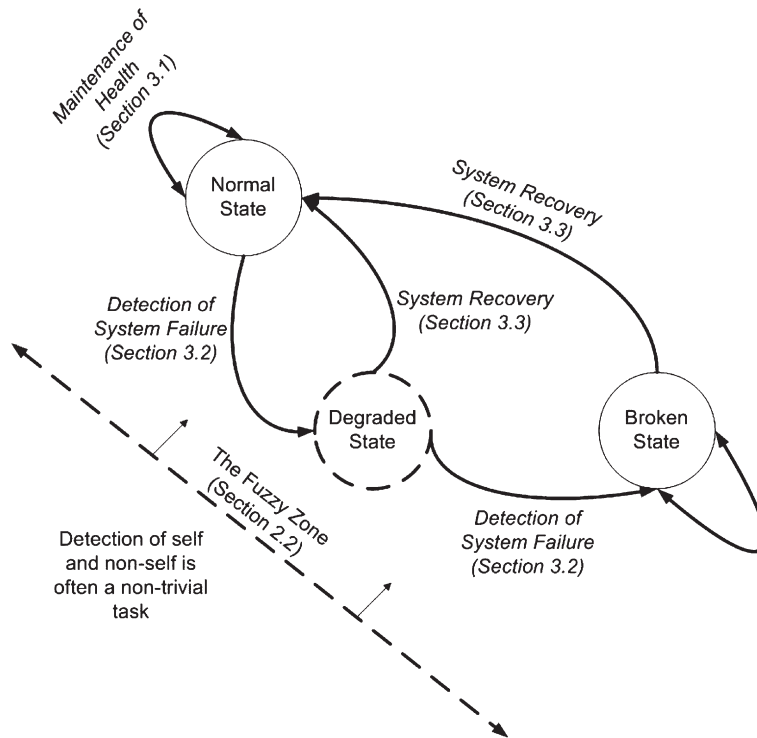


Fig. 1. State diagram of health/broken states of a system.

intervention is needed in any template design mechanism to make subjective selections of the proper number of variables, training set data, and so on. Some researchers have commented that self-healing systems should be gradually automated, lowering human intervention. In certain situations, users may profit from adapting their own behavior in a manner designed to help systems discharge their functionalities at a time of failure. IBM's autonomic computing initiative focuses on systems becoming self-configuring (dynamic adaptation to changing environments), self-healing (the discovery and diagnosis of disruption and the reaction to that), self-optimizing (the monitoring and modulation of resources automatically) and self-protecting (wherein systems anticipate, detect and protect themselves from attack) [16]. Some researchers are working towards autonomous healing components; partial human interference in a system is also a popular design choice.

Critical issues in self-healing systems typically include detection of self/non-self mechanism, maintenance of system health, recovery processes to return the state from an unhealthy state to a health one. Self-healing components or systems typically have the following characteristics: (a) perform the productive operations of the system, (b) coordinate the activities of the different

agents, (c) control and audit performance, (d) adapt⁶ to external [environmental] and internal [organizational] changes and (e) have policies to determine the overall purpose of the system. System detection of failure: any secured system should detect failure or the presence of a malicious agent in a timely manner. It must be smart enough to gauge the degree of malfunction in the system, and assess whether the system actually needs the intervention of a recovery program.

2.2.1. Maintenance of system health

Different strategies may be employed to check or maintain the normal functionality of the system. Further the system may check for malfunction periodically or constantly. Moreover, the strategy employed has to be smart enough to understand the fuzzy separation zone between "healthy" and "unhealthy" states. One approach

⁶ It should be pointed out that adaptive systems and self-healing systems are often considered similar. As defined by Laddaga [32], "Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible". Typically, adaptive systems are contingent upon varying environments. They can reason about the environment and adapt to changing situations.

is to make sure that the system maintains a redundancy policy. Ensuring adequate component redundancy or replication is an important consideration while designing a healing policy. Such strategies make available a number of component implementations, providing the same services, at run-time. In the event that one component instance fails, malfunctions, or is attacked, it can be replaced with another component version that offers equivalent functionalities. For example, in a typical biological system, during the healing of a wound, organisms produce a surfeit of cells dedicated to healing, only a fraction of which are required to heal the wound; the redundant cells are however always available to take the place of active healing cells that may be damaged.

2.2.2. System recovery from an unhealthy state and return to a healthy state

This is an open issue that includes how to isolate the faulty component, fix it and re-induct it into the system without disruption of other applications. Unaffected modules should maintain their functionalities even in the presence of fault in other components. The analogy in biological systems would be the expectation that the body recovers from a wound, leaving the other functions unaltered.

2.2.3. Self-healing systems may follow various architectural schemes

The healing agent may either be a component of the overall system or be an independent agent. Normal self-healing systems need high dependability, robustness and a high degree of adaptability. Grishikashvili [20] pointed out that these characteristics of a self-healing system usually match well with software agent architectures that are autonomous (those that can operate without human intervention); can interact with other agents, such as in a multi-agent design style; and can perceive their environment.

Various self-healing strategies use different policies to probe into and assess the state of the system. The type of information gathered to make the decision for adaptation is an important parameter. The next section is devoted to providing a more detailed classification and review of the literature in the area of self-healing systems.

3. Survey of self-healing systems

This section provides a more detailed description of the various approaches to self-healing system that has been adopted by researchers. It also includes a classification and the review of the literature as well. The self-healing process usually consists of (a) maintenance of the system health, (b) discovery of non-self and (b) system recovery process as shown in Fig. 2. The literature in this area can also be broadly classified into these categories. This section is therefore divided into three subsections as shown in Fig. 2.

3.1. System maintenance of health

The system should check for faults periodically to continue monitoring its health. Additionally, after a system recovers from malfunctioning, different tactics may be applied to maintain the normal functionality of the system (Fig. 3).

There are various strategies that have been used to maintain the system health such as maintaining redundancy of components, probing into the system and assessing the state of the system, building diversity, etc. as depicted in Fig. 4. The research in this area as such can be classified as falling into one of these categories and therefore this section is partitioned accordingly into subsections as indicated in Fig. 4.

3.1.1. Maintaining redundancy

Replicating components to maintain redundancy is always a popular choice in maintaining system health.

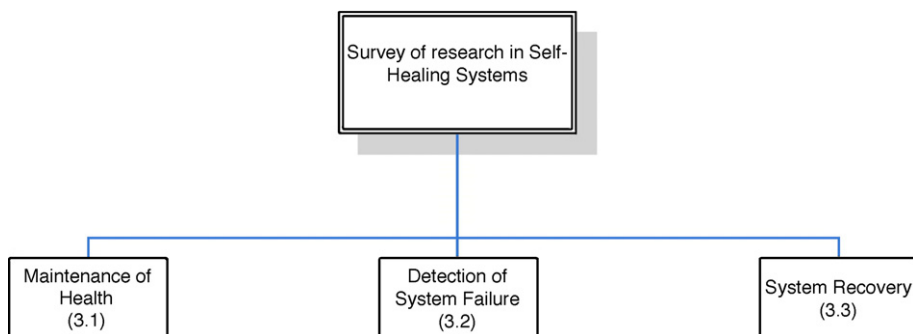


Fig. 2. Different paradigms of a self-healing system.

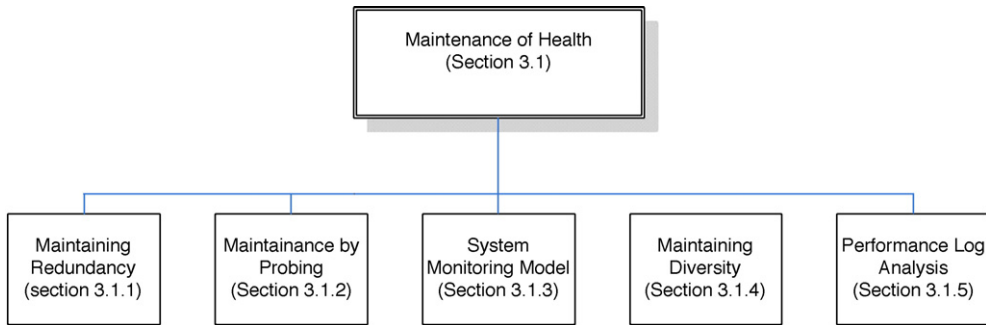


Fig. 3. Maintenance of health.

However, there are many different redundancy strategies that researchers in the area of self-healing have pursued. Fig. 4 shows the classification of the literature into three different areas. The remainder of this section is devoted to providing an insight into each of these categories.

Nagpal et al. [41] present ‘Self-Assembly of components’, a programming methodology for self-assembly of complex structures using techniques inspired by biology. Specifically, this involves the folding in of epithelial cell sheets, which consist of a single layer of thousands of randomly and densely distributed, but identical and autonomous, locally interacting agents, using a language called Origami Shape Language (OSL). This is a scale-independent program that can generate many different related shapes without modifying the agent programs. George et al. [18] describe the actions of biological cells and demonstrate the ability of systems to survive failures based on the cell division model. The cell division model in the human system is a cyclical model. While a human system is in healthy state, variant concentrations of chemicals and other agents produced by cell action cause gene actions which in turn result in cell actions; the cycle that thus obtains precludes the absolute failure of the system. This model starts with cells in their initial configuration and all the cells seem to follow transition rules like a finite state machine. This method, when applied to a self-

healing system, evidences localization, adaptation, adequate redundancy and the unique distinction of awareness towards the environment.

This paradigm is illustrated with a real world scenario in Distributed Wireless File Service (DWFS), an application layer peer-to-peer file sharing service. When a new file is published, the publishing server broadcasts two messages: one the ‘inhibit’ message and the other the ‘replicate’ message. Nodes, which serve to replicate what they receive, relay both these types of messages.

For a multi-agent system, Huhns et al. [25] propose a different type of redundancy policy. This maintains two types of decision-making algorithms (for pre-processing and post-processing decision making). Redundant agents are used to keep the system running with different algorithms to provide a robust software solution.

3.1.2. System maintaining by probing

Probing is a mechanism that is commonly used to get information from the system in order to monitor its health. Fig. 5 below depicts different strategies using probes to get updated information about the system for maintaining system health. The root node portrays the probing concept for maintaining system health as a whole. Leaf nodes represent different system probing procedures.

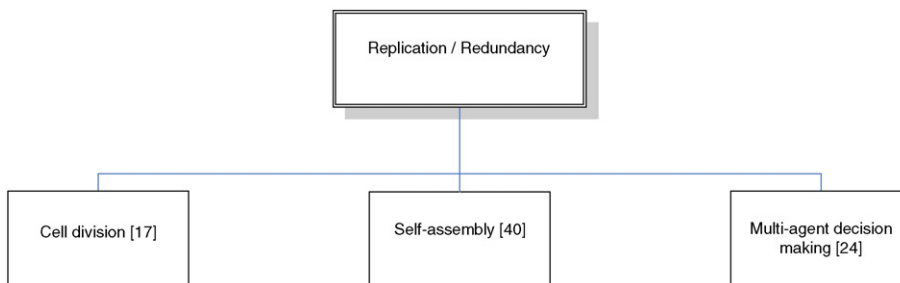


Fig. 4. Redundancy policies for system health maintenance.

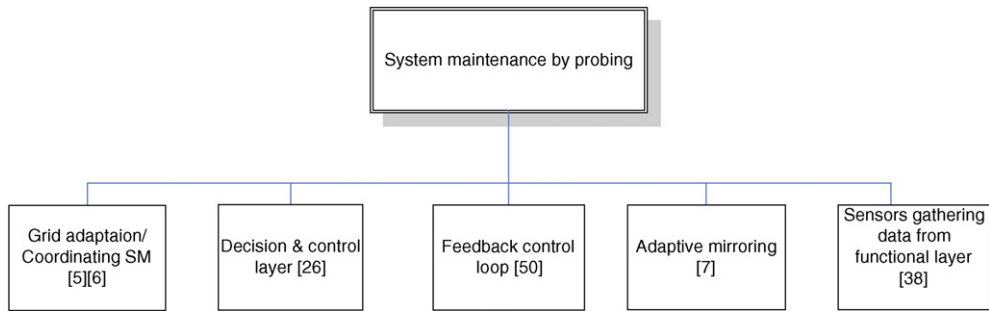


Fig. 5. System health maintaining by probing.

Cheng et al. [6] have proposed a novel software architecture model and monitoring infrastructure, which can be maintained at runtime and can be used as a basis for system configuration and adaptation. It consists of several layers. The task layer decides which application should be executed, based on pre-defined policies, and also sets the performance objectives and resource constraints. The system-monitoring layer incorporates the lowest level of abstractions or “probes”, which monitor the system and enunciate observations via a “probe bus”. At the second level, a set of “gauges” are used to report information via a bus called a gauge reporting bus. A third level comprises gauge consumers which consume information broadcast by the gauges. These different layers would coordinate and combine multiple self-manageable components to maintain consistent sensing and information acquisition, and to ensure coherent decisions.

A similar architecture, employing “probes” and “gauges”, has been utilized by Kaiser et al. [27] to retrofit legacy systems with self-healing, self-management and adaptation capabilities. That schema features a “decision and control” layer, which receives information from the gauge-reporting bus, and optimizes the deployment of gauges or probes accordingly. This layer also reconfigures the system appropriately by introducing new modules or by changing system parameters as required.

Valetto et al. [51] have proposed the usage of a feedback control loop in a targeted system for software adaptation. Data is instrumented with probes, which report raw data into the probe bus. The probe here is like an independent but attached sensor. This raw data is then interpreted by gauges, which map the data into various system models with the help of a gauge bus. Gauges operate within a framework consisting of two main components, Event Packager and Event Distiller. The Event Packager transforms raw data collected from probes into smart event-compatible event streams. The

Event Distiller collects the multifaceted event patterns from the probe’s data set and reflects the system state specified by the events. Combs et al. [8] propose an adaptive mirroring strategy for external agents. This approach is based on utilization of probes to intercept a system workflow and circumnavigate data and control through a different path. These probes move data to and from an agent-based service channel that mirrors the target system. This mirrored space can adaptively shape its inner workflow; reliable data links among many points within the target application can thereby take place. In the DMonA architecture proposed by Michiels et al. [39], two types of sensors are used to gather information from the functional layer. Sensors that collect data about the internal states are called State Sensors (SS) and those that collect information about the messages flowing through the system are called Analysis Sensors (AS). The information gathered is thus processed by different monitoring strategies.

3.1.3. System monitoring architecture model

Research in the area of self-healing systems includes literature dealing with architecture models. The literature in this area can be broadly classified into three different categories based on the type of functionality addressed by the literature. The three categories include research: (a) representing the system using Architectural Description Language (ADL), (b) representing regularities in a system, (c) providing a relational model of the system. Garlan et al. [17] have created an architectural model which can be represented by plotting a graph of the interacting mechanism with the help of Acme, an Architecture Description Language (ADL). ADLs such as Acme, xADL, Darwin, etc. are being utilized to represent system architectures and facilitate the adaptation and reconfiguration of system components which are a necessary part of a healing system. This kind of usage is general and open-ended and utilizes a set of components, con-

nectors, and interface re-usage; if required, new repair plans can be added.

3.1.3.1. ADL-based approach. This section is devoted to describing the different papers that focus on architecture description languages. Georgiadis et al. [19] discuss an implementation dependent upon Darwin ADL and Jackson's alloy language in conjunction with a monitoring tool to view the runtime structure of a distributed system. This retrieves the configuration view from components and maintains the same when update events are broadcasted. Aldrich et al. [2] suggest that the amalgamation of ArchJava language with Java programming will make it simpler for system implementers to determine the self-healing properties of their code. They hope to design a ubiquitous architecture where components must typically self-heal in response to changes. They cite the example of a gardening service,⁷ one of several such services in the PlantCare⁸ System at Intel Research. Here a gardener periodically executes a cycle of codes (to water plants or check on their humidity level) to ensure the good health of each plant. Another approach by Dashofy et al. [12] delineates the architecture of the software system as represented by xADL 2.0, an extensible XML-based ADL.

Dabrowski et al. [10] have proposed an architecture-based adaptation in a service discovery system. Discovery protocols in general help software components find each other over a network and ascertain if revealed components match their prerequisites. The authors construct an architectural model of each discovery protocol, define appropriate metrics for comparing the behavior of each model, and contrast the results from executing similar scenarios against each model. This analysis took part in different architecture models facilitated by different components such as Service Manager (SM), Service User (SU) and Service Cache Manager (SCM). This system employs two mechanisms consistency-maintenance: Polling and Notification. In polling, a SU sends queries to get up-to-date information about a previously discovered Service Description (SD). In the notification scheme, after an update occurs, a SM sends out a notification that the SD has changed. As we discuss different approaches to system architecture, we

⁷ A gardening service [33] examines sensor data, consults species-specific care instructions in a plant encyclopedia service, and posts watering tasks to the task server.

⁸ A. Lamarca et al. [33] stated that the core architecture of Plant Care consists of wireless sensors for environmental sensing, a robot for actuation, and loosely coupled component-based software to integrate the hardware and provide application logic.

Table 2
ADL-based approach

ADL approach	Literature
Grid adaptation	[6]
Darwin's ADL and Jackson's Alloy for runtime view	[19]
ArchJava with Java	[2]
xADL 2.0	[12]
Rapide for service discovery	[10]
PitM	[45]

focus on factors like dependability, dynamicity, adaptability, etc. and how they are provided for in various architectures.

Rakic et al. [45] presented the design of a specific architectural style called PitM for supporting self-healing systems. They have identified some characteristics of the architectural style, such as external structure, topology rules, behavior, interaction and data flow. The PitM structure supports the following basic elements.

- 1) Components — maintain states and perform application specific computation.
- 2) Explicit connectors — mediate all components interactions.

Table 2 provides an overview of the papers in this area.

3.1.3.2. Component relation and regularities. Lemos et al. [35] have based their strategy on the isolation of faulty components and the reconfiguration of the system. Their system architecture utilizes multiple layers for doing each of the following:

- 1) Computation between the components of the system.
- 2) Coordination between interactions of components.
- 3) Configuration — which decides when and how components and connectors link up.

A system with a wide range of probable arrangements for self-healing should possess suitable regularities as pointed out by Minsky [40]. This can be satisfied by various possible configurations. Distributed systems often possess components built on different software and hardware, and Minsky [40] pointed out the desirability of imposing some artificial laws over such a system to induce desired regularities among the components thereof. A scalable control mechanism called Law Governing Interaction (LGI) is used for formulating the artificial laws. LGI is a message exchange mechanism between distributed components of a system.

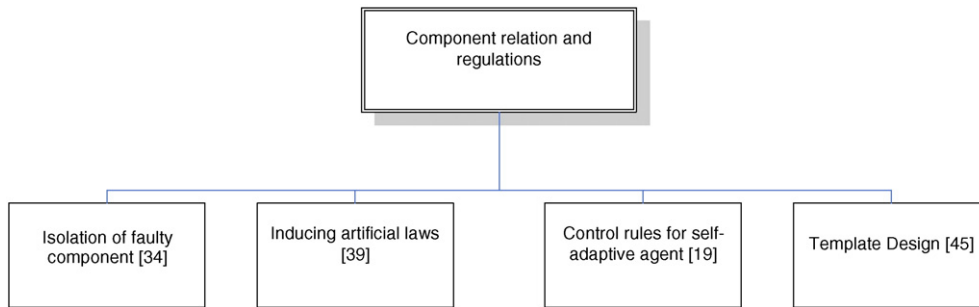


Fig. 6. Component relation and regularities.

The function of an LGI law is to regulate the exchange of messages in a community. A reference monitor can mediate this architecture of message exchange, in which case the scheme is denoted to centralized enforcement interaction. A remarkable feature of the LGI law is its localized nature; it can be defined over events occurring over only some members of a larger community. This Local Nature of LGI has been used in the Decentralization of Law-Enforcement scheme. Grishikashvili [20] pointed out that appropriate control laws governing communication must exist for any system to remain functional in a changing environment. Similarly, self-adaptive software agents can also be activated under the control laws in two ways: either the design of activities can match the conditions or actual activities can be identified by their system type and be assigned as appropriate element of the model to meet described conditions.

Raz et al. [46] have proposed a template design mechanism to lower the requirements of human attention, especially in cases where dynamic data feeding into systems from online data sources are not dependable. Examples of data sources are stock quotes, news reports and so on. This mechanism concentrates on enabling the automatic detection of semantic anomalies [47], that is, of cases in which the data feed are responsible for delivering results, but the results are conflicting, incorrect or unreasonable. This closed-loop mechanism prompts the user to decide upon whether the form of adaptive invariants is useful or not. These adaptive invariants are used as proxies for specifications that may serve as a replica of normal behavior to support semantic anomaly detection and also the repair mechanism. In this system, human intervention is desired as feedback regarding true positives and false negatives. This may be an iterative process with different variables, like a training set, attributes, etc. Fig. 6 provides a summary of the research in this area.

3.1.4. Diversity in system

Diversity is an important foundation of robustness in biological organisms. In computer systems by contrast, a lack of diversity is evident; this homogeneity renders a system vulnerable as most systems share the same architecture or configuration,⁹ an intruder or attacker is able to replicate a successful attack on one system in various others.

Sharman et al. [49] have proposed a novel paradigm for securing functionality-based systems. Healing is not performed in a single system, but on the functionality. Diversity is engendered by designing a single function (which is to be secured) which will run on different systems, and by then comparing the results. As the systems are designed independently, it becomes difficult for any intruder to break through all of them. Similarly, Forrest et al. [14] have proposed a diverse design model where the main strategy is of avoiding unnecessary consistency in software, thus making intrusion much more difficult to replicate. The guidelines are concerned mainly with adding/deleting nonfunctional codes, reordering the basic blocks of compiled codes in random order, reordering the memory layout, etc.

P. Inverardi et al. [26] have pointed out that identifying critical parts in the source code is important for providing different alternatives, such as offering various display capabilities in correspondence with output operations. Fig. 7 depicts different strategies using system diversity for maintaining system health. The root node of the tree structure used here illustrates the concept as a whole, while the leaf nodes represent different strategies.

3.1.5. Performance log analysis

Knop et al. [29] propose a scheme for the analysis of data on performance measures of the system, which can

⁹ As a model we can cite the example of IE web browser, which accounts for almost 90% of the browser market. An attacker can exploit this; a successful attack on IE can reproduce attacks on almost 90% of the whole world browser domain!

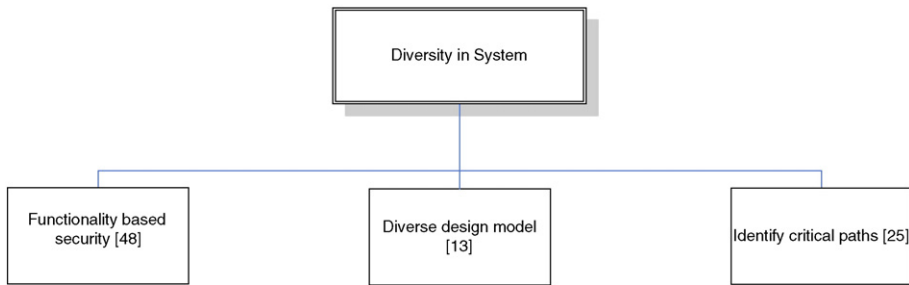


Fig. 7. Diversity in system.

be used for the diagnosis of faults, the detection of intrusion and the facilitation of the healing process. They build a library called “Watchtower”, based on Windows NT/2000, which monitors the command lines, the console and any streaming operations. This library gathers performance counters using Microsoft’s performance data helper (PDH) APIs. Event Log Analysis, which uses various standard time-series techniques for predicting rare events, can also be used to predict target events [48] across a computer network.

Hong et al. [21] propose a Finite Automata scheme to describe the aging and rejuvenation states of software. This model contains elements (converters) that detect software aging by monitoring periodically for typical symptoms. A supervisor element maintains the level of free memory in the system. It creates commands depending upon the past and present events generated by the converters to create a closed loop for the real process.

Rejuvenation commands are generated at two levels here; Level-1 rejuvenation is a service level upgrade which kills and restarts only some specific applications, while Level-2 rejuvenation is a system level upgrade which kills all services and restarts the whole system.

Fig. 8 depicts different strategies using performance log analysis for maintaining system health. The root node of the tree structure used here illustrates the concept as a whole, while the leaf nodes represent different strategies.

Table 3 below gives a summarized view of the different strategies to maintain system health.

In the above subsection we surveyed various literatures that focus on policies of maintaining the health of a system. In the method described by Kaiser et al. [27] this system continuously self-evaluates its performance and fine tunes itself in runtime to maintain system health. There are no restrictions on the system design in inserting application specific solutions into the main code. So in runtime, the system can continuously improve its performance and maintain good system health, without applying any offline recovery mechanism for repair.

Similarly, the DMonA [39] architecture can recognize I/O access patterns and can adapt to caching strategies accordingly in run time. The basic monitors and sensors are reusable. This architecture can gracefully handle loads of peak requests while maintaining system health.

The design of evaluating rejuvenation policies [21] is based on selection of system parameters, than selection of best applicable policy. This can be regarded as a weakness in the design.

Though maintaining redundancy in various layers of system for detection of failures and recovery from attacks or faults may be critical, sometimes avoiding unnecessary redundancy by maintaining diversity in functionality [14] is also important. But diversity techniques might disrupt

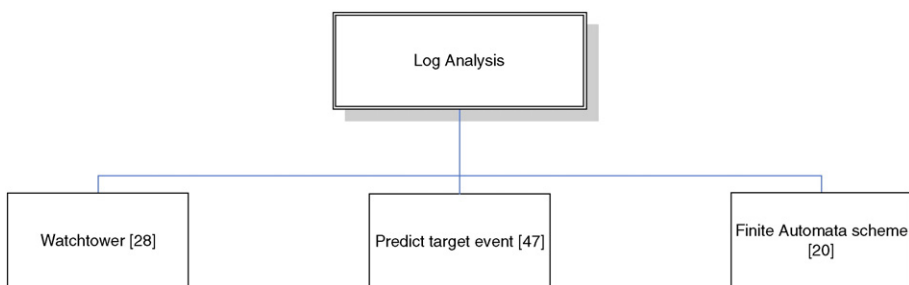


Fig. 8. Performance log analysis.

Table 3
System health maintenance policies

Models	Descriptions	References	
Redundancy approach	Cell division	[18]	
	Self-assembly of components	[41]	
	Multi-agent decision making	[25]	
System maintaining by probing	Grid adaptation/coordinating multiple SM	[6,7]	
	Decision and control layer	[27]	
	Feedback control loop	[51]	
	Adaptive mirroring	[8]	
	Sensor probing into functional layer	[39]	
	ADL inspired methods	(ACME) grid adaptation	[7]
		Darwin ADL and Jackson's alloy for runtime view	[19]
ArchJava language with Java for ubiquitous computing		[2]	
xADL 2.0		[12]	
Rapide for service discovery		[10]	
Component interaction	PitM	[45]	
	Isolation of faulty component regularities in architecture	[36]	
	Artificial law cybernetic foundation	[40]	
	Control rules for self-adaptive systems	[20]	
	Template design	[46]	
	Diversity maintenance	Functionality-based healing	[49]
Diversity in vulnerable places		[14]	
Identification of critical path		[26]	
Performance monitoring	Monitoring and reducing performance data	[29]	
	Predicting events	[48]	
	FSA for system rejuvenation	[21]	

legitimate usage by unmasking unintended implementation dependencies in benign code. This approach can be successful if it is low cost, having minimal impact on run time efficiency and maintainability.

3.2. Detection of system failure

Failure detection is another major area of research. The expectation is that any secured system should readily detect failure or the presence of malicious agents. It must be smart enough to gauge the degree of malfunction in the system, and whether the system actually needs a recovery or not. To cite an analogy in biological systems, we can say that the system should recover exactly as the body recovers from a wound while the other functions remain unaltered. Similarly, if any module of a system is under threat, other modules should ideally continue to perform as before. Several approaches have been adopted to detect non-self or abnormality in the functioning of the system. Research in this aspect can be broadly

categorized as dealing with missing components, system monitoring models and identification of foreign elements as shown in Fig. 9. Various policies to encounter failure detection are discussed in the sub-sections.

3.2.1. Something amiss — missing component, missing response, etc.

In this subsection, we discuss policies where the strategy is to sense something missing from the normal behavior of the system (such as those components are not meeting requirements, or that messages or elements are missing). Fig. 10 shows the different strategies, for detecting failure by sensing something amiss from regular behavior of a system. While the root node of this tree structure depicts the concept as a whole, the leaf nodes illustrate different strategies employed to sense irregularities in the system.

In the strategy proposed by Nagpal et al. [41], agents are always aware of the environment and can produce replications when they sense the disappearance of their 'neighbors'. Similarly, George et al. [18] point out that in the DWFS paradigm, any failure is sensed by the absence of messages. Aldrich et al. [2] explain that in their gardening module, the system features a simple policy to detect defects. It senses a defect when responses to a query are not received. In the service-discovery architecture, the system uses two policies to detect failures: monitoring periodic announcements and employing "bounding retries". Debrowski et al. [10] propose that failure in receiving scheduled announcements may indicate that the announcing component has failed, or that there is a blockage in the network path. Jini [24] provides broadcast messages about the accessibility of resources.

3.2.2. System monitoring model

This subsection deals with the policies suitable for architectures where the system monitors components by probing. Fig. 11 below depicts different models for monitoring the system and detecting any failure therein. While the root node of the tree diagram used here depicts the concept as whole, the leaf nodes illustrate different strategies for effective monitoring of the system.

Garlan et al. [17] have pointed out that monitored values can be abstracted and related to the architectural properties of a model. This is the typical model representation scheme adopted by most ADLs. Cheng et al. [6] propose that for Grid architecture, the probes and gauges report low level monitoring information that can be used to update an abstraction or to trigger an alarm. This model is also provided in Rainbow [5].

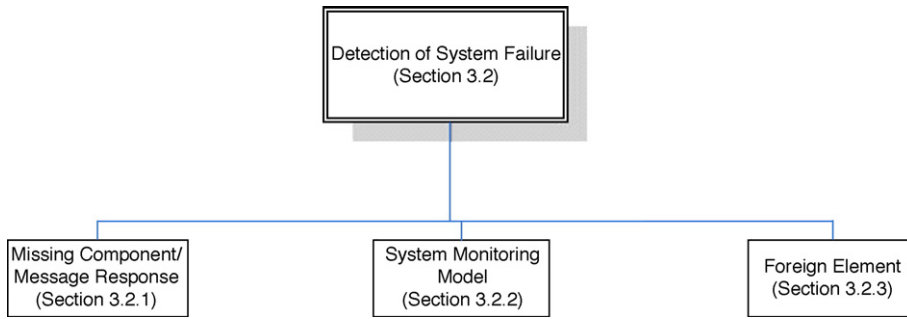


Fig. 9. System failure detection.

Similarly, components like Event packager [51] transform raw data collected from probes into smart, event-compatible event streams. Event distillers collect multifaceted event patterns from a probe’s data set and make an inference of the system state indicated by those events. This may be supported by an architectural transformation tool that acts in response to the gauges that detect differences between running and actual system architecture. In the external mirroring scheme by Combs et al. [8], we see that the probes navigate data into the parallel space where services are executed, and return. Services are requested and translated into workflows that represent the service commitments of service providers. The parallel space can adaptively substitute services which may fail with eligible substitutes.

Merideth et al. [38] propose that proactively probing a system and extracting information about it when a fault is detected could improve the survivability of the system. Dashofy et al. [12] propose that, two xADL 2.0 architecture descriptions can be given as input, to monitor if any component is added or is missing.

3.2.3. Notification of foreign element

In this subsection we review the literature that deals with the different approaches for detecting system failure by discerning foreign elements as depicted in

Fig. 12. While the root node in the tree diagram employed here depicts the failure detection concept, the leaf nodes represent different strategies designed to detect failure by using such methods.

Merideth et al. [38] propose that notification of the presence of a malicious replica is an essential part of a proactive containment strategy. Faulty processes typically attempt some communication; this scheme tries to minimize the period during which this is possible. This method tracks all possible avenues (secure and covert) to determine if a malice replica in a group of processes is affecting processes of other groups. Dashofy et al. [12] seek to represent the difference between two software architectures specified by xADL 2.0 in the form of an architectural difference document, called a “diff” document, that is generated by the system itself. This “diff” document also identifies the presence of foreign elements in the system.

Table 4 below gives a summarized view of the different strategies to detect failures in system. Column A depicts the various models for the same. Column B gives descriptions of various approaches to failure detection. Column C cites the relevant references.

In the above subsection we discussed various failure-detection policies. In Aldrich et al. [2] an important asset of the architecture is if a service needs to interact with a

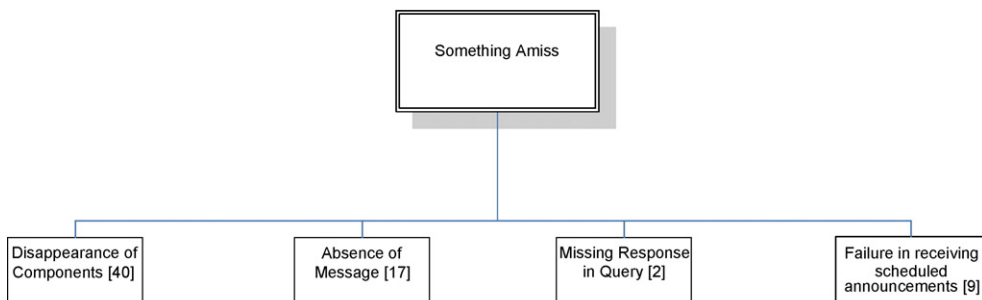


Fig. 10. Something amiss — missing messages, missing components, etc.

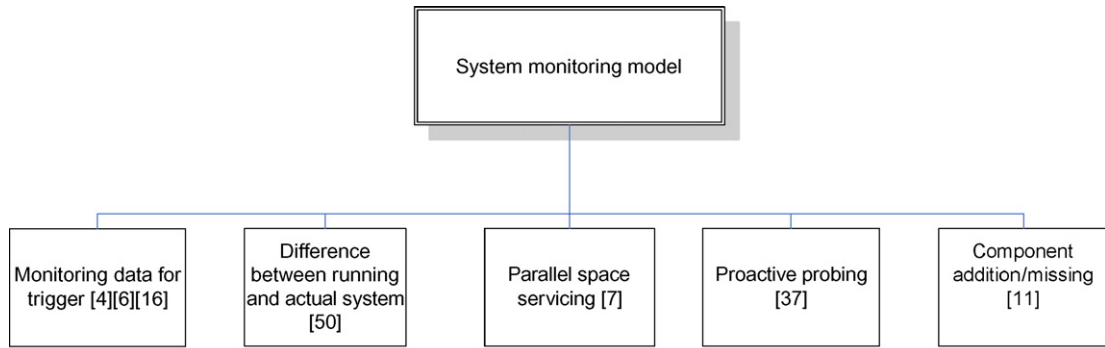


Fig. 11. System monitoring model.

device which cannot produce proper XML messages, a new connector can be used for further communication. Any existing service is reusable by connecting with the new connectors in the message protocol service of the new settings for further failure detection. This is obviously an advantage of using the architecture described here.

A few methods discussed here have weaknesses and they are expensive to maintain. For example, the proactive fault-containment systems as described in Merideth et al. [38] are costly to preserve and run. Enough information needs to be extracted from the system in order to enable untainted parts of the system to detect failures and defend them from impending malice. Identifying this information in distributed systems is difficult, because it cannot be found at a centralized location; therefore, it needs to be first aggregated from different sources, and then interpreted to yield something of value. Also, the failure detection system in George et al. [18] is suitable for only the models described, such as P2P network architecture.

3.3. System recovery from an unhealthy state to a healthy state

This section focuses on actual healing techniques as proposed in the literature. The main hallmark of self-

healing systems is the components dealing with healing. It is therefore expected that any self-healing system should readily apply the recovery policies for healing the abnormality in the system. This may consist of redundancy techniques like producing replicating components, applying various repair strategies, or employing Byzantine recovery and this forms the basis of the classification of the research in this aspect. Fig. 13 depicts the classification and also shows the subsections of this section. Various policies to tackle system recovery process are described below.

3.3.1. Redundancy techniques for healing

Nagpal et al. [41] suggest that for the model involving self-assembly of components, the system has the potential to effect self-repair and regeneration: agents can replicate components to replace dead neighbors and thus even recreate the entire structure. Regeneration can thus be effected even if a large part of the system is destroyed, as long as the system has a circle center and enough reference points. George et al. [18] point out during the healing process in a biological system, the system produces cells far in excess of what is necessary to combat the intrusion so that some may in every even survive the malicious action. Drawing an

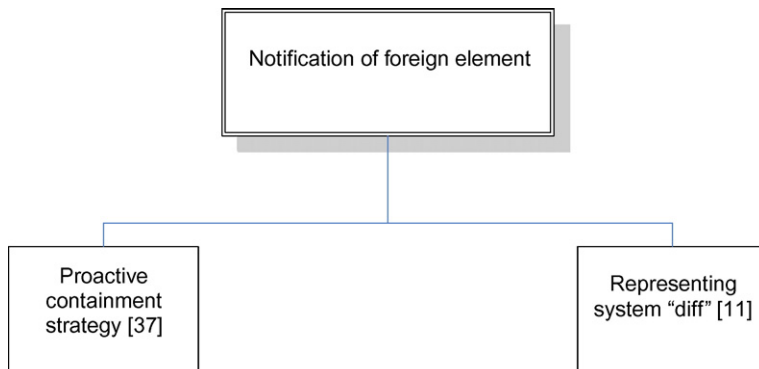


Fig. 12. Notification of foreign element.

Table 4
Failure detection policies

Models	Descriptions	References
Something amiss	Disappearance of components	[41]
	Absence of message	[18]
	Missing response in query	[2]
	Failure in receiving scheduled announcements	[10]
System monitoring model	Monitored data for trigger	[5,6,17]
	Difference between running and actual system	[51]
	Parallel space servicing/adaptive mirroring	[8]
	Proactive containment strategy	[38]
Notification of foreign element	Component addition or missing	[12]
	Proactive containment strategy	[38]
	Representing system “diff”	[12]

analogy in the DWFS scheme, failure can be repaired by proper action taken by the servers, like when the “replicate” nodes transmit messages to the “generate” nodes to initiate file distribution. The Recovery-Oriented Computing research project [23] at UC Berkeley is also utilizing a similar approach to effect the isolation of faulty components and provide redundancy techniques for fault containment and safe online recovery.

Fig. 14 depicts different approaches for system recovery. While the root node in this tree diagram depicts the concept as whole, the leaf nodes represent different strategies based on the recovery and regeneration principle.

3.3.2. Architecture models and repair strategies

Traditional exception catching mechanisms are often tightly integrated with the application and coupled with software code. Such mechanisms are fine for trapping any error or runtime problem, but they may not be able to find the true source of the problem. Also, these

methods are not well suited to catch the finer anomalies of the system, such as performance degradation or a prototype of unreliability.

The use of formal architectural models is advantageous for many reasons. These models allow non-local (i.e. global) properties to be monitored and non-local adaptations to be effected. These global externalized adaptations can be further extended; they can exploit shared monitoring and adaptation schemas.

Garlan et al. [17] propose that for a system to be self-healing, it should possess the capacity to adapt should system behavior falls outside admissible performance limits. Different repair strategies/plans can be provided for different system motivations. For example, an architectural style suited to the nature of the problem can be selected. Should the system be focused on performance, a repair style that depicts performance-related properties and makes constraints explicit may be adopted. Rainbow’s [5] framework is a low-cost, time-saving solution that enables software developers to add self-adaptation capabilities to various systems that use the framework described above. Based on the usage of repair plans, architectural models, constraints and component interactions we subdivide this section further.

3.3.2.1. Repair plans for healing. In this section we discuss the different repair strategies that have been proposed in the literature on self-healing. Fig. 15 below depicts different repair plans for system recovery. While the root node in this tree diagram depicts the concept as whole, the leaf nodes represent different strategies based on the recovery principle.

In the architecture proposed by Cheng et al. [6], the system information provided by gauges is used in Repair Strategies. These strategies firstly ascertain the cause of the problem and secondly, assess how to repair it. Repair strategies may form a sequence of preconditioned Repair Tactics that pinpoint the problem and determine appli-

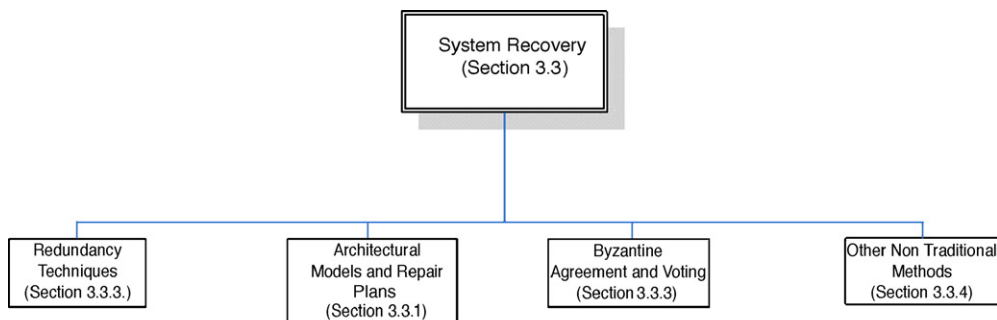


Fig. 13. System recovery techniques.

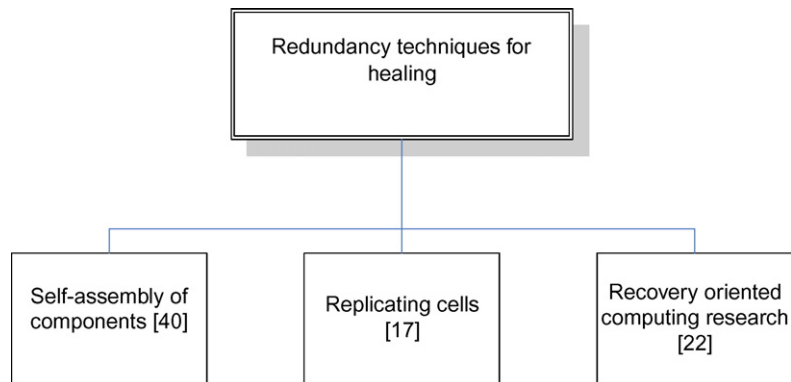


Fig. 14. Redundancy techniques for system recovery.

capability. The tactics execute an appropriate repair script depending upon the tactic chosen. Each self-healing module should be coordinated to act in concert with others to maintain consistency in information acquisition, and to ensure coherent decision making. Similarly, Valetto et al. [51] have proposed the utilization of high level repair action plans in the architecture, with feedback control loop in a targeted system for software adaptation.

Combs et al. [8] observe that in the adaptive mirroring strategy for external agents, the self-healing approach is limited to the replacement of service and connectors by a mirroring design plan. This architecture describes a Service and Contract (S+C) protocol, which can dynamically substitute a healthy service in place of a failed one. If any service of the software system fails, it can be replaced by a suitable substitution if one can be found. In the event-based configuration proposed by Dashofy et al. [12], the following steps for executing a repair plan have been outlined: (a) components and connectors about to be removed are given the chance to execute a clean up, save their states, or send a final message, (b) components fringing such unhealthy components are suspended to prevent messaging, (c) components, connectors and links are removed and

added as required and (d) the fringing components resume normal operation.

3.3.2.2. Component interaction-based healing. Literature relating to component interactions can be broadly classified as being either (a) computer binding to satisfy architectural constraints or (b) soft state and application level recovery or (c) isolation of faulty components as shown in Fig. 16. The literature pertaining to each of these healing strategies forms the focus of the discussion in this section.

Georgiadis et al. [19] have proposed a system architecture where, with the addition and removal of components, the system will remain well formed with respect to its specifications. When a change in current configuration takes place, each configuration manager computes the binding needed to satisfy the architectural constraints for each required port. The managers evaluate a set of configuration rules and re-evaluate selector functions. When all the required ports are bound, the system stabilizes.

In the service-discovery-based architecture proposed by Dabrowski et al. [10], the system supports two recovery techniques: soft-state recovery techniques,

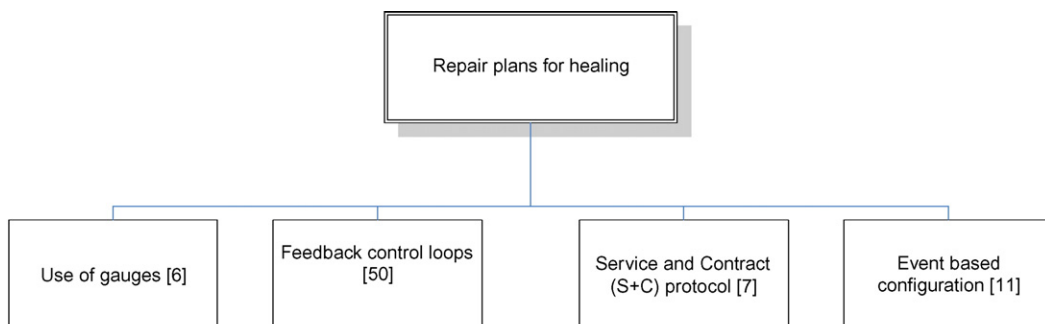


Fig. 15. Repair plans for healing.

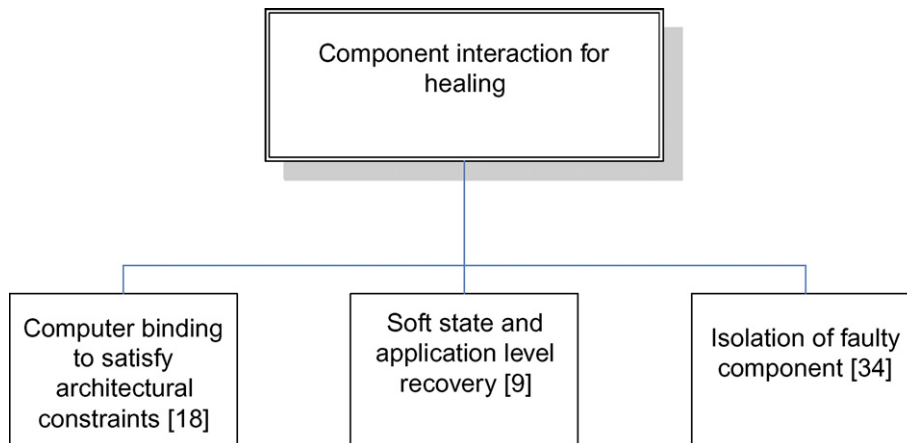


Fig. 16. Component interaction-based healing.

which involves periodic announcements about the state of the system, (where announcements can be cached until the system is updated) and application level persistence, which guided by typical application level persistency policies for recovery.

De Lemos et al. [35] have focused on the isolation of faulty components and the reconfiguration of a system. By externalizing all communications, it becomes possible to restrict the healing of faults occurring at the level of an element to the connectors through which that element interacts with the rest of the organization. For fault treatment, this approach relies on the dynamic reconfiguration of system architecture. The proposed solution would achieve system reconfiguration after performing a series of atomic operations until a stable state of the system configuration is attained.

3.3.2.3. Event-based action. There is not much literature in this area. Most predominant are the work of Aldrich et al. [2] and Dashofy et al. [12]. These papers primarily deal with systems where recovery action is taken based on an event. In the PlantCare System, (Aldrich et al. [2]) any component can self-heal itself in response to changes. In that system, the gardening module remains in a given state cycle if it fails to receive any query within a fixed time limit.

Similarly, Dashofy et al. [12] propose that, in the event-based configuration architecture, two xADL 2.0 architecture descriptions can be given as input, one being the current architecture and the other the proposed architecture which will obtain after the repair plan is activated.

An architecture-merging engine called ArchMerge can “apply” a “diff” (which is an architectural difference document which describes difference between two

software architectures) to the base architecture, transforming it into the targeted one.

$$d = \text{diff}(\text{architecture}_1, \text{architecture}_2),$$

$$\text{merger}(\text{architecture}_1, d) = \text{architecture}_2$$

It is also possible, however, to merge a patch into a new architecture; that is, instead of inputting the old and the new architectures, one may input only the new architecture and the patch that has been developed.

3.3.3. Voting methods for healing/Byzantine agreement

In this section we discuss research that relates to work that deals a system taking ion based on a voting mechanism. Merideth et al. [37] propose that for proactive containment of a malicious fault (which can taint other parts of the system by replication), using active replication with majority voting technique, the system can tolerate processor and process level faults, as well as arbitrary faults. This method tracks the possible avenues (secure and covert) to determine if a fault replica in a group of processes affects the processes of other groups. Byzantine agreement¹⁰ allows identifying a malicious processor, using active replication with majority voting. If a group of processes is infected with a malicious fault (which is replicating in the system), by using the voting algorithm on the

¹⁰ Lamport et al. [34] suggested that one way to achieve reliability is to have multiple replica of system (or component) and take the majority voting among them. The following two conditions should be satisfied for the same: (a) all non-faulty components must use the same input value and (b) if the input unit is non-faulty, then all non-faulty components use the value it provides as input.

output results of the processes faults can be detected and tolerated.¹¹

Similarly, as pointed out by Huhns et al. [25], in a multi-agent environment, multi-agent decision-making algorithms can be categorized into two elements: preprocessing decision making and post processing decision making. The preprocessing methods contain typical decision-making processes like lotteries (selecting results randomly among many multi-agents), auctions or voting methods (selecting the best outcome) and team approach (distributing tasks among agents and combining the collective results from the agents). In post processing, decision strategies are like selecting the agent that runs fastest or utilizes the least space. The post processing method also exploits a different kind of voting algorithm where all agents in the system compare their results to other agents and the result that receives most votes is the final answer.

3.3.4. Other methods — non-traditional approaches

The Recovery-Oriented Computing research [23] at UC Berkeley/Stanford is investigating novel techniques for building dependable internet-services. Diverting from traditional approaches of building fault-tolerant system, ROC¹² emphasizes *recovery* from failures rather than failure-avoidance. The research scope of ROC is described below in Table 5.

Failure reporting methodologies used by the PSTN [13], such as using several different metrics to report failures, also can be followed in the case of Dependable Systems. Architects of self-healing systems can learn from records of past failures to gain quantitative information that might help predict future failure of a system.

Table 6 below gives a summarized view of the different strategies to system recovery policies.

In the above subsection we surveyed various articles in the field of diverse policies of system recovery from failures. In Georgiadis et al. [19] the architecture as described is a fully decentralized architecture to support structural re-organization at runtime. A new component can be added or removed from the architecture, in response to any action external to the system. This is an important advantage if we are thinking of modular component-based architecture.

¹¹ The weakness with this strategy is the faulty component can provide correct values whenever its outputs are to be subjected for a vote, thus can escapes Byzantine fault detection [37].

¹² The ROC [23] approach takes the following three assumptions as its basic tenets: failure rates of both software and hardware are increasing, systems cannot be completely modeled for reliability analysis, human error by system operators and during system maintenance is a major source of system failures.

Table 5
Recovery-oriented computing

Research scope	Description
System-wide support for undo	Undo from system configuration to application management to software and hardware upgrades.
Integrated diagnostic support	Rapid detection of presence of failures and identifying their root causes. Masking of latent errors before they cause catastrophic chain-reaction failure.
Online verification of recovery mechanisms	Proactive testing proper behavior of the recovery mechanisms.

A similar benefit of using repair plans for healing [5–8,12,51] is that choosing of repair plans in action can be based on various techniques. For example, the system simply can choose to repair the client which reports an error first (first come first serve plan), or the system can prioritize (maintaining a priority queue) to repair a client which is experiencing the worst latency.

However, this architecture of using repair plans might fail in different situations, where the server load is too high and there is no available server for repairing. Also a situation may arise, where no prior repair plans can suffice or be suitable to mend the client for repair. Further, another drawback in [51] is that the usage of the feedback loop is ad-hoc in terms of the nature of the architecture.

4. Application of self-healing systems

Though most of the self-healing concepts are still in their infancy, there are several possible applicable areas for self-healing systems. We will briefly discuss the emerging applications of self-healing systems here. Industry leaders, such as Microsoft, IBM, SUN Microsystems are carrying out research on autonomous systems. Self-healing concepts are typically part of the same. These applications utilize self-healing techniques in their decision support systems, to enhance system capability and restore the normal state from the broken state. Grid computing, software agents, middleware computing are typically very promising software applications that cater to industry needs. If the self-healing concepts are coupled to these industry applications, the decision-making capability will be raised. Self-healing systems can work without human intervention and can take own decisions for healing, so the procedure of bringing back a system to normalcy from broken state will decrease maintenance time, thus saving money.

4.1. Grid computing

Grid applications must self-adapt themselves dynamically to changing environments. The heterogeneous

Table 6
System recovery policies

Models	Descriptions	References
Redundancy techniques for healing	Self-assembly of components	[41]
	Replicating cells	[18]
Repair plans for healing	Gauges are used in the repair strategies	[6]
	Feedback control loop	[51]
	Service and contract (S+C) protocol	[8]
	Event-based configuration	[12]
Component interaction-based healing	Computes binding to satisfy architectural constraints	[19]
	Soft state and application level recovery	[10]
Event-based action	Isolation of faulty components	[35]
	Start same state cycle	[2]
	Representing system “diff”	[12]
Voting method	The proactive containment of malice	[37]
	Multi-agent decision making algorithm	[25]
Other methods	System-wide support for undo	[23]
	Integrated diagnostic support	[23]
	Online verification of recovery mechanisms	[23]
	Learning from failure records	[13]

nature of the network and the running modules, the dynamic load balancing requirements, and the ever-changing user needs make adaptation an important necessity for grid applications. As discussed earlier, Cheng et al. [6] have proposed a novel software architecture mode for Grid Computing, which can be maintained at runtime and can be used as a basis for system configuration and adaptation. System monitoring using “probes and gauges”, error detection and an adaptation scheme of this has been discussed already.

4.2. Software agent-based self-healing architecture

Huhns et al. [25] presented the concept of multi-agent redundancy to fabricate software adaptation. Software engineering intersects multi-agent systems in many ways. Such as, multi-agent systems can be used to assist conventional software systems or traditional software engineering techniques can be used to build multi-agent systems.

The benefits of using agents as building blocks for conventional software are: agents can dynamically compose in a system when all the components of the system are unknown till runtime. Also, as agents can be added to a system in run time, software can be customized over its lifetime, even by the end-users too. This can produce more robust systems.

4.3. DWFS application

A biologically inspired programming model for a self-healing system is designed with the paradigm of the cell division approach in [6]. This paradigm is illustrated with a real world scenario with Distributed Wireless File Service (DWFS), an application layer peer-to-peer file sharing service.

This approach is too application dependent; if the application needs to transmit or communicate data into various replicated nodes (or replicate data), this model holds well. Though designing real life complex projects is not possible only by a system state diagram, work is continuing in the process of producing a robust self-healing system that can accomplish a complex task.

4.4. Service discovery systems

These are the systems that provide reliable views of distributed systems in different network conditions. Dabrowski et al. [10] have proposed an architecture-based adaptation in a service discovery system. The protocol of emerging service discovery systems provides the groundwork for finding components in the system, organizing themselves, and adapting themselves to changes in system topology. The consistent maintenance policies too can help in effecting self-healing in distributed computing. This module utilizes various consistency maintaining mechanisms and failure detection and recovery techniques.

4.5. Reflective middleware

Middleware exists in between the operating system and the application level, thus mediating and facilitating interaction and communication between them. Technologies like OMG’s CORBA, Sun’s J2EE, and Microsoft’s .NET are some example of industry standard middleware technologies. The major advantages of using these technologies are that they typically hide the technical detail of network communication, remote procedure calls, and construction of complex distributed system. Usually the codes that run on top of the middleware are portable and the users do not bother about the operating system or network detail.

However, some applications that run on top of the middleware can enhance their performance if they are aware of the facts in the underlying details. For example, for client–server architecture if the system is conscious about the resource utilization or the middleware’s request scheduling process it can improve the load-balancing of the system or can create replicas of its most used service.

The above discussion about middleware raises the subject of Reflective Middleware model. Kon et al. [31] have proposed that the reflective middleware model, unlike traditional middleware, is represented as a compilation of various components that is reconfigurable. In this paradigm, it is possible to select networking protocols, security policies and various other mechanisms to improve the performance of an application, keeping the interfaces of middleware unchanged.

We will briefly discuss two different approaches to reflective middleware, namely dynamicTAO [30] and Open ORB [4].

4.5.1. *dynamicTAO*

dynamicTAO is created to enable the reconfiguration of the ORB internal engine of the TAO ORB. Component Configurators are the typical C++ objects here, and they store the dependency relationships between ORB components and application components as a list of references. dynamicTAO supports the safe dynamic reconfiguration of the middleware components, whenever a request for a component turns up, the system checks the dynamic dependencies between the component and middleware and other application components using the Component Configurators object. This architecture facilitates a meta-interface for dynamically loading and unloading modules, changing the ORB's configuration states. dynamicTAO offers support for the interceptor, which is a part of TAO. We will discuss the interceptor approach to a reliable system later.

4.5.2. *Open ORB*

The Open ORB project architecture maintains the components as identifiable entities, which in turn facilitates runtime reconfiguration. There is a clear separation of base and Meta levels of the architecture here: where the base level executes the usual middleware services, the Meta level is provided to facilitate the reflective actions, adaptation, etc. To support behavioral reflection the Open ORB supports different meta-models such as interception and the resource meta-model. Some examples of the self-adaptive nature of Open ORB, such as self-adaptive stream binding, self-adaptive mobile middleware, etc. can be found in [4].

4.5.3. *Interceptor-based approach*

The Eternal system [43] is a CORBA 2.0 compliant system, which enhances the fault tolerance nature of CORBA with replication. Narasimhan et al. [42] have proposed the Eternal architecture, which is based on the interception approach, by capturing Internet Inter-ORB Protocol (IIOP) specific system calls made by the ORB.

The interceptor calls, which were originally directed by the ORB to TCP/IP, are now mapped onto a reliable ordered multicast group's communication system. In this method, the advantage of the approach is that neither the ORB nor the objects need ever be aware of being interpreted, nor is the fault tolerance visible from the application objects. In the Eternal system, a replicated object enables any client object to address the replicas of a server object as a whole using a unique object group's identifier.

4.6. *GRACE approach*

The goal of the GRACE project is to develop an integrated cross-layer adaptive system, where the hardware and all the software layers cooperatively adapt to the changing demands of system resources and applications while meeting resource constraints of energy, time, and bandwidth and while also providing the best possible Quality of Service. This architecture proposes that all system layers be able to adapt in response to system or application changes. Adve et al. [1] suggest that the various layers of a system, such as hardware, operating systems, the scheduler layer and network protocols be coordinated at a fashion that enables them to adapt to changing system resources and application demands. This architecture is well suited for applications running on resource-constrained systems, such as multi-media applications in mobile devices.

4.7. *Clustering*

Self-healing strategies also can be implemented in Clustered systems (a collection of computers, heterogeneous or homogeneous, connected by a private network that facilitates the cluster being used as a combined resource). Corsava et al. [9] suggest a scheme to manage application inter-dependencies, which ensures the allocation of various resources to the running systems as and when needed, with no service interruption. This also serves to enhance security and recovery from disasters. Corsova et al. [9] argue that this scheme is different in many aspects from typical available clusters such as Solaris, HP-UX or AIX, in that the latter are fault-tolerant but not self-healing, and are furthermore most often non-secure. Table 7 shows the application areas of Self-healing strategies. Column A represents the different application models. Column B shows the references.

5. Conclusion and further research

There is a crying need in the world of business for reliable and dependable computing. The goal of self-

Table 7
Application areas of self-healing strategies

Models	References
Grid computing	[6]
Software agent-based computing	[25]
DWFS application	[18]
Service discovery systems	[10]
Reflective middleware (dynamicTao)	[30]
Reflective middleware (Open ORB)	[4]
Reflective middleware (Eternal)	[42,43]
Grace project	[1]
Clustering	[9]

healing systems is to provide survivable systems that are reliable, highly available and dependable. Self-healing systems are an integral part of most biological systems, which exhibit properties such as adaptation, mutation, self-preservation, etc. The mechanisms used for self-preservation, adaptation, etc. provide good inspiration and a rich source of ideas. Though there is some research in this area, and we are beginning to see some commercial interest, this area is still in its infancy.

Self-healing systems are becoming more popular and industry software systems and hardware architecture are being developed with self-healing properties. Microsoft MSI files for fixing Windows XP by creating a system restore point, inclusion of self-healing capabilities in IBM's WebSphere initiative, etc. provide testimony to this trend. IBM has now developed the "Autonomic Blueprint" and "on-demand" initiative based around the concept of self-healing. Microsoft announced its Dynamic Systems Initiative (DSI) and Sun Microsystems has a detailed plan for its N1 technologies and utility computing that make use of the self-healing concept. Hewlett-Packard's Adaptive Enterprise strategy is just another example.

The structure of self-healing systems is quite modular (for example, failure detection, system recovery and maintenance of health). This provides for possibilities to improve and generate various types of architectures for the different modules. Typically self-healing systems are based on different paradigms of computational models (such as, architectural models, repair plans, agent-based models) or by imitating natural (biological) models, so the opportunity to develop systems based on new sources inspiration is substantial. Developmental biology can educate us on how to make complex architectures, with well-defined topological and practical structures or how to focus on self-assembly of components (pointed out by Nagpal et al. [41]). From a managerial and decision sciences perspective there is a need for research relating to cost effectiveness of the solutions. There is a tradeoff between the gains from self-healing, the cost and performance. In fact there is no

currently published research that deals with managerial issues as it relates to self-healing systems.

Much of the literature focuses on repair plans for single process systems. However there is a great lacunae of work that relates to distributed architectures which are increasingly becoming commonplace. Undoubtedly recovery issues relating to distributed systems are more complex for a variety of reasons. This is therefore a new area for research as it relates to self-healing systems.

Another important area of research that would make self-healing systems more appealing is the necessity to improve speed of recovery. Recovery should be quick so that the malfunctioning system can be restored to normalcy from the broken state as soon as possible. Caching of repair plans and architectures to support expedient propagation of repair plans to the broken server will help to expedite the recovery. The recently employed repair plans can be stored in a "cache" so that if an analogous fault occurs in the system the probable repair plan to mend the fault can be selected from the "cache" of repair tactics. However this area needs to be studied from a performance as well as cost effectiveness perspective.

Yet another area for further research is in the development of intelligent adaptive architectures that allow for human intervention in cases where there are no prior-saved or existing repair plans. Automatic and adaptive coordination can be planned with diagnostic messages to guide administrators through tasks that require human intervention.

From an algorithmic perspective self-healing systems should have a roll-back facility for all aspects of system operations, configurations and application management. This is a very challenging task as the system should quickly determine the fault and restore itself automatically back to the original state (or normal state) with an undo plan. This includes a scheme of untangling the problem, rewinding and replaying the system back. The ROC project [23] and Cylab at CMU are working towards this research goal.

New research directions also include the creation of systems and architectures that have system-behavior prediction models appended to the system maintenance schema in self-healing systems. This will result in behavioral reflection which will allow for runtime information to be used to extract failure and model information. The data will provide time dependent representation of the system during attacks on the system as well as when components begin to malfunction. An intelligent adaptation of machine learning algorithm can automatically classify and predict software behavior based on execution data (such as by profiling event transitions or system calls).

Self-healing systems should include components that analyze the performance and error log intelligently and

quickly. The volume of performance data logged by the system can become very large and research is needed to support intelligent data reduction and quick analysis. The analysis can also lead to learning about behaviors before the occurrence of certain states of the system and this can then be used for predictive purposes. Further, the study of performance log will catalyze the discovery of any unapparent information, dependency and similarity of information between various error logs. Current literature does not address these areas.

This paper presents an in-depth review of the literature along with a survey and a synthesis. The reviewed literature is presented using a taxonomy which we have developed. Self-healing systems are interactive, flexible and adaptive computer-based information system. Such systems can be developed for supporting the solution of a non-structured management problem for improved decision making. As such there is no research that provides behavioral or economic insight regarding self-healing systems.

Self-healing systems are relatively new both for the academia and the industry. However, we are likely to see a large number of systems, software and architectures that borrow from nature, ideas and concepts. Biological systems have exhibited characteristics such as evolution, adaptation, self and assisted healing including social networking. These are good models to emulate and recreate to provide functionality defense through heterogeneity and social networking. Modeling computer security using biology as a motivation can lead to adaptive survivable systems that provide functionality despite the possibility of catastrophes.

References

- [1] S. Adve, A. Harris, C.J. Hughes, D.L. Jones, R.H. Kravets, K. Nahrstedt, D.G. Sachs, R. Sasanka, J. Srinivasan, W. Yuan, The Illinois GRACE Project: Global Resource Adaptation through Cooperation, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [2] J. Aldrich, V. Sazawal, C. Chambers, D. Nokin, Architecture-centric programming for adaptive systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [3] A. Avizienis, J. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (2004).
- [4] G.S. Blair, G. Coulson, L. Blair, H.D. Limon, P. Grace, R. Moreira, N. Parlavantzis, Reflection, self-awareness and self-healing in OpenORB, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [5] S.W. Cheng, A.C. Huang, D. Garlan, B. Schmerl, P. Steenkiste, Rainbow: architecture-based self-adaptation with reusable infrastructure, IEEE Computer 37 (10) (2004).
- [6] S.W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, N. Hu, Software architecture-based adaptation for grid computing, The 11th IEEE Conference on High Performance Distributed Computing (HPDC'02), Edinburgh, Scotland., 2002.
- [7] S.W. Cheng, A.C. Huang, D. Garlan, B. Schmerl, P. Steenkiste, An architecture for coordinating multiple self-management systems, Proceedings of the 4th Working IEEE/IFIP Conference on Software Architectures, 2004.
- [8] N. Combs, J. Vagle, Adaptive mirroring of system of systems architectures, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [9] S. Corsava, V. Getov, Self-healing intelligent infrastructure for computational clusters, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [10] C. Dabrowski, K.L. Mills, Understanding self-healing in service-discovery systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [11] D. Dasgupta, F. Gonzalez, Artificial immune systems (AIS) research in the last five years, Proceedings of the International Conference on Evolutionary Computation Conference (CEC), Canberra, Australia, 2003.
- [12] E.M. Dashofy, A.V.D. Hoek, R.N. Taylor, Towards architecture-based self-healing systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [13] P. Enriquez, A. Brown, D. Patterson, Lessons from the PSTN for dependable computing, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [14] S. Forrest, A. Somayaji, D. Ackley, Building diverse computer systems, In Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, 1997.
- [15] S. Forrest, A. Somayaji, S.A. Hofmeyr, Computer immunology, Communications of the ACM 40 (10) (1997) 88–96.
- [16] A.G. Ganek, T.A. Korbi, The dawning of the autonomic computing era, IBM Systems Journal 42 (1) (2003).
- [17] D. Garlan, B. Schmerl, Model-based adaptation for self-healing systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [18] S. George, D. Evans, S. Marchette, A biological programming model for self-healing, First ACM Workshop on Survivable and Self-Regenerative Systems, 2003.
- [19] I. Georgiadis, J. Magee, J. Kramer, Self-organizing software architectures for distributed systems, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [20] E. Grishikashvili, Investigation into Self-Adaptive Software Agents Development, Distributed Multimedia Systems Engineering Research Group Technical Report, 2001.
- [21] Y. Hong, D. Chen, L. Li, K. Trivedi, Closed loop design for software rejuvenation, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [22] <http://research.microsoft.com/dtas/>.
- [23] <http://roc.cs.berkeley.edu/>.
- [24] <http://www.jini.org/>.
- [25] M.N. Huhns, V.T. Holderfield, R.L. Gutierrez, Robust software via agent-based redundancy, AAMAS'03, 2003.
- [26] P. Inverardi, F. Mancinelli, G. Marinelli, Correct deployment and adaptation of software application on heterogeneous (mobile) devices, Proceedings of the First Workshop on Self-Healing Systems, 2002.
- [27] G. Kaiser, P. Gross, G. Kc, J. Parekh, G. Valetto, An approach to autonomizing legacy systems, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [28] K. Knapp, F. Morris, K. Rainer Jr., T.A. Byrd, Defense mechanism of biological cells: a framework for network security thinking, Communication of the AIS, vol. 12, 2003.

- [29] M.W. Knop, J.M. Schopf, P.A. Dinda, Windows performance monitoring and data reduction using watch tower, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [30] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L.C. Magalhaes, H. Campbell, Monitoring, security, and dynamic configuration with the dynamicTAO Reflective ORB, Multimedia Middleware Workshop, IFIP/ACM International Conference on Distributed Systems Platforms. New York, United States, 2000.
- [31] F. Kon, F. Costa, G. Blair, R.H. Campbell, The case for reflective middleware, *Communications of the ACM* 45 (6) (2002).
- [32] R. Laddaga, Active software, 1st International Workshop on Self-Adaptive Software, 2000.
- [33] A. Lamarca, W. Brunette, D. Koizumi, M. Lease, S.B. Sigurdsson, K. Sikorski, D. Fox, G. Borriello, PlantCare: an investigation in practical ubiquitous systems, *UbiComp*, 2002.
- [34] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, *ACM Transactions on Programming Languages and Systems* 4 (3) (1982) 382–401.
- [35] R.de. Lemos, J.L. Fiadeiro, An architectural support for self-adaptive software for treating faults, *Proceedings of the First Workshop on Self-Healing Systems*, 2002.
- [36] Z. Lu, J. Hein, M. Stan, J. Lach, K. Skardron, Control-theoretic dynamic frequency and voltage scaling, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [37] M.G. Merideth, Enhancing survivability with proactive fault-containment, *The 2003 International Conference on Dependable Systems and Networks (DSN-2003)*, 2003.
- [38] M.G. Merideth, P. Narasimhan, Proactive containment of malice in survivable distributed system, *International Conference on Security and Management*, Las Vegas, NV, 2003.
- [39] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, P. Verbaeten, Self-adapting concurrency: the DMonA architecture, *Proceedings of the First Workshop on Self-Healing Systems*, 2002.
- [40] N.H. Minsky, On condition for self-healing in distributed software systems, *Autonomic Computing Workshop Fifth Annual International Workshop on Active Middleware Services (AMS'03)*, 2003.
- [41] R. Nagpal, A. Kondacs, C. Chang, Programming methodology for biologically-inspired self-assembling systems, *AAAI Symposium*, 2003.
- [42] P. Narasimhan, L.E. Moser, P.M. Melliar-Smith, The interception approach to reliable distributed CORBA objects, *Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, 1997.
- [43] P. Narasimhan, L.E. Moser, P.M. Melliar-Smith, A component-based framework for transparent fault-tolerant CORBA, *Software, Practice and Experience* 32 (8) (2002) 771–788.
- [44] V.P. Nelson, *Fault-Tolerant Computing: Fundamental Concepts*, IEEE Computer Society Press, 1990.
- [45] M.M. Rakic, N.R. Mehta, N. Medvidovic, Architectural style requirements for self-healing systems, *Proceedings of the First Workshop on Self-Healing Systems*, 2002.
- [46] O. Raz, P. Koopman, M. Shaw, Enabling automatic adaptation in systems with under-specific elements, *Proceedings of the First Workshop on Self-Healing Systems*, 2002.
- [47] O. Raz, P. Koopman, M. Shaw, Semantic anomaly detection in online data sources, *24th International Conference on Software Engineering (ICSE'02)*, 2002.
- [48] R.K. Sahoo, M. Bae, R. Vilalta, J. Moreira, S. Ma, M. Gupta, Providing persistent and consistent resources through event log analysis and predictions for large-scale computing systems, Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN), 2002.
- [49] R. Sharman, H.R. Rao, S. Upadhyaya, P. Khot, S. Manocha, S. Ganguly, Functionality defense by heterogeneity: a new paradigm for securing systems, *37th Hawaii International Conference on System Sciences*, 2004.
- [50] M. Shaw, Self-healing: softening precision to avoid brittleness, *Position Paper for WOSS'02: Workshop on Self-Healing Systems*, 2002.
- [51] G. Valetto, G.E. Kaiser, Case study in software adaptation, *Proceedings of the First Workshop on Self-Healing Systems*, 2002.

Debanjan Ghosh received his master's degree in Computer Science and Engineering at SUNY Buffalo in 2006. His thesis work was on sentence level event mining with kernels using Support Vector Machines. His research interests include information extraction, text mining, kernel methods and self-healing systems.

Dr. Raj Sharman is a faculty member in the Management Science and Systems Department at SUNY Buffalo, NY. He received his PhD in Computer Science from the Louisiana State University in 1998. He is currently involved in several research projects in the areas of Emergency Management Systems, Medical Image Registration and the use of biologically inspired computer security models. He received his Bachelors degree in Engineering and Masters Degree in Management from the Indian Institute of Technology, Bombay, India.

Dr. Rao's interests are in the areas of management information systems, decision support systems, and expert systems and information assurance. He has chaired sessions at international conferences and presented numerous papers. He has authored or co-authored more than 100 technical papers, of which more than 80 are published in archival journals. His work has received best paper and best paper runner-up awards at AMCIS and ICIS. Dr. Rao has received funding for his research from the National Science Foundation, the Department of Defense and the Canadian Embassy and he has received the University's prestigious Teaching Fellowship. He has also received the Fulbright fellowship in 2004. He is a co-editor of a special issue of *The Annals of Operations Research*, the *Communications of ACM*, associate editor of *Decision Support Systems*, *Information Systems Research* and *IEEE Transactions in Systems, Man and Cybernetics*, and co-editor-in-chief of *Information Systems Frontiers*.

Shambhu J. Upadhyaya, Ph.D. is an Associate Professor of Computer Science and Engineering at SUNY Buffalo where he also directs the Center of Excellence in Information Systems Assurance Research and Education (CEISARE), designated by the National Security Agency. Prior to July 1998, he was a faculty member at the Electrical and Computer Engineering department. His research interests are information assurance, computer security, fault diagnosis, fault tolerant computing, and VLSI Testing. He has authored or coauthored more than 165 articles in refereed journals and conferences in these areas. His current projects involve intrusion detection, insider threat modeling, security in wireless networks, analog circuit diagnosis, and RF testing. His research has been supported by the National Science Foundation, Rome Laboratory, the U.S. Air Force Office of Scientific Research, DARPA, and National Security Agency. He is a senior member of IEEE.