
Lead contractor: Lero

Submission date: December 2010
Dissemination level: Public

Project co-founded by the European Commission under the Seventh Framework Programme
© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: As software maintenance processes still rely on human expertise and manual intervention, they usually require important efforts in terms of time and costs. Research has been conducted in the past recent years in order to try to overcome this issue by considering systems that would maintain themselves. This concept gave birth to Autonomic Computing (AC), which aims to transfer and automate part of the maintenance processes to the system itself. Self-healing is one of the properties that AC seeks to equip system with. Self-healing aims for the system to automatically recover from faults, possibly fixing them through patch generation. This document introduces self-healing approaches as well as other research areas that are very related to this concept, such as fault-tolerance, Automatic Diagnosis and Automatic Repair. The latter contains works related to automatic patch generation.

This document has been produced in the context of the FastFix Project. The FastFix project is part of the European Community’s Seventh Framework Program for research and development and is as such funded by the European Commission. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability is respect of this document, which is merely representing the authors view.
## Contents

1 Introduction ......................................................... 7

2 Foundation ......................................................... 10
   2.1 Software Maintenance ......................................... 10
      2.1.1 Definition .............................................. 11
      2.1.2 Software Maintenance Categories ...................... 11
      2.1.3 Software Maintenance Processes ...................... 12
      2.1.4 Debugging and Patch Generation ....................... 13
   2.2 Autonomic/Self-Managing Systems ......................... 16
      2.2.1 Autonomic Elements .................................... 17
      2.2.2 Architecture-Based Self-Healing ...................... 18
      2.2.3 Biological Inspiration ................................ 19
      2.2.4 Current State .......................................... 20
      2.2.5 Research Venues ....................................... 20
      2.2.6 AC Formalisms, Initiatives, and Frameworks ......... 23

3 Fault Tolerance and Self-Healing ............................. 33
   3.1 Fault Tolerance .............................................. 33
      3.1.1 Single-Version Techniques ............................. 37
      3.1.2 Multi-Version Techniques ............................. 38
   3.2 Self-Healing .................................................. 39
      3.2.1 Research articles on Self-Healing .................... 41
      3.2.2 Other EU projects on Self-Healing .................... 43
   3.3 Summary ..................................................... 44

4 Automatic Diagnosis and Repair ................................. 47
   4.1 Automatic Diagnosis ......................................... 47
      4.1.1 Architecture-Based Approaches ....................... 47
      4.1.2 Correlation-Based Approaches ....................... 48
      4.1.3 Probabilistic-Based Approaches ..................... 50
      4.1.4 Model-Based Approaches .............................. 51
4.1.5 Summary and Conclusion ........................................ 52
4.2 Automatic Repair and Bug Fixing ................................. 52
  4.2.1 Rollback Techniques ........................................... 53
  4.2.2 Genetic Programming and Mutations Techniques .......... 55
  4.2.3 Event Filtering Techniques ...................................... 60
  4.2.4 Learning and Probabilistic Approaches ...................... 62
  4.2.5 Model-Checking Based Techniques ............................. 65
  4.2.6 Other Approaches .................................................. 66
  4.2.7 Summary and Conclusion ........................................ 67

5 Conclusion ............................................................ 69
List of Figures

2.1 The ISO/IEC 12207 maintenance process [58]. ........................................... 12
2.2 AE Architecture [92]. ............................................................................... 18
2.3 ASSL Multi-tier Specification Model [120]. ............................................. 24
2.4 Self-managing Features in Tivoli Software [87]. ...................................... 26
2.5 Automated Software Distribution Mechanism for Self-healing[119] .......... 27
2.6 Darwin Architecture for the Adaptive Enterprise [49]. ......................... 28
2.7 Rainbow Framework [21] ................................................................. 32

3.1 Recovery Block Model ................................................................. 39
3.2 The self-healing loop [70]. ............................................................. 40
3.3 The architecture of self-healing systems proposed by the SHADOWS project [108] ................................................................. 45

4.1 Creation of application signatures from runtime behaviors [35]. .......... 49
4.2 Rx Methodology [95] ........................................................................ 54
4.3 General Architecture of FLIPS[78] ................................................... 61
4.4 ShieldGen System Overview [27] ....................................................... 62
4.5 ClearView Architecture [94] ............................................................ 64
4.6 The AutoFix-E approach [127]. ......................................................... 66
List of Tables

4.1 Summary on Automatic Repair techniques . . . . . . . . . . . . . . . . . . . . . . . . . 68
1 Introduction

The software development cycle aims to produce systems with better quality in terms of reliability and dependability. Traditionally this cycle comprises several phases such as Design, Development, Testing, Deployment and Maintenance. Even though all these steps help ensure a better quality of the software to be produced, it is however not error free. The main reasons for this are the complexity of the developed systems as well as their evolution over time (or the evolution of their environment). Therefore planning before deployment every way the system will be used and interact with his environment is a fairly unreachable objective. There is therefore a need for handling issues that raise once the system is deployed and in use, i.e. during runtime execution. This is one of the main roles of the maintenance process (with addition of new features). This goal becomes even more important for critical systems for which errors may have dramatic consequences. This obviously applies to systems such as spaceships or automatic underground trains where errors can jeopardize lives. It can also simply apply to companies whose business relies on the system and for which recurrent losses may in worst cases lead to bankruptcy and in general prevent the company from growth and competitiveness. With this in mind, it appears that it is crucial for the maintenance process to be extremely reactive to prevent failures and correct issues as quickly as they raise. Unfortunately, this is a rather difficult task for the same reasons as mentioned previously (system, environment and interaction complexity) and it can require an important investment in terms of people and time in order to be performed promptly.

Self-healing is a concept which aims to tackle this problem by automating the maintenance task in presence of system failure. This concept came with the notion of Autonomic Computing initiated by IBM in [67]. The main idea of Autonomic Computing is to develop techniques for computer systems which are inspired from biological processes. Self-healing definitely falls into this category and can be defined from there as “a mechanism which detects malfunctions and respond to them without any external intervention”. In the case of software systems, this can be associated to different existing concepts such as fault-tolerance, diagnostics, repair, recovery, etc. This document aims to present research works that were conducted in the area of self-healing. The difficulty of achieving this goal is twofold:

First there are few research venues that are dedicated to self-healing. To our knowledge,
the “Workshop on Self-Healing System (WOSS)” is the only venue whose scope was focused on self-healing. One reason for this is probably that research related to the self-healing are of interest and find their place in many other venues, especially the ones related to Autonomic Computing or Self-Adaptive Systems such as IBM System Journal, IJAACS, IJAC, JoATC, SESAS, TEAS, ADAPTIVE, EASE, ICAC, ICAS, IWSAS, SASO, SEAMS, SELFMAN, SELFSTAR, SORT and WASELF. However, looking through the proceedings of the mentioned conferences (including WOSS), we established that around 800 articles were published in these venues over the years among which only 27 were explicitly concerned with self-healing (i.e. “self-healing” was part of the title), representing less than 4% of the publications.

Secondly there is a lot of work that is related to self-healing but which uses a different terminology. The most typical area in this case is fault-tolerance. Self-healing and fault-tolerance are so tightly related that researchers felt the need for a seminar which aimed to explicit the differences between them as in [31]. It needs to be pointed out that fault-tolerance is a much older research topic than Self-healing and was initiated for hardware systems while self-healing is focusing more on software systems. Because of the precedence of fault-tolerance, this area clearly inspired the techniques described later for self-healing.

But fault-tolerance is only the obvious topic of several ones that are closely related to self-healing. In [102] for instance, the authors describe self-healing as consisting of self-diagnosis and self-repair. The consequence of this observation is that works related to automatic diagnosis and automatic repair or recovery are relevant to the field of self-healing. Moreover, as discussed in works such as [42, 97], even non fully automated healing approaches may fall into the category of self-healing techniques (in [42] the authors introduce the notion of assisted-healing for systems that require some human intervention during their healing process). This greatly broaden the research field that is relevant to self-healing.

Under these considerations, this document is structured in the following way:

First as self-healing approaches rely on Autonomic Computing principles and aim to facilitate and automate the software maintenance process, Chapter 2 recalls some foundational aspects of software maintenance and Autonomic Computing.

Then Chapter 3 describes fault-tolerance and self-healing techniques. These two concepts share indeed similar techniques and from a historical point of view, self-healing followed inspiration from fault tolerant approaches. Section 3.2.1 presents works on self-healing that are explicitly presented as “healing” approaches, i.e. from the self-healing community.

As previously explained, self-healing can be seen as a combination of automatic diagnosis and automatic repair or recovery. The type of errors handled, as well as the techniques used differ from the works presented in Section 3.2.1 in that case. For this
reason we present these works in a different Chapter (i.e. Chapter 4) which focuses on automatic diagnosis and repair. In this document, as well as in the presented articles, automatic repair also refers to automatic patch generation. Finally, Chapter 5 presents some conclusions on this report.
2 Foundation

The inherent complexity of today’s software systems and the limited availability of resources within a competitive environment makes the development of reliable software a difficult goal. Though there are many analysis and testing techniques for improving the quality of software during development, software products are still being released with missing features and errors. This is due to the fact that these techniques until now show limited success or are not used because of time-to-market pressures and limited development resources. Furthermore many of today’s software products are run in complex and variable environments: a software product may need to interact through a network with software products running on other computers; a software product may also need to be able to run on computers with different configurations, hence it may be impractical to analyze and test these software products before their release under all possible runtime environments and configurations.

Software maintenance deals with handling resulting errors as well as additional enhancements. This section recalls some parts of Deliverable D2.1 on “State-of-the-art in Monitoring Control for Remote Maintenance” and presents some basic overview of software maintenance and shows techniques and practices from the corrective part of software maintenance, which concerns the identification and removal of errors in software systems. A section presenting generalities on debugging and patch generation is also introduced.

2.1 Software Maintenance

Modifiability of software systems after delivery is a necessary quality, even though software does not deteriorate with time and while being in use. According to Lehman’s laws of evolution [75], successful software systems must change over time. A predominant proportion of changes is to meet ever-changing user needs which are essential to keep a software system useful. Significant changes are also due to the fact that software needs to be adapted to interact with external entities, such as users, security concerns, organizations, and other (software) systems. Since software is a very flexible concept, it is often considered the easiest part to change in a system [15]. A quite big part of necessary changes results finally from the fact that software products are still being released with missing functionality and latent errors.
This section presents software maintenance as the process that deals with these problems. It shortly explains the different categories of software maintenance and provides an overview of current software maintenance processes.

2.1.1 Definition

The term “Software Maintenance” is used for a very broad range of activities, often defined as all changes made on a software system after it has become operational [82]. This description covers the correction of errors, the enhancement, removal and addition of functionality, the adaptation of the system to changes in data and operation requirements and also the improvement of the system regarding non-functional requirements like performance or usability. Although the common view is that software maintenance is an activity that takes place after the delivery of the system, beginning with the release of the system to the customer or user and comprising all activities ensuring that the system remains operational and suits the needs of its users, the reality is different. In all current lifecycle models software maintenance starts already before the delivery of a software system. The standard ISO/IEC 14764: Standard for Software Engineering – Software Life Cycle Processes – Maintenance [59] gives the following definition for software maintenance:

“the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage.”

2.1.2 Software Maintenance Categories

Software maintenance was broadly researched with the goal of identifying the reasons for changes made to software systems, to investigate their relative frequencies and estimate their costs. The result of this research comprises classifications of maintenance activities which help to understand its significance and its relation to the costs and quality of software systems in use.

First proposed by Lientz and Swanson in 1980 [76], maintenance today is distributed into different categories. The number of software maintenance categories changed since their first use in literature, and so did their description. Today the commonly accepted categories of software maintenance are defined in the ISO/IEC 14764 standard:

- **Corrective maintenance**: the reactive modification of a software product performed after delivery to correct discovered problems.

- **Adaptive maintenance**: the modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment.
2.1.3 Software Maintenance Processes

In literature there exist several proposals of process models for software maintenance. These models organize software maintenance into a sequence of related activities, or phases, and define the order in which these phases are to be executed. Sometimes they additionally suggest the deliverables that each phase must provide to the following phases. The commonly accepted standards for software maintenance processes today are ISO/IEC 12207: Software Life Cycle Processes [58] and ISO/IEC 14764.

The standard ISO/IEC 12207 deals with the totality of the processes comprised in the software life cycle. The standard identifies seventeen processes grouped into three broad classes: primary, supporting, and organizational processes. Processes are divided into constituent activities each of which is further organized in tasks. Figure 2.1 shows the activities of the ISO/IEC 12207 maintenance process.

ISO/IEC 14764 describes in greater detail management of the maintenance process described in ISO/IEC 12207. It also establishes definitions for the various types of maintenance. It provides guidance that applies to planning, execution and control, review and evaluation, and closure of the maintenance process.

Figure 2.1: The ISO/IEC 12207 maintenance process [58].
2.1.4 Debugging and Patch Generation

An important activity of software maintenance is the process of dealing with faults that have occurred in the system once deployed. From a technical point of view, this activity comprises debugging and patch generation. First faults, bugs, errors, crashes are defined in [96] as the following:

Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result.

Fault: An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program.

Failure: An incorrect result. For example, a computed result of 12 when the correct result is 10.

Mistake: A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator.

It is important to note that also according to [96], a “bug” refers to an error or a fault and all of the above terms are often used interchangeably. [96] also provides definitions for debugging and patch:

- Debugging is the action of “detecting, locating, and correcting faults in a computer program.”.

- A patch can be one of the following:
  1. A modification made directly to an object program without reassembling or recompiling from the source program.
  2. A modification made to a source program as a last-minute fix or afterthought.
  3. Any modification to a source or object program.
  4. The action of performing a modification as in (1), (2), or (3).

Debugging can be performed during development, at the testing phase, or when software is in production, after deployment. In the later case, debugging becomes part of the maintenance process. According to the definitions above, correcting a fault is part of the debugging process and generating a patch can be seen as corrections that are performed on the system source code. The debugging and patch generation activities are quite heavy in terms of time and cost. Some reasons for this are that these processes still rely on human intervention and expertise.
Debugging in general is a very challenging task. Quoting [46]: “Debugging in production or deployment is very complicated. Short of using some advanced problem determination techniques for locating the specific defect or deficiency that led to the unexpected behavior, this debugging can be painful, time consuming and very expensive. When this involves multiple software products and critical interdependencies, with no easy way to narrow down location of the offending code, this is a real nightmare. As the debugging moves away from the source code, it becomes even more manual and time consuming.”

Part of the challenge of debugging is that this is an activity that still relies very much on human expertise and manual processes. As explained in [43], debugging a program usually follows the same principles. First, when an error is reported, the developer aims to reproduce the error attaching a debugger to the running program. This debugger allows to access information on the program states and determine where the execution deviates from the expected program behavior. This helps the developer locate the bug. The programmer then needs to fix the code in order to remove the bug and then provide an updated program to the user.

Therefore in the post-release maintenance stage, traditionally bug fixing starts with the submission of a bug report. Once the bug report is submitted and before it is fixed it goes through several stages. This whole process starts with bug triaging. Bug triaging in most of the cases is still performed manually and is a big research area [126]. In this process a bug is checked for duplication, correctness and then its assigned to a suitable developer. Developers manually identify duplicate bug reports, but this identification process is time-consuming and exacerbates the already high cost of software maintenance [61]. Although there has been a lot of work in automating this process, barely any solution is available with proper tool support which also considers later complexities involved in this process. Some of the existing research work has also proposed to merge duplicate bugs in order to provide more contextual information to a maintenance engineer rather than completely discarding the duplicate bug [12].

The next step in the bug fixing process involves assigning a valid bug to a suitable developer. This particular topic has also drawn attention of many researchers to come up with new approaches for identifying a suitable developer for fixing a given bug. In most of the approaches finding a suitable developer considers two main criteria i.e. expertise and workload. There are several approaches which consider textual information extracted from various artifacts and then apply various heuristics from natural language processing and ontologies while many approaches consider structural information and machine learning techniques [8] [7][48].

Once the bug has been assigned to a suitable developer he tries to understand the exact nature of bug by going through all the contextual information he can get from the bug report. Sometimes the submitted bug report does not contain enough information to help
the developer localize the scope of problem and to identify a potential solution. This has led a large group of researchers to assist this process by mining various development artifacts and provide as much contextual information as possible by recommending related software objects and artifacts [5]. Now with all the available information, the developer tries to find and fix a bug using various debugging tools available widely. One of the most advanced tools analyzes the debugging situation at a statement level and reports a prioritized list of relevant bug-fix suggestions [62]. It relies on machine learning techniques to automatically learn from new debugging situations and bug fixes over time while taking into account the static and dynamic structure of a statement.

The bug fixing process is a bit different prior to the release of actual software as in this stage mostly bug submitters are developers and testers only. During this phase finding and fixing a bug is usually much more faster and easier as submitters usually tend to provide much more contextual information in bug reports due to their familiarity with the system compared to normal users. A major challenge at this stage is finding the bug itself, which in some part can be assisted by tools to detect bug patterns in code during the development phase itself [54].

In [77], the authors consider security patches and describe patch generation as a “critical phase in the life-cycle of a software vulnerability”. They also explain that although quick patch generation and release is required, it is in practice quite a lengthy process. They cite as an example, an analysis that was conducted on 10 recent Microsoft patches (MS06-045 to MS06-054) and that shows that it took 75 days on average to generate and release patches once the vulnerability has been identified.

[105] presents a study on developers of the Eclipse project. This study shows that stack traces are often part of bug reports and also that code changes in order to fix bugs are related to information contained in stack traces. This indicates that stack traces represent a useful means for developers to tackle bugs. The types of bugs that can be encountered can be quite diverse, but also seem to follow some patterns. In [107], the authors studied the fixes applied to 7 open source projects written in Java: Eclipse, Columba, JEdit, Scarab, ArgoUML, Lucene, and MegaMek. The authors found that the most common fix patterns are 1) change the expression or value of method calls parameters (14.9% to 25.5% depending on the project); 2) change of IF conditions (5.6% to 18.6%); and 3) change of assignment expressions (6% to 14.2%). These results indicates the type of bugs that are usually introduced into source code: wrong parameter expression in method calls, wrong IF condition, and wrong assignment expression. These bugs show a mismatch between the developer’s intent and the source code, or else it shows a misunderstanding of the systems requirements from developers. In [69], the authors study the impact of the initial developer’s intent on fixing concurrency bugs in Java. Typical fixes for concurrency bugs are race-freedom, atomicity and refinement. The authors show that
without any indication of the developer’s intent (e.g. atomicity or refinement annotations for code blocks), debugging tools miss an important piece of information for identifying root causes and fixing the bugs. It makes the debugging task difficult and even makes the developer choose a fix that may resolve the occurrences of symptoms associated to the bug but that may impact to the system in an unacceptable way (e.g. introducing new performance issues).

Facing such issues, approaches to help developers identify the cause of a bug have been developed in order to reduce the debugging process. These approaches usually rely on some automation. As explained in [43], in case of large systems deployed over numerous machines and interacting with possibly several servers, the classical debugging approach is no more applicable. For this reason, the Microsoft teams have developed automatic collection tools, which gather information about the system at runtime, as well as automatic diagnosis tools which help identify the root cause of an error. Such tools were connected in order to create the Windows Error Reporting (WER) system which relies on previously identified bugs in order to automatically generate bug reports and diagnosis to the developers.

Algorithmic debugging (see e.g. [19]) is such an approach. It is a semi-automatic debugging technique which proposes to the developer some questions in relation to a sequence that leads to a failure. The debugging system uses the answers provided by the developer in order to help the developer identify the specifics of this sequences that characterize the observed fault.

From this observation, IBM initiated research in Autonomic Computing, which main aim is to reduce time and cost of the maintenance process by automating it. Self-healing is a subpart of Autonomic Computing and corresponds to techniques for bringing back to normal states a system whose behavior deviated to an improper state. This broad definition includes the case of a system which encountered a fault and that needs to recover from it and be brought back to a state from which normal behaviors can be executed again.

## 2.2 Autonomic/Self-Managing Systems

Autonomic computing (AC) is a concept to describe computing systems that are intrinsically intended to reduce complexity through automation. Today, many researchers agree on the fact that AC provides the set of capabilities required to make computing systems self-managing via inherent self-adaptation. Typically, adaptive systems are contingent upon varying environments where they can reason about the environment and adapt to changing situations [42]. In that context, AC emphasizes adaptive features and other self-managing ones to provide solutions to one or more real world problems in one or
more domains spanning intra or inter-domain applications.

The “Vision of Autonomic Computing” [68] defines the concept of self-management as comprising four basic policies - self-configuring, self-healing, self-optimizing, and self-protecting [56, 68]. In addition, in order to achieve these self-managing objectives, an autonomic system (AS) must constitute the following self-* properties (see e.g. [102, 1]): 1) self-awareness – aware of its internal state; 2) self-situation (context awareness) – environment awareness, situation and context awareness; 3) self-monitoring – able to monitor its internal components; 4) self-adjusting (self-adaptiveness) – able to adapt to the changes that may occur. Thus, despite their differences in terms of application domain and functionality, all ASs are capable of self-management and are driven by one or more self-management objectives. Note that this requirement automatically involves 1) self-diagnosis (to analyze a problem situation and to determine a diagnosis), and 2) self-adaptation (to repair the discovered faults). The ability to perform adequate self-diagnosis depends largely on the quality and quantity of system knowledge. Moreover, AC recognizes two modes of self-healing - reactive and proactive. In reactive mode, an AS must effectively recover when a fault occurs, identify that fault, and when possible repair the same. In proactive mode, an AS monitors vital signs to predict and avoid health problems or reach undesirable risk levels.

2.2.1 Autonomic Elements

A key concept in the AC paradigm is the so-called autonomic element (AE). A widely accepted architecture for autonomic systems considers AEs as the system’s building blocks [56]. By its nature, an AE extends programming elements (i.e., objects, components, services) to define a self-contained piece of software with specified interfaces and explicit context dependencies. Essentially, an AE encapsulates rules, constraints and mechanisms for self-management, and can dynamically interact with other AEs of the AS in question to provide or consume computational services. As stated in [56], the basic structure of an AE is a special control loop. The latter is described as a set of functionally related units: monitor, analyzer, planner, and executor, where all share knowledge. By sharing knowledge, those units form an intelligent control loop that forms the self-managing behavior of the AE in question. A closer look at the generic structure of an AE is presented by Figure 2.2, which is borrowed from [92]. As depicted, an AE operates over a special managed resource. The latter is a generic presentation of software that can be managed by the control loop in order to leverage its functionality to a self-managing level. Here, through its control loop, an AE monitors the managed resource details, analyzes those details, plans adjustments, and executes the planned adjustments. It is important to mention that for these activities, an AE can use information from humans (administrators) as well
as rules and policies defined (by humans) and learned by the AS [56]. Thus, a control loop helps an AE make decisions and controls the managed resource through monitoring and intensive interaction. Note that the managed resource is highly scalable [56], i.e., it can be any software system such as a file, a server, a database, a network, a middleware software, a standalone software application, a business object, etc.

A key factor in the success of AE self-management is the shared solution knowledge. Note that the lack of solution knowledge is a drawback for self-managing. For example, today there are a number of maintenance mechanisms for installation, configuration, and maintenance. The vast diversity in system administration tools and distribution packaging formats creates unexpected problems in the administration of complex systems. In addition, the same problems are leveraged by the use of dynamic environments where application functionality can be added and removed dynamically. Here, common solution knowledge removes the complexity introduced by diversity in formats and installation tools. Moreover, when acquired in a consistent way, this very knowledge is used by the AE’s control loop in contexts different than configuration and maintenance, such as problem formation and optimization.

### 2.2.2 Architecture-Based Self-Healing

To develop ASs capable of self-healing, there are two distinct elements that must be taken into consideration. First we need an AE to be present to make the decision of when and how to perform a repair on a target system, i.e., a system to which the AE brings autonomic facilities. Second, an infrastructure for executing the repair strategy must be available to that AE. This infrastructure is provided by the AE’s control loop...

(see Section 2.2.1). The AE control loop typically maintains comprehensive knowledge of one or more explicit models of the target system and uses those models as a basis for configuring, repairing, and optimizing that system. A branch of the AC research suggests an architectural model of the software as a useful basis for dynamically changing the target system [30]. An architectural model can provide a global system perspective, expose important properties and constraints, and support problem analysis. It therefore allows adaptation to be done in a principled and possibly automated fashion. Using the architectural model as a basis to monitor, adapt and heal a target system is known as architecture-based self-adaptation (or architecture-based self-healing).

2.2.3 Biological Inspiration

Inspiration from biology necessitated paradigms related to ASs. The mechanisms behind the principles of self-management and self-governance are inspired by the human body’s Autonomic Nervous System (ANS). In fact, this is the focus of AC. The idea is that mechanisms that are autonomic, in-built, and requiring no conscious thought in the human body are used as inspiration for building mechanisms that will enable a computer system to become self-managing (and thus self-adapting) [52]. Hence, the general properties necessitated from ANS for an AS can be summarized by the four AC self-managing objectives: being self-configuring, self-healing, self-optimizing and self-protecting. Note that the self-managing objectives cannot be considered independently. Similar to the complex behavior observed in the human body, an AS adapts to changes by following encoded objectives. For example, if a malicious attack is successful, this will necessitate self-healing actions, and a mix of self-configuration and self-optimization actions, in the first instance to ensure dependability and continued operation of the system, and later to increase self-protection against similar future attacks [50]. Moreover, being computing software, an AS goes even further by ensuring that autonomic features add minimal impact over the system performance, thus avoiding significant delays in processing.

Colonies of ants or other insects, flocks of birds, swarm of bees, herds of animals, schools of fish, etc., also inspired approaches to building ASs, such as evolutionary systems or highly-efficient and highly-complex distributed systems consisting from large numbers of largely homogeneous components. Insects have tremendous diversity in color, shape and size and their behavior is interesting to observe. For example, the so-called emergent behavior [13]inspired a special class of self-managing systems called “intelligent swarms”, e.g., the NASA’s ANTS (Autonomous Nano-Technology Swarm) prospective mission [124]. This behavior helps social insect colonies to collectively solve complex problems without centralized control. Thus, colony (or swarm) behavior appears out of local interactions between individuals with simple rule sets and no global knowledge.
2.2.4 Current State

Since 2001, AC has emerged as a strategic initiative for computer science and the IT industry [92]. Certain progress has been made on a number of fronts to help make the vision of AC real. However, the most visible progress of AC today is the implication of self-managing autonomic features (embedded self-managing features) into individual software products. The following elements describe a few examples of such progress [92].

- Chips can now sense change and alter the configuration of circuitry to enhance processor performance or avoid potential problems.
- Databases can automatically tune themselves as workload fluctuates and optimize performance as data organization changes.
- Networking components can intelligently route traffic.
- Blade servers [91] can automatically populate new blades with the required software as they’re plugged in.

Although facing such a successful implication of AC features in real software products, the world is still far from the full realization of the AC vision [87]. The reported success is only one aspect of the self-managing autonomic infrastructure. Thus, regardless of the autonomic capabilities built into individual software products, today the vision of AC is still not realized.

2.2.5 Research Venues

To date, many research and development initiatives have arisen for developing technologies and frameworks for ASs.

2.2.5.1 Grid Computing

Grid computing relies on self-managing features to achieve two distinct but related goals: 1) to provide remote access to IT infrastructures; and 2) to aggregate computational resources. The general idea behind grid computing [39] is to make efficient use of computational resources with the power of distributed computing. A computing grid is conceptually like an electrical grid, but composed of computers. All the computers within a computing grid form a computational structure by coupling wide-area distributed resources. These resources are considerably scalable and can be databases, storage servers, high-speed networks, supercomputers, etc. The key concept of the grid computing is the transparent access to the computational resources in the grid irrespective of their physical source. Thus, in order to ensure resource transparency, grid computing introduces a
special self-managing middleware that is used to coordinate disparate IT resources across a grid network. As a result, these resources are transparently exposed to the final user as a virtual whole.

2.2.5.2 Intelligent Swarms

Artificial Intelligence (AI) emerged over 50 years ago when Alan Turing introduced his prototype for intelligent machines. Here, for over five decades, AI research has gone over tremendous evolution conceiving research fields such as natural language processing (including speech recognition and speech generation), data mining, machine learning, automated reasoning, neural networks, genetic algorithms, fuzzy logic, etc. Intelligent agents provide both concept and technologies product of this very evolution. Today, intelligent agents are considered one of the key concepts that could help us realize self-managing systems. This is the reason why a great deal of research effort is devoted to developing the intelligent agent technology, which has become a rapidly growing area of research and new application development. An extension of the “intelligent agent” paradigm is the so-called multi-agent systems, where many intelligent agents interact with each other [117]. These agents are considered to be autonomous entities that interact either cooperatively or non-cooperatively (on a selfish base).

Intelligent swarm systems are multi-agent systems based on biologically-inspired concepts (see Section 2.2.3). By their nature, such systems are complex multi-agent systems where the individual members of the swarm imply independent intelligence. A good example of an intelligent swarm system is the NASA ANTS (Autonomous Nano-Technology Swarm) mission, a swarm-based exploration mission representing a new class of concept exploration missions based on the cooperative nature of hive cultures [124]. A mission of this class necessitates a self-managing software system, comprising a set of autonomous mobile units exhibiting self-managing properties. Note that a swarm-based system offers many advantages compared with the single-spaceship system, like greater redundancy, reduced costs and risks, and the ability to explore regions of space where a single large spacecraft would be impractical.

2.2.5.3 Sensor Networks

A sensor network [28] is a network consisting of interconnected autonomous intelligent devices equipped with sensors to provide cooperative monitoring. Although equipped with intelligent software, each individual network node of a SN is inherently resource constrained in terms of limited processing speed, storage capacity, and communication bandwidth. As a result, the network nodes have substantial joint processing capability and not so-significant individual computational power. Sensor networks operate over
sensors collecting and processing data in diverse domains such as air quality control, weather forecast, traffic control, security and surveillance applications, etc. In most SN applications, a network is intended for a long-term operation and the SN nodes are wireless and must provide self-managing features to some extent. For example, a SN node must operate in an optimal energy-saving mode, i.e., it must monitor its energy resources and find the optimal tradeoff between performance and long-term operation. Here, to minimize energy consumption, most of the node’s hardware components (controlled by the self-managing piece of software), such as radio are likely to be turned off most of the time. Note that these factors make the networking protocol highly complex, which necessitates self-managing capabilities to tackle self-organizing and self-healing problems.

### 2.2.5.4 Web Services

Web services (WSs) share some principles with self-managing systems characterized by the concept of program-to-program interactions [65]. By its nature, a web service is an abstraction of a collection of operations that are network-accessible. Each web service is presented in the network of WSs with a special service description. This description provides all the details necessary to make the interaction with a service possible, such as message formats, transport protocols, service location, etc. WSs imply the so-called service-oriented architecture (SOA). Initially, SOA sets forth three service roles and three service operations:

- **roles** - service provider, service requester, and service registry;
- **operations** - publish, find, and bind.

One of the key issues for a broad range of WSs, such as admission control, server selection, scheduling, pricing, and specifying service-level agreements, is the quality of service (QoS). The problem is that the QoS delivered to a customer is highly affected by various factors, e.g., performance of the web service itself, performance of the service provider (hosting platform), and performance of the underlying network. Thus, in order to achieve a high level of QoS, the software behind WS must incorporate autonomic features (the emphasis is on self-adapting and self-healing), which can deal with the issues of service publishing, service discovery, and especially with the issue of service selection.

A solution to the problem of self-managing WSs is the Web Service Level Agreement (WSLA) framework developed at IBM (see http://www.research.ibm.com/wsla/). WSLA is a framework for specifying, creating, and monitoring special service level agreements for web services. Note that WSLA complements various other WS-related software approaches addressing issues on proactive management of a web service environment, e.g., provisioning resources, workload management, etc.
2.2.6 AC Formalisms, Initiatives, and Frameworks

The idea of self-managing systems has inspired a great deal of research effort in both applied and fundamental science and today we have a number of leading IT companies and universities, which started initiatives on research and development of self-managing systems.

2.2.6.1 AC Formalisms

Formal methods have proven to be a valuable approach to the development of ASs. The advantage is that AC formal methods provide developers with special formal notations (AC formalisms) that help developers build AS specifications that serve as a basis for further design, implementation, and verification of ASs. Formalisms dedicated to AC have been tackled by a variety of industrial and university projects. IBM Research developed a framework called Policy Management for Autonomic Computing (PMAC) [57]. The PMAC formalism emphasizes the specification of self-management policies encompassing the scope under which those policies are applicable. A PMAC policy specification includes: 1) conditions to which a policy is in conformance (or not); 2) a set of resulting actions; 3) goals; and 4) decisions that need to be taken.

The so-called Specification and Description Language (SDL) is an object-oriented, formal language defined by the International Telecommunications Union – Telecommunications Standardization Sector (ITU-T) [24]. SDL is dedicated to real-time systems, distributed systems, and generic event-driven systems. The basic theoretical model of an SDL system consists of a set of extended finite state machines, running in parallel and communicating via discrete signals, thus making SDL suitable for the specification of self-management behavior.

Cheng et al. talk in [101] about a specification language for self-adaptation based on the ontology from system administration tasks and built over the underlying formalism of utility theory [98]. In this formalism, special self-adaptation actions are described as architectural operators, which are provided by the architectural style of the target system. A script of actions corresponds to a sequence of architectural operators. This sequence forms the so-called adaptation tactic defined in three parts: 1) the conditions of applicability; 2) a sequence of actions; and 3) a set of intended effects after execution.

Another formalism for ASs is provided by the so-called chemical programming, based on the Gamma Formalism [60], which uses the chemical reaction metaphor to express the coordination of computations. The Gama Formalism describes computation in terms of chemical reactions (described as rules) in solutions (described as multi-sets of elements). When applied to AS specification, the Gama Formalism captures the intuition of a collection of cooperative components that evolve freely according to some predefined constraints.

Figure 2.3: ASSL Multi-tier Specification Model [120].

(rules). System self-management arises as a result of interactions between components, in the same way as “intelligence” emerges from cooperation in colonies of biological agents. [60] presents a biologically inspired formalism for AC called Higher-Order Graph Calculus (HOGC). This approach extends the Gama Formalism with high-level features by considering a graph structure for the molecules and permitting control on computations to combine rule applications. HOGC borrows various concepts from graph theory, in particular from graph transformations, and use representations for graphs that have been already intensively formalized.

**ASSL Formalism.** ASSL (Autonomic System Specification Language) [120] is a declarative specification language and framework for ASs with well-defined semantics. It implements modern programming language concepts and constructs like *inheritance, modularity, type system,* and high *abstract expressiveness.* Conceptually, ASSL is defined through *formalization tiers.* Over these tiers, ASSL provides a scalable multi-tier specification model that exposes a judicious selection and configuration of infrastructure elements needed to specify an AS (see Figure 2.3). ASSL defines ASs with special *self-managing policies, interaction protocols,* and *AEs (autonomic elements).* Note that each tier is intended to describe different aspects of the system in question, such as *service-level objectives, policies, interaction protocols, events, actions,* etc. This helps to specify at different levels of abstraction imposed by the ASSL tiers.

ASSL has been successfully used at Lero (The Irish Software Engineering Research Centre - see www.lero.ie) to model space-exploration systems, e.g., the NASA’s Voyager
Mission [122] and the NASA’s ANTS prospective mission [121], in a stepwise manner (feature by feature) and generate a series of prototypes that are evaluated under simulated conditions. The approach allows different prototypes to be tried and tested (and benchmarked as well), get valuable feedback before implementing the real systems and discover eventual design flaws in both the original system and the prototype models. For example, the ASSL prototypes for self-healing in ANTS [121] target recovering from failures, including those caused by damage due to a crash or any outside force. The autonomic features assume that each spacecraft sends, on a regular basis, heartbeat messages that can be used to determine when a unit is not able to continue its operation, due to a crash or a malfunction in its communication device. Moreover, a spacecraft unit sends a notification message if its instrument is malfunctioning or it has been broken, due to a collision with an asteroid or another spacecraft. In similar projects, ASSL has been used to model home automation systems based on wireless sensor networks [36] and pattern-recognition systems [123]. In all cases, ASSL policies such as self-healing are specified at both system (AS tier) and autonomic element (AE tier) levels (see Figure 2.3), where events, actions, metrics, and special managed element interface functions are used to incorporate the AC behavior (e.g., self-healing).

One of the biggest challenges AC is facing today is how to model policies (objectives) that may evolve to satisfy system goals that may change in the course of system adaptation. Thus, the problem is how to address uncertainty in the self-* objectives and consecutively facilitate the development of ASs that are more flexible to adaptation. One of the possible solutions is coming with the new version of ASSL where developers will be able to specify special loose self-managing policies, i.e., evolving policies that determine points of flexibility in their behavior.

2.2.6.2 AC Initiatives

IBM Tivoli Management Suite. Tivoli Management software [87] is an IBM initiative devoted to AC and intended to develop solutions towards fulfilling the ultimate goal of having fully ASs. Many of the Tivoli products have some level of automation that can help achieve the benefits of an AC environment. The Tivoli applications support and incorporate one or more of the core autonomic objectives including self-healing. Figure 2.4 illustrates the self-managing features of these applications.

The IBM Tivoli self-managing software is based on a special Tivoli enterprise architecture, built as a distributed object-oriented Tivoli framework. In fact, most of the Tivoli applications incorporate services included in this framework. In addition to the framework, Tivoli Enterprise provides products for development and management of ASs in terms of deployment, availability, and security. For example, a part of the Tivoli initiative is a framework called policy management for autonomic computing (PMAC) [55]. This

Figure 2.4: Self-managing Features in Tivoli Software [87].

framework provides a standard model for the definition of policies and an environment for the development of software objects that hold and evaluate self-managing policies. In general, PMAC is used for development and management of intelligent autonomic software agents. With PMAC, these agents incorporate the ability to change dynamically their behavior. This is provided by a formal specification of policies encompassing the scope under which these policies are applicable. Here, a PMAC policy specification includes:

- conditions to which a policy is in conformance (or not);
- a set of resulting actions;
- goals;
- decisions that need to be taken.

In this framework, the PMAC agents are possibly able to determine the relative value of multiple applicable actions, goals, and decisions in order to change dynamically their behavior. In order to write and store policies, PMAC uses a declarative XML-based language called Autonomic Computing Policy Language (ACPL) [55].

The IBM Tivoli Configuration Manager provides AC developers with a self-healing mechanism to repair software packages that were already distributed to target machines [119]. Such a mechanism requires a special activity plan that depicts the activities required to properly update the target set of machines for a specific software package. The solution automatically launches the plan associated to each package for the required workstations. In order to automate the software distribution process, the involved machines should be assigned with roles representing the specific function of a target machine, e.g., Legal
Division, Manufacturing, File Server, and Web Server. Each target machine requires a set of packages that will be grouped by role as well. Further, it is required to assign the relationships (or dependencies) among packages already installed on each workstation and to assign an activity plan to each package. Combining all these pieces of information gives a means of an Automated Software Distribution mechanism relying on software distribution and inventory components, as shown in Figure 2.5.

Hewlett-Packard Adaptive Enterprise. Hewlett-Packard’s initiatives for self-managing software is called “Adaptive Enterprise” [49]. The latter is a framework based on a special architecture termed “Darwin Reference Architecture”. In general, this framework is business oriented and targets at self-managing solutions to help businesses create a tighter connection between the system management and the business process. The Darwin Architecture provides a set of design rules and guidelines to help organizations become more agile (see Figure 2.6). The main design principles imposed by the architecture are simplification, standardization, modularity and integration. These principles are necessitated due to fundamental elements of the Darwin Architecture such as service-oriented architectures (SOAs), model-driven automation, and virtualized resources.

A key role in the Darwin Architecture is the Manage and Control Software nested in IT Business management vertical dimension (see Figure 2.6). This software incorporates self-managing features and coordinates and controls the enterprise infrastructure. For this, the whole environment is a subject of continuous inventory, monitoring, planning, provisioning, control and automatic maintenance. Here, due to these operations, self-managing features are propagated over the entire infrastructure. Some of these are presented by the following paragraphs:

- **Self-configuring.** The Adaptive Enterprise framework establishes an IT environment
that can quickly adapt to dynamic changes, “mergers and acquisitions”, and integration of new business rules. For example, the special HP Blade System Infrastructure allows for automated, policy-based deployment.

- **Self-healing.** The same HP Blade System Infrastructure also provides automatic failure detection and protection, and advanced workload management across the entire enterprise infrastructure. Here, to help, HP includes a special HP Blade System Management Suite that helps in the integration, virtualization, and automation of solutions.

- **Self-optimizing.** The Adaptive Enterprise framework strives to self-optimize resource utilization. The following elements present a few examples of self-optimization through virtualization:
  - utilization of single or multiple server environments is optimized through their configuration as virtual pools of resources;
  - a special storage architecture abstracts the physical storage by providing improved utilization capacity and data availability;
  - mechanisms are provided (e.g., HP Open Call) to dynamically allocate resources over networks.

It is important to mention that the Adaptive Enterprise strategy for resource virtualization includes three main levels of virtualization:

- **Element virtualization (first level)** - single resources are optimized to fulfill requirements within a single application. Here resources can be servers, storage devices, network, software, etc.
• Integrated virtualization (second level) - multiple resources are optimized within a single application to automatically fulfill service-level objectives.

• Compete IT utility (third level) - all resources are pooled and shared over applications to automatically fulfill demand in real time.

Microsoft Dynamic Systems Initiative. The Dynamic System Initiative (DSI) is a research program started at Microsoft aiming at developing special dynamic systems incorporating self-managing features [25]. Note that DSI is an initiative claiming to provide a complete business solution where dynamic systems help to increase the agility and dynamic capability of IT business. In general, DSI recognizes three principle elements as the foundation for building such systems:

1. Design for operations - provides special system models to capture and embed within the IT infrastructure the needed knowledge of diverse people such as business architects and managers, application developers, IT professionals, and industry partners.

2. Knowledge-driven management - provides techniques and tools to capture system states concerned with both configuration and health relevant to business priorities and demands. Further this inherent knowledge is used to provide a level of self-management to the system in question.

3. Virtualized infrastructure – provides a virtual pool of services that consolidate system resources and thus, increase agility of the existing infrastructure. An advantage is that the system in question is allowed to efficiently operate with the needed resources.

Microsoft dynamic system architecture is based on a special System Definition Model (SDM). In general, SDM is a model that is used to create definitions of distributed system, i.e., definitions of resources, endpoints, relationships and sub-systems. In addition, it contains deployment information, installation process, configuration schemas, events, automation tasks, health models and operational policies. Moreover, the SDM approach is systems to be designed with operation in mind, where an integrated feedback loop can facilitate improvements in the system itself. The DSI approach helped Microsoft to implement the following self-managing features in the Windows Vista operating system [25]:

• faster and more precise configuration through a special imaging deployment;

• built-in ability to diagnose configuration issues and seek out solutions on the Web;

• image-based deployment for better support of virtualized environments;
advanced automation scripting (through the Windows PowerShell tool);

• native support for industry management standards including the so-called common information model (CIM) for instrumentation, support for modeling local and distributed systems, and support for cross platform interoperability.

Another successful DSI product is Microsoft Exchange, fully delivered in 2008. Due to its “dynamic” nature, Microsoft Exchange is able to maintain high levels of performance and availability, and adapt and expand its structure to meet changing business needs. Moreover, freely available configuration models can be used as plug-ins by the Exchange Server to create baseline configurations by combining these models with corporate requirements for security and compliance. In addition, modern Microsoft Windows systems incorporate self-managing features in the form of special windows-embedded tools, some of which we present here:

• Microsoft operation management (MOM) incorporates event management, proactive monitoring and alerting, reporting and trend analysis, and system and application specific knowledge to improve the manageability.

• Corporate error reporting (CER) tool which is a mechanism to provide information about problems in applications to the vendor or the in-house application developer.

• Windows management instrumentation (WMI) provides system administrators with direct and unified management tools.

• Software update services automatically deliver critical patches download from the Interne.

• Virtual disk service provides a vendor independent interface for identifying and configuring storage devices from multiple vendors.

Sun N1. N1 is an initiative at Sun Microsystems for self-managing software systems. The acronym “N1” stands for managing “n” number of computers as “1” [86]. By its nature, N1 is intended to provide intelligent services for managing heterogeneous environments. The N1 initiative has inspired a family of products based on the Sun’s Grid Engine technology leveraged with enterprise-level capabilities and equipped with a powerful system administration tool (called Sun Management Center). An example is the Cluster Grid, which implies a three-tier general architecture comprising the following tiers:

• The access tier provides access and authentication to grid users. Access methods are based on special authentication schemes such as NIS, LDAP, and Kerberos [86].
The management tier includes one or more servers, which run the server part of self-managing software to perform distributed resource management, hardware diagnosis, and system performance monitoring.

The compute tier includes servers that run the client part of self-managing software incorporating special daemons associated with message passing environments, and agents for system health monitoring. The compute tier communicates with the management tier, receiving jobs to run and reporting job status and accounting details.

The Sun Management Center management tool is based on a special intelligent agent reference model. In this approach, a manager monitors and controls managed entities by sending requests to agents residing on the managed node and agents collect management data on behalf of the manager. Moreover, the Sun Management Center tool relies on a special autonomous agent technology to drive special autonomous capabilities. The autonomous agent technology is a technique in which agents are not dependent on other software components and all data collection and processing is done locally at the agent side. These intelligent agents can act on data to initiate alarms, to issue administrative notifications or to perform specific actions.

2.2.6.3 AC Frameworks

Rainbow. Rainbow is an AC framework that provides mechanisms to monitor system and environment states, manage and use an architectural model to detect problems, determine problem state and decide on a course of action, and act on the adaptation, corresponding to the AE’s control loop (see Section 2.2.1).

Figure 2.7 shows the Rainbow self-adaptation framework [21] functions as follows. Monitoring mechanisms, consisting of various kinds of probes, observe the target system. Gauges relate observations to properties in the architecture model managed by a Model Manager. When new updates occur, a special Architecture Evaluator checks the model to ensure that the system is operating within an envelope of acceptable range, determined by constraints on the architecture. If the evaluation determines that the system is not operating within an acceptable range, another framework component called Adaptation Engine is triggered to determine what appropriate adaptation action to take. Finally, an Executor carries out the chosen action on the target system via system-level effectors.

Rainbow targets at a special decision-making mechanism where are considered the mental model, knowledge, and cognitive tasks of the system administrators in keeping a system operational. This helps to emulate the decision-making process as well as formulate a set of first-class concepts important to self-adaptation and self-healing. For example, once a problem has been identified, Rainbow must determine which repair action to
undertake. When a system administrator determines a course of action, her decision is based on her knowledge of and her experience with the target system. After she chooses an adaptation strategy, she may change her mind based on changes in system conditions. Moreover, she may perform more detailed inquiries into particular states of the system before committing to a strategy, such as deciding whether a sudden increase in network traffic is due to legitimate requests or a denial-of-service attack.

In Rainbow, this process consists of selecting a self-healing strategy that best meets the current conditions and objectives of the system. To emulate the system administrator’s decision process, all of the considerations described above are taken by Rainbow as uncertainties. In particular, the cost and effect of actions are uncertain, conditions of applicability are uncertain, and outcomes of actions are uncertain. Therefore, Rainbow must select a strategy while accounting for all of these uncertainties up-front.
3 Fault Tolerance and Self-Healing

3.1 Fault Tolerance

Fault-tolerance is an approach to obtaining dependable software. It can be defined as the mechanisms and techniques that enable a system to deliver its specified service despite the presence of faults [11]. If we consider a more strict definition of fault-tolerance such as [42] and assume that a fault-tolerant program must be able to continue to operate in its entirety after a fault, we may say that the goal of the fastFIX project is more in the domain of “survivable systems”: systems that continue to operate despite the presence of faults. fastFIX’s goal is not to create mechanisms for applications to withstand faults without external support. The ultimate goal of this project is to self-heal or patch applications and, in that sense, fastFIX programs need only be fault-tolerant or survivable from the moment that a fault is detected up until the arrival of corrective measures from the application’s developers. This section describes fault-tolerance concepts and techniques that are essential to understand and contextualize the fastFIX project and its approach to self-healing and survivability.

Software Maintenance and Fault Tolerance

Fault-tolerance considerations permeate all the phases of a modern and robust software lifecycle: from design, to implementation choices, to testing, to auxiliary platform mechanisms for error detection and reporting.

fastFIX focuses on the maintenance aspects of the software lifecycle and therefore the fault-tolerance considerations described here, are restricted to the execution and maintenance phases of the software lifecycle.

More specifically, fastFIX aims to handle faults by i) detecting their presence (ideally before their occurrence); ii) if possible preventing their manifestation, by enforcing fault containment; iii) reporting the detected faults; iv) and patching/healing/neutralizing the fault’s source.

Fault Detection

fastFIX is aimed at single version software. As stated before, our goal is not to create a fault tolerant framework. Therefore, there are several fault-tolerance techniques that cannot be contemplated in this context, such as replicating application state [44] or using multi-version programs with several replicas of the running application or even different
implementation of a design integrated by a voting mechanism.

Detection of software faults can be done before their occurrence during the execution of a program. This type of detection may be embedded in the application itself or may require an error detection mechanism concurrent to the main execution flow of the program. This type of preventive fault detection requires almost always modifications to the application.

Applications can be designed or modified to prevent or detect several types of faults \textit{a priori} [4]:

- **Timing issues**: In the cases where applications are expected to make progress in bounded time, e.g. consecutive points of the code should be reached within a certain time interval, watchdog timers can be used to detect faults such as fail-stop errors in components, deadlocks or livelocks.

- **Coding checks**: Such as verifying the relation between data and integrity metadata, e.g. checksums, can guarantee that data structures do not enter invalid states.

- **Invariant checks**: Another popular fault-tolerance addition to source code is the verification of logical invariants mostly expressed as pre- and post-conditions to be verified before and after the manipulation of data structures.

- **Structural checks**: A particular case of invariant verification is the addition of code to check whether operations that violate structural characteristics of the program’s data structures such as buffer overflows, stack underflows or violations of type restrictions.

Software faults may also be detected after their occurrence. Detection of faults \textit{a posteriori} is normally done by the operating system or executing environment. The most common external symptoms of software faults are exceptions and runtime errors. Exceptions correspond to abnormal application operations and are usually handled within the application. If not, the execution environment of the application will intercept exceptions thrown by the application. Runtime errors are provided as standard error detection mechanisms in hardware systems (e.g., divide by zero, overflow, underflow). Although they are not application specific, they do represent an effective means of detecting design errors. The inability of an application to handle an exception or the failure of a runtime check can be used as the trigger for fault containment and/or self-healing measures.

**Fault Containment**

Software faults can be contained by avoiding particular execution paths of an application that are known to be faulty [88]. This can be done mainly by avoiding the execution of particular portions of the program or by modifying inputs that are known to be fault-inducing. This may be achieved by replicating the state machine that is implicit in all
programs on a parallel component, a controller model for the application. This component would be queried every time before the application executes a state transition in order to determine whether the particular transition is not suspected to cause a fault and is therefore enabled. Whenever a fault (a crash, exception or other condition) is detected, the state transition that caused that fault is marked as faulty and is disabled.

Checkpointing [132] is the technique of periodically storing the state of an application. It is not a fault detection mechanism but prevents applications from having to be restarted from the beginning for recovery or for debugging re-execution. Checkpointing can also be useful for masking transient faults. If faults are related to temporary conditions a checkpoint enables the re-execution of a small part of the program that was running just before the fault occurred. These temporary conditions may be hardware problems, disturbances in the application’s environment (a network failure), transient results of calls to external libraries or the operating system, or even concurrency bugs, in the case of multi-threaded applications, that only occur in particular interleavings of the application’s threads.

Fault Reporting

Another critical step in fastFIX is the reporting to the application’s developers of application faults that happen at a client installation. There are several sources of relevant fault description information that can be used for addressing the problems. This information can come from the execution environment or from the application itself, which may need to be modified or not to provide this information. The most relevant examples of fault reporting information are:

- Core dumps: Core dump is the general term to designate a snapshot of an application’s state which is recorded when a process crashes. Normally, it contains the state of the application memory and a snapshot of the processor registers at the moment of the crash.

- Context information: In some cases the reason why a program crashes may be better described by information external to the application. Most modern operating system include instrumentation mechanisms that provide metrics describing execution characteristics of the application (open files, bytes read/written to disk and/or network, page faults, etc.) Additionally, in the case of fastFIX we aim to include context awareness and event correlation sub-systems that can record additional environment information such as the activities that were taking place in other applications and what the user’s intentions were at the moment of the crash.

- Execution traces: If one wishes to understand the exact execution flow that lead to a software fault, one must record an execution trace of the application. This may be achieved by executing the application inside a virtual machine, e.g. [3, 23], or by instrumenting the code using for example Intel’s Pin [79]. Instrumenting code for
tracing is notoriously costly in terms of performance overheads. Therefore, there have attempts to reduce the amount of tracing, thereby resulting in incomplete non-deterministic traces which require debuggers to experiment with different execution paths [93].

- Non-deterministic Input: Finally, an essential component for accurately describe a fault is recording all non-deterministic input to the application. This includes user input data and the return values of library and system calls that do not return deterministic results.

Fault Tolerance and Self Healing

There is a synergistic interaction between fault-tolerance and self-healing.

In fastFIX, there needs to be a strong coupling between monitoring and fault-tolerance, because monitoring is normally heavy but essential to fault description and replication.

Fault containment can be useful in maintaining an operating system while the self-healing support system attempts to reconfigure or patch the application.

Self-healing can be considered as a natural consequence of fault-tolerance considering that faults are virtually unavoidable in a world where: i) software development is very fast; ii) release cycles are very short; iii) agile and prototyping development approaches make it difficult to use formal specification and verification methods; and iv) software is designed for an ever-growing array of devices and environments. Therefore, in such volatile development circumstances rigorous fault-tolerance design techniques are difficult to implement thoroughly. Consequently, fault-tolerance and software survivability is easier to achieve through a blend of runtime fault-tolerance techniques, fault replication and self-healing/patching techniques.

Finally, it should be noted that self-healing actions may be triggered by other events besides faults. For example, context monitoring may detect software or execution environment conditions that presage faults/crashes and that may lead to self-healing and patching measures, such as timing problems or data integrity violations.

In [130], the authors present an approach which allows for reconfiguration of distributed systems at runtime. This approach offers means to introduce fault-tolerance capabilities into the system. Operations that execute code blocks can be triggered at runtime. Models of these operations based on their signatures can be analysed in order to check if it is safe for some operations to be executed in the multi-threaded system. This approach and corresponding analyses are presented by the authors as a means for checking that the introduction of a fault-tolerance mechanism such as micro-checkpointing does not disturb the system initial functionalities.

A distinction is usually made between single-version and multi-version software fault-tolerance techniques. Single-version techniques deal with a single piece of software by
adding mechanisms into its design in order to handle errors at runtime. Multi-version fault-tolerance techniques use multiple versions (or variants) of a piece of software. This approach is based on the principle (somehow controversial) that different pieces of software designed and implemented by different persons contain different faults. A characteristic of the software fault-tolerance techniques is that they usually rely on some redundancy in the system and that they also can, in principle, be applied at any level in a software system: procedure, process, full application program, or the whole system including the operating system. In the following, we rely on [118] to provide an overview of single-version and multi version fault-tolerance techniques.

3.1.1 Single-Version Techniques

Single-version fault-tolerance techniques usually rely on some redundancy applied to a single version of a piece of software. Such common techniques are exception handling, checkpoint and restart, process pairs, and data diversity [80].

- Exception handling is a mechanism that allows for fault detection and handling. More specifically, exceptions can be raised when a normal interruption of the system is interrupted in an abnormal manner. The exception handling mechanism makes it possible to catch such an exception and design actions to be performed when it occurs.

- Checkpoint and restart consists of recovering from a system fault by simply restarting the system from a different state from the ones at which the fault occur. Checkpoint and restart relies on the principles that some faults rarely occur and are hard to reproduce: they do not seem to systematically occur even in similar situations. For these types of faults, restarting the system is usually enough to recover. In a basic recovery, the system can be restarted in a predefined state such as its initial state. More complicated cases can also be considered where the system can be restarted and set according to some checkpoints, i.e. snapshots of the system states are recorded at fixed time intervals or following some trigger events.

- Process pair techniques apply checkpointing techniques and rely on the presence of two processors in order to avoid a system restart. The first processor executes the piece of software and generates checkpoints that are sent to the second processor. When a fault occur, the second processor takes the processing role over from the last checkpoint it receives. The first processor then does not participate to the execution anymore and performs some diagnostics. After this phase, it can return to service but roles are swapped as it now receives checkpoints from the second processor, that is performing computations.
• Data Diversity techniques applies checkpoint and restart techniques and also considers new inputs for the system after restart. The principle of this approach relies on the fact that some faults are input dependent. For these types of faults, re-executing the software with similar inputs after restart will lead to similar faults. In order to overcome this issue, Data Diversity techniques apply the checkpoint and restart approach and also automatically compute “input sequence workarounds”. The automatically created inputs are meant to be similar to the ones that were initially provided to the system and aim to induce equivalent output. The notions of similarity (or equivalence) between inputs and outputs are usually application specific.

### 3.1.2 Multi-Version Techniques

Multi-version fault-tolerance techniques consider several versions (or “variants”) of a piece of software, that implement the same features. This approach is based on the expectation versions that were implemented differently by different people should also fail differently. The main multi-version fault-tolerance techniques are recovery block, N-version programming, N self-checking programming, consensus recovery block and t/(n-1)-Variant Programming. All of these approaches rely on similar principles, whose basic version is Recovery Block. Figure 3.1 represents the model for the recovery block approach. It considers that several version of the same piece of software are available. As long as no faults are detected, the primary version of the software process the inputs and performs some checkpointing. However, when a fault is detected, checkpoint and restart is applied to alternate versions and the execution is tried on these. The alternate versions can be designed in order to offer degraded performance, e.g. providing less accurate results.

N-version programming is very similar but assumes that each version fulfill the same requirements. Each alternate version process the input and a selection algorithm decides on a good output to be considered. N self-checking programming is yet another more sophisticated approach that combines recovery block and N-version programming approaches. Similar to recovery block, the different versions may not all fulfill the same requirements and possess their own set of tests in order to determine to detect the correctness of the generated output. Similar to the N-version programming approach, outputs are compared and a selection is made. Finally consensus recovery block and t/(n-1)-Variant Programming approaches also combine the recovery block and N-version programming approaches. They offer more sophisticated ways for testing and selecting outputs from the different alternate system versions.
3.2 Self-Healing

Self-Healing as it can be found in the literature derives from the fields of autonomic systems and techniques were inspired by work on fault-tolerance. Deriving from AS concepts, [70] presents a feedback loop for self-healing, similar to the one of Figure 2.2. This feedback loop is illustrated in Figure 3.2 and consists of 4 phases: monitoring, diagnosis, adaptation and testing. These phases match some of the phases of Figure 2.2 where the analysis phase corresponds to the diagnosis phase and the planning phase corresponds to the adaptation and testing phase. The execution phase corresponds in this case more to the deployment of the generated fix.

Another important point of Figure 3.2 is that self-healing is based on the fact that some anomalous events occur due to some failure in the system showing that the self-healing mechanism is only triggered whenever a fault occurred to which the system has to recover from.

As shown in Section 3.1, some studies in the field of fault-tolerance have been conducted in order to solve similar issues. As we will show in this section, some of the techniques from fault-tolerance are also considered by self-healing approaches such as redundant resources. According to [108], one difference between self-healing and fault-tolerant techniques resides in that self-healing usually tries to determine and remove (or mitigate) the cause of the fault, while fault-tolerance generally only bring the system to a state from which it
can resume with the same functionality as initially implemented. The authors also point out that self-healing is still a relatively immature field and that the class of faults tackled by this field remains quite narrow.

It exists different vision of self-healing in the literature. In [100], the author consider self-healing as equivalent to self-repairing and self-immunity, i.e. the ability to be robust to infections. A self-healing system must be able to recover and go back to a proper state following some disturbance. This is a view shared in [42], where recovery oriented computing is presented as a key aspect of self-healing. The authors consider that healing systems are more concerned about post-fault or post-attack states and more specifically about bringing the system back to “normal”. More specifically they consider that a self-healing system “should recover from the abnormal state and return to the normative state, and function as it was prior to disruption”. In [70], the author has a broader view of self-healing. In that work, fault-tolerance is seen as trying to bring the system to a state from which it can resume execution while self-healing performs a deeper job by trying to identify and eliminate or mitigate the root cause of the fault. In [74], the authors have a similar view of self-healing and recall that the system requires knowledge about its expected behaviors in order to automatically discover system malfunctions or possible futures failures.

Consequently, in this section as well as in Chapter 4, we take a broad view on self-healing in the sense that works that deal not only with repair but also with diagnostics are considered. We also focus more on software system under the occurrences of faults or where vulnerabilities are being exploited with possible malicious intentions.

In this section we present existing work on self-healing that can be found in the literature under an explicit “self-healing” banner. As described in the introduction of this document, the concept of self-healing relate to a broad area of computer science and many works on self-healing are not explicitly described as such. Most of these works are related to automatic diagnosis and repair and are presented in Chapter 4. Works that are explicitly
presenting self-healing approaches are presented in subsection 3.2.1 regarding articles found in the literature and in subsection 3.2.2 regarding other EU funded projects tackling self-healing issues.

### 3.2.1 Research articles on Self-Healing

In [108], the author contribute to the developing self-healing systems in three directions. First, they are interested in self-healing problems related to concurrency. Secondly they consider the self-healing of functional issues. Finally, they also consider self-healing related to performance issues. This work is related to the SHADOWS project, which is detailed in Section 3.2.2.

In [110], the authors present ASSURE, a self-healing approach based on rollback and error handling facilities. ASSURE considers rescue points in the program, which are introduced at implementation time in place where it is suspected that failures may be introduced. When a fault occurs at execution time, ASSURE rolls back and restores the program to the nearest previously encountered rescue point. From there, the execution is virtualized and error handling facilities are executed in this virtualized environment. When an error occurs at runtime and the system is brought back to a rescue point, pre-defined error handling are executed in the virtualized environment and tested. If it is satisfactory, then the error handling code is applied to production code. This approach makes it possible to self-heal from unknown issues by applying recovery approaches for known issues, that also seem to apply to the unknown failure.

[73] presents a tool for concurrency testing called ConTest. ConTest has plugins which allow for concurrency bug runtime detection as well as bug healing. Two approaches are considered for the detection of concurrency bugs. The first approach consists for each shared variable of analyzing the set of locks that are used at runtime to protect this variable. This approach makes it possible to detect race conditions but can provide false alarms. The second approach focus on detecting the consequences of concurrency bugs rather than its possible causes. Although some bugs may be missed with this approach, it has the advantage of not producing any false alarms. This second approach relies on some descriptions of atomic sections (i.e. sections of code that cannot be interleaved with other threads) and a bug is detected when some interleaves occur while executing an atomic section. These atomic sections can be described manually or can be extracted from static analysis.

In this work, two approaches are also considered for bug healing. Neither of them actually correct the detected bugs but mask their symptoms instead. The first one consists of affecting the scheduler behavior so that interleaves do not happen while a thread is executing critical code. The second approach consists of injecting new locks into the code.
Although this approach fixes bugs, it might impact on the system performance and also introduce new deadlocks.

In [112], the authors introduce DIRA, an approach which allows program to recover from attacks. DIRA tackles automatic detection, identification and repair of control-hijacking attacks. This solution is implemented as a GCC compiler extension that instruments program source code so that the resulting program can perform these tasks. Checkpoints are introduced in the program in order to ensure that corruption of state can be detected if control sensitive data structures are overwritten. Unfortunately, DIRA impacts on system performance in a non negligible way.

In [111] presents STEM, an approach for recovery from attacks. The STEM approach relies on two principles: code emulation and error virtualization. In STEM parts of the code that are known as being error prone can be emulated before being executed. If the emulation is successful, snapshots of internal states can be copied to the real environment. If an error occurs STEM can handle that failure thanks to the error virtualization mechanism, which allows to catch errors that were not initially meant to be caught by the application. This mechanism refers to some predefined fixes in order to handle caught errors.

In [131], the authors present a self-healing approach for the automatic identification of memory corruption vulnerabilities. The approach relies on the fact that memory corruption attacks usually cause the application to crash. Crashes are used as a trigger for a diagnostics which indicates the instruction that is tricked to corrupt data, the call stack at the time of corruption, and the propagation history of corrupted data. Moreover, the diagnosis process also generates a signature of the attack using data/address values embedded in the malicious input message.

The generated fix is based on this signature, which is used to block future attacks. The overall approach enables the development of a decentralized self-diagnosing and self-protecting defense mechanism for networked computers.

The work performed in [47] achieve self-healing using numerous techniques from different fields. The techniques used rely on aspect-oriented programming, static analysis, dynamic analysis, model-checking, artificial intelligence and machine learning.

First, aspect-oriented programming is considered to instrument the code of the application with sensors. Then the analysis techniques (static, dynamic, model-checking) are used to determine the root cause of failures whose symptoms are observed through the sensors. Finally healing actions to be taken or suggested are decided by applying artificial intelligence and machine learning techniques.

In [17, 18, 6], Carzaniga et al. consider the “workaround” approach, which consists of replacing a faulty sequence of operation with another that produces the same outputs or effects. Therefore workaround do not correct faults per say. Instead, they mask their
occurrences by providing alternative program executions that produce the same effects. The idea behind this approach is that libraries often contain feature redundancy. A typical example, provided in [6], is the one of changing an item in a shopping basket. The “change item” feature can be achieved by composing the “remove item” and the “add item” features, i.e. an item change can be seen as the removal of an item followed by the addition of another one.

More specifically in [6], the authors consider the case of web applications, which they show contain a significant amount of feature implementation redundancies. Workarounds are used as a way of ensuring proper feature execution by disabling incorrect implementation of these redundancies.

In order to achieve this, they assume that fault detection is provided by users themselves through the use of a “fix-it!” button on the web application interface. The idea is that users can indicate the system that the application is not behaving as expected by pressing this button. The system will then consider the sequence of operations that were attempted and will look for a workaround using a formal representation of the possible equivalent sequences. Finite State Machines are one of the possible representations envisaged by the authors. When a workaround has been found, the corresponding sequence of operations is invoked instead of the faulty one.

### 3.2.2 Other EU projects on Self-Healing

The SHADOWS project aims to combine different techniques in order to provide self-healing capabilities to a system. Several types of issues can be self-healed through this approach. Depending on the issue, the self-healing process may be fully automatic or at least facilitates the system design phase. Therefore, the approach enables self-healing throughout the software lifecycle and help developers at both design time and runtime. The SHADOWS approach tackles different types of issues to be self-healed.

- Concurrency related issues.
- Functional issues.
- Performance issues.

The approach taken in SHADOWS is to self-heal different steps of the system life cycle (design, testing, and deployment stages) rather than only the system itself. Such an approach keeps human intervention in the loop but allow for more relevant solutions.

For instance, the concurrency problem is tackled by considering models for safe concurrency and developing algorithms that examine the program during coding and testing phases in order to detect inconsistencies with the model (i.e. unsafe concurrency patterns.
in the code). Such detected issues are then solved by automatically introducing scheduling and synchronization statements in order to affect the concurrency behaviors of the code.

The approach for healing from functional issues is model-based. Three types of models are considered: system, fault, and component models. The system model represents expected behaviors of the system. The fault model specifies the types of faults that can be identified and repaired by the functional self-healing solution, such as faults related to incompatibility between components. The component models represent normal operation inter-component interaction patterns. Faults that are detected from using these models are recovered through reconfigurations. These reconfigurations are the result of intra or inter component self-healing capabilities.

Finally the approach to solve performance issues is also model-based and relies on a system-level performance model and component-level performance models. The system level performance model specifies system performance requirements while the component-level performance models specifies requirements on expressed by thresholds set on component-level parameters. Diagnosis is performed by correlating the system level and component-level models. When a performance issue is detected, the healing process consists of computing new values for the component-level parameters, which will ensure the avoidance of these performance issues in the future.

Figure 3.3 illustrates the self-healing approach taken in the SHADOWS project. The architecture of this approach relies on several layers and repositories. It includes a model repository, a monitoring layer, an analysis layer, and a correction layer. The model repository contains models for each of the issues targeted, i.e. safe concurrency, functional requirements, and performance. These models are used to detect the occurrence of issues by comparing the deviation of the runtime behavior with the ones described by the models. This detection is performed in the monitoring layer. The detected deviations are analysed further in the analysis layer. This provides some determination of the issue that occurred. Once an issue has been determined, it needs to be corrected. Errors that can be fixed at runtime are handled by the Run Time Self-healing layer while the other errors are handled by the Design Time Self-Healing layer. Therefore these error corrections are performed on the system either at design phase or after deployment.

### 3.3 Summary

This chapter introduces fault-tolerant and self-healing approaches. Self-healing principles were initially inspired from fault-tolerance, especially by relying on redundancies that are in network systems. Then self-healing was further developed and this chapter also presents self-healing approaches that do not rely on redundancy. The works presented
Figure 3.3: The architecture of self-healing systems proposed by the SHADOWS project [108]
consider several types of techniques employed, as well as several types of faults to heal from: vulnerability attacks, concurrency and functionality issues as well as performance issues.

It is also important to note that the community agreed on the differences that characterize fault-tolerance and self-healing. In [100] for instance: “Fault-tolerance aims at keeping the system running at 100% of its designed functionality, while self-healing can mean that after the healing the system operates at less than 100%. If a system is able to self-heal completely the occurrence of faults such that the operation of the system is not impaired at all, it could be called fault-tolerant.”.

Other characterization such as in [42] are given: “fault tolerant computing systems are able to continue discharging their normal operation despite the presence of malicious or arbitrary faults. However, they do not comprehend 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.”

Finally, as pointed out in [42] again, the current research on self-healing focuses on “the offline healing of components, being fault-diagnosis, recovery and re-induction of repaired element into the system.”. Following this idea, Chapter 4 presents works on automatic diagnosis and automatic repair. Although these works are not explicitly described as related to self-healing, they are very relevant to achieving this goal.
4 Automatic Diagnosis and Repair

Chapter 3 discussed works considering fault tolerant or self-healing approaches. In this chapter, we present works that are related to self-healing without explicitly stating it. The scope of this section is driven by the observation that self-healing can be seen as a combination of self-diagnosis and self-repair (see e.g. [102]). However, there is very little work that explicitly state dealing with self-diagnosis and self-repair. For this reason, we extended here these notion to automatic diagnosis and automatic repair.

4.1 Automatic Diagnosis

Diagnostics pertains to the detection and isolation of faults or failures ([96]). In this section, we present works on automatic system diagnosis. The aim of such automation is to reduce the diagnosis time and therefore to react more quickly to faults occurrences.

4.1.1 Architecture-Based Approaches

In this section, we present automatic diagnosis techniques that make use of the architecture of the system to be diagnosed. These architectural properties usually rely on resource redundancy. The following works for instance consider either processor, network node or processes redundancy. The main goals of these works is to take advantage of the architectures of the system in order to conduct automatic diagnostics.

In [106], the authors consider the feasibility for a multi-processor system to perform self-diagnosing on some of its processors by some others. The underlying techniques rely on communications between processors. The possible interaction link between processors are modeled by an undirected graph, where nodes represent processors and an edge between two nodes represents a communication link between two processors. A processor linked to two distinct ones can contribute to their diagnosis by sending to both processors similar stimuli and comparing the outputs they each return. A mismatch between outputs that correspond to a same test stimulus may indicate the presence of a fault in one of the processor. This work considers conditions under which it is possible to diagnose the whole system considering not every processor is linked to every other and that several processors may be faulty.
In [53], the authors study the diagnosis of networks whose nodes represent processing elements. They consider the case where no central entity can control and interact with these elements. The system nodes can determine each other’s status via inter-communications. This work considers the feasibility and conditions for diagnosing the network considering that possibly both nodes and links may be faulty. Similar to [106] (although applied to a different type of systems), diagnosis is performed by using some nodes in order to test other nodes.

[106] and [53] consider physical duplicates (processor or network nodes) in order to achieve diagnostics, which is conducted through testing. In [66], the authors consider process duplicates that may run in a virtualization environment such as a cloud. A tool called PeerWatch was implemented, which diagnoses processes in a cloud through comparison of the behaviors of different instances of a same process running in the cloud. Correlations between the characteristics of instances are performed. A fault and the process that initiated it are detected whenever the correlation between this process and its other instances drops. Once a fault has been detected, PeerWatch can perform some diagnosis which also relies on the presence of multiple instances of the process. The diagnosis combines spatial and temporal measurements across multiple instances before and after the failure.

4.1.2 Correlation-Based Approaches

Correlation based diagnosis techniques consider the case of diagnosing systems which may provide several types of alarms. An alarm may not correspond to an actual issue, but some combination of alarms may. The following works consider these cases and study approaches for automatically deciding on a diagnosis in the presence of alarms.

In [32] the authors study conditions under which failures can be diagnosed in a system where some alarms can be observed. Their approach applies to Discrete Event Systems for which a model of possible behaviors and alarms is described by Finite State Machines (FSM). The authors consider the case where several FSM are available, each representing different partial views of the system. The problem under consideration is the detection of such alarms whenever they are not directly observable. In other words, the authors study the possibility of inferring the presence of unobservable alarms from other event observations. This approach was considered in [104] for the case of a single view of the system. It is here augmented to the case of several partial views where the information from these views are correlated at runtime.

In [83] the authors consider fault localization in the presence of alarms and through the use of architectural information. Alarms here correspond to abnormal metric scores of some components. Architectural dependency graphs and rule sets are used in order

Figure 4.1: Creation of application signatures from runtime behaviors [35].

In [64] the authors correlate metrics as a means for diagnosis. These metrics are collected periodically and can relate to system states or performances. The authors’ approach resides on the fact that during normal system behaviors, some correlation holds between pairs of metrics. These stable relationships between metrics characterize normal system behaviors and faults can be detected whenever observations deviate from stable metrics correlations. Finally, when considering knowledge on component dependencies, this approach makes it possible to locate faulty system components.

In [35] the authors propose an approach on black box systems, based on signature correlation. The approach relies on ptrace ([45]) in order to collect information such as system call attributes, signals, environment variables, resource limit and access control. Signature of normal system behaviors are built from these attributes. Figure 4.1 illustrates how signatures are created. Basically, different possible values observed at runtime for an attribute are merged into a signature.

Whenever a fault occurs, the values of the corresponding attributes are correlated to normal signatures. If the value of an attribute is not part of the possible values of the signature, then this attribute is considered as a possible cause for the fault. In order to illustrate this idea, the authors consider a signal SIGXFSZ in a faulty execution. As SIGXFSZ is not a possible signal value according to the signature provided in Figure 4.1, it is considered as abnormal. Further investigations on the semantic of the signal can show that it is trigger when writing a file larger than the maximum allowed size. Therefore, the cause of the fault can possibly be determined by checking the size of the files used by the application. Such an attribute would be available as it is monitored in this approach.

In this work, the authors further consider system calls as attributes as well as multiple
threaded applications.

4.1.3 Probabilistic-Based Approaches

In this section we present works which rely on probabilistic models in order to perform diagnostics.

In [125] the authors deal with the diagnosis of faulty links in networks. They consider an architecture where a designated node is in charge of determining occurring faults. Faults are detected when this node cannot communicate anymore with some nodes while other nodes still can. It is crucial for the network to be able to identify as quickly as possible the most likely failing link. This makes it possible to limit the amount of tests to be performed in order to diagnose the model by restricting the set of links to be tested. The approach for determining possibly failed links relies on the computation of maximum a posteriori tests and requires expertise to define a priori link failure probabilities.

In [51] the authors consider a learning approach of the normal behaviors of the system. Deviations from the norm are detected and the information gathered is combined in a Bayesian Network. Bayesian Networks can be seen as graphs whose nodes represent random variables and edges between nodes encode some direct influence in between the two corresponding variables. An advantage of this approach is that it makes it possible to detect unknown faults.

[109] deals with diagnosing software which were compromised from malicious attacks. The approach uses a mix of symbolic and bayesian inference. The symbolic reasoning is used to predict the behavior of the system for a given mode (e.g. normal mode). If a conflict is detected a new node is added to the bayesian model. Bayesian inference is then applied in order to update probabilities over the model. This makes it possible also to discover unknown failures.

[20] presents an approach for diagnosing failures in Internet web sites. The approach uses a decision tree model and learning techniques in order to encode the likelihood that some behaviors lead to some failure.

In [114] the authors present a probabilistic fault localization technique. The approach relies on a symptom-fault map model which helps identify faults from observed symptoms. Symptom-fault maps are modeled by bipartite directed graphs where edges encode causal relationships between faults and symptoms.

In [41], the authors consider Bayesian Networks (BN) as a model for diagnosis. An advantage of BN is that they offer a graphical representation that make them easy to manually be modified, hence facilitating the addition of human expertise. BN are also used as a model for diagnostics in [40] and are considered there again not only for their learning facilities but also for their ease to be manually modified.
Finally [22] presents HOLMES, which is a tool based on statistical debugging techniques. The underlying approach considers program runs over time and generates bug reports due to failures. Once HOLMES collects a sufficient number of bug reports, it combines them with information from static analysis of the program. This aims to identify portions of the program that are most likely to contain the root causes of observed bugs. It then automatically identifies parts of the program that should be monitored in more detail, making dynamic the monitoring facilities. Following this, more accurate bug reports are generated whenever errors occur again. HOLMES then uses statistical analysis to assign scores to paths that represent possible predictors of a failure. HOLMES can then either decide that the root causes of reported bugs have been identified from these paths, or to continue the search by collecting more detailed information from even more accurate monitoring.

4.1.4 Model-Based Approaches

In this section, we present works that consider a Model-based but non probabilistic approach for diagnosis. More specifically, these models take as inputs some observations of the current state or behavior of the system and return a diagnosis.

First [99] introduces the notion of Model Based Diagnosis. Roughly speaking this approach consists of comparing a model of the behaviors of the system to be diagnosed with actual observed executions. When the observed execution deviates from the expected behavior provided by the model, this indicates that a fault has occurred. If moreover a model of faulty behaviors is available, then it is possible to identify the faulty system component. In [115] the author considers the case of systems for which a static model is available, requiring the use of model simulation in order to compare an actual execution to normal (or abnormal) ones. As the computation induced by such simulation can be relatively expensive, the authors provide theoretical foundations for which this can be avoided by considering temporal descriptions of the system model.

In [2] the authors consider Model-Based Diagnosis (MBD) techniques using logic in order to reason about program traces and hence perform fault localization whenever a fault occurs. Their approach also assumes that the system under consideration consists of several components, that test cases are available for it, and that some traces are collected during the program execution. When a fault occurs, a diagnosis results in determining the subset of the system faulty components. Techniques similar to the MBD approach are considered. However, in this work, not only static information but also dynamic information is taken into account. In this framework, the authors show that it always exists a test set that makes it possible to reveal actual faults.

In [104, 103] the authors consider systems modeled with Finite State Machines (FSM).
These models describe the possible executions of a system as well as the occurrences of possible failures. As these failures are usually not observable from the actual system, the problem to be solved consists of building a model for diagnosis which, given a possible system behavior, indicates whether the system has reached a faulty state. Such a model is called a diagnoser. The authors introduce a theoretical framework in order to formally specify and solve the problem of automatically building such a diagnoser. In particular, they define the notion of system diagnosability, which roughly corresponds to the property a system must ensure so that a non observable failure can always eventually be detected by observing other events. The authors provide necessary and sufficient conditions for a system to be diagnosable as well as an algorithm which computes a diagnoser.

4.1.5 Summary and Conclusion

This section presents different techniques and approaches to automatic diagnosis. These works were classified among several categories: architecture-based, correlation-based, probabilistic and model-based.

Similar to works on fault-tolerance, the architecture-based approaches relies on some redundancy and applies to systems with several processing units such as multi-processor or distributed systems. The correlation-based approaches assume that some alarms/triggers are present in the systems and actual global diagnosis is performed by correlating these alarms. Probabilistic approaches rely on a probabilistic model in order to infer a diagnosis from system observations. Designing such models is challenging and requires historical data or expertise. Finally, as for probabilistic diagnosis, model-based approaches consider models that automatically perform diagnosis. However, unlike for the probabilistic-based approach, the model-based one appears to have been conducted at a quite theoretical stage.

4.2 Automatic Repair and Bug Fixing

As explained in [102] Self-healing can be seen as a combination of self-diagnosis and self-repair approaches. A broader view of these concepts are “automatic diagnosis” and “automatic repair”. Section 4.1 gives an overview of some software automatic diagnosis approaches. In this section, we focus on automatic repair approaches, which also tackles automatic patch generation. These approaches are grouped according to the techniques they use. These techniques span over Rollback, Genetic Programming and Mutations, Event Filtering, Probabilistic and Machine Learning techniques, and model-checking techniques.
4.2.1 Rollback Techniques

The so-called rollback techniques maintain a record of “healthy” system states to allow a rollback to the last such state when a fault occurs. Once successfully rolled back to a healthy state, the system re-executes after applying certain changes to its input data or execution environment. Such changes are necessary to prevent fault repetition. In general, the rollback techniques improve the overall system availability via fault-tolerance. However, most of them suffer from one or more of the following limitations [95]:

- required application restructuring;
- inability to address deterministic software errors;
- unsafe speculation on program execution;
- relatively long recovery time.

Rx. In [95] is proposed an innovative rollback technique, called Rx, which is intended to quickly recover programs from arbitrary types of software failures, both deterministic and non-deterministic. Before re-executing, Rx makes changes in the execution environment presuming that many of the software bugs are correlated with the execution environment, and therefore can be avoided by removing the “allergen” from the environment. Depending on the fault analysis, there could be a few to none modifications in the environment, e.g., external applications providing services or resources. Rx is implemented for Linux and experimented with server applications containing various priory-known bugs. The benchmark with two rollback alternatives, a whole program restart approach and a simple rollback and re-execution without environmental changes, demonstrated that Rx handles better both deterministic and nondeterministic failures.

The Rx rollback methodology is to rollback the system to a recent checkpoint upon a software failure and re-executes it after applying symptom-related changes in the environment. By changing the environment eventually the allergen is removed and the failure does not occur again. Further, in case of a successful re-execution, the environmental changes are disabled to reduce time and space overhead imposed by those very changes. The Rx methodology got inspired from real life cases of medical treatment of allergy, where the most common treatment is to remove the allergens from their living environment. This helps to come up with diagnosis of the cause of the symptoms, when the latter are successfully treated by removing one or more allergens from the environment. Note that such treatment cannot start before the patient shows allergic symptoms. The tradeoff coming with this approach is that the system may temporarily deny services, e.g., dropping user requests. This effect can be minimized by giving a spectrum of possible
environmental changes and try the least intrusive changes first and reserving the more extreme ones as a last resort for when all other changes have failed.

Figure 4.2 shows the Rx methodology. As shown, Rx periodically takes light-weight checkpoints. The latter are specially designed to help the system survive software failures. Thus, when a failure is detected, the system is rolled back to the most recent checkpoint. The failure detection is upon exception or is provided by special integrated failure-detection mechanism called Rx Sensors. Once reversed to a checkpoint, Rx analyzes the failure based on the failure symptoms and “experiences” accumulated from previous failures, and determines how to apply environmental changes to avoid this failure. Finally, the program re-executes from the checkpoint in the modified environment. Note that this process might go over a few iterations by repeating re-execution from different checkpoints and applying different environmental changes until either the failure does not reoccur or Rx times out. In the latter scenario, Rx tries alternative options such as whole-system rebooting. At the end, when the failure is fixed, the environmental changes are disabled to avoid further overhead associated with those changes.

The execution environment targeted by Rx includes all the external systems and resources that can affect the execution of the system in consideration. For example, those might be:

- hardware such as processor architectures, devices, etc.
- OS kernel such as scheduling, virtual memory management, device drivers, file systems, network protocols, etc.
- standard libraries, third-party libraries, etc.

Note that the execution environment cannot be arbitrarily modified for re-execution. Thus, a useful re-execution environmental change should satisfy two properties:

1. It should be correctness-preserving, i.e., every step (e.g., instruction, library call and system call) of the system is executed according to the APIs.
2. It should be able to inherently avoid some software bugs. For example, padding every allocated buffer can prevent some buffer-overflow failures.

**ASSURE.** ASSURE [110] is another rollback technique that introduces *rescue points* to recover software from unknown faults while maintaining both system integrity and availability. ASSURE strives to preserve system availability in server applications by mimicking system behavior under known error conditions. Rescue points are locations in existing application code for handling a given set of programmer-anticipated failures, which are automatically repurposed and tested for safely enabling fault recovery from a large class of faults. Thus, when a fault occurs, ASSURE restores system execution from an appropriate rescue point by virtualizing the program’s existing error-handling facilities. Rescue points are identified by using *fuzzing*, a software testing technique that provides invalid, unexpected, or random data to the inputs of the system. To implement this, ASSURE uses a fast and coordinated checkpoint-restart mechanism that handles multiprocess and multi-threaded applications. This mechanism is “injected” into the production code by using *binary patching*. ASSURE is implemented for Linux and operates with changes neither in the application source code nor in the OS kernel.

**Other Rollback Techniques.** Reboot techniques, including whole program restart [116], software rejuvenation [72], and micro-rebooting [16], attempt to return a system to a clean state before or after encountering a fault. Whole program restart can take a long time, resulting in substantial application down-time. Micro-rebooting can be faster by only restarting parts of the system, but requires a complete rewrite of applications to compartmentalize failures. None of these techniques effectively deal with deterministic bugs, since these may recur post-restart.

Other checkpoint-restart techniques [14, 71] can be used in a manner similar to whole program restart, but can provide faster restart times since restarts are done from a checkpoint. When used in this way, these techniques still do not handle deterministic bugs, since these bugs will still occur after restarting. Other uses of checkpoint-restart in conjunction with running multiple program versions have also been proposed [14] which may survive deterministic bugs if failures occur independently. However, they incur prohibitive costs for most applications in terms of developing, maintaining, and running multiple application versions at the same time.

### 4.2.2 Genetic Programming and Mutations Techniques

It is recognized that software testing can take up to half of the resources of the development of new software. Although there has been a lot of work on automating the testing
phase, fixing an error after its presence has been discovered is still a duty of the programmers. Genetic programming (GP) is a computational method inspired by biological evolution, which discovers computer programs tailored to a particular task. In general, GP techniques consider *automatic modifications* of the source code in order to obtain variation of the faulty piece of program. In GP, programs typically evolved to accomplish a particular task (a typical example is the evolution of a machine learning classifier). Some metrics are defined in order to determine whether these modifications are valid. New modifications are considered until a valid one has been computed. The *mutation techniques* rely on similar concepts but are closely related to data structure linking and modification.

**Automatic Bug Fixing (ABF).** Automatic Bug Fixing (ABF) [10, 9] is a generic programming approach to automated testing based on the so-called *co-evolution*. In this approach, programs and test cases co-evolve, influencing each other with the aim of fixing discovered program bugs. ABF takes as input both a formal specification of a software system an implementation of the same and uses special evolutionary techniques to fix the eventual errors (bugs). Here, the GP approach is used to allow a system evolve in order to satisfy its formal specification. A special training set composed of unit tests is provided. The unit tests are generated based on the formal specification. Ideally, there could be generated many unit tests, however, only a small set of relative test cases should be considered, because the computational costs for evaluating an evolutionary program would be too high. Thus, GP is used to produce the initial set of unit tests and special *Search-based Software Testing* techniques [84]are used further to derive new unit tests that help to discover errors in the evolving system. The targeted errors need to be revealed by at least one of those test cases. This eventually leads to co-evolution of both test cases and the system itself, where the latter will evolve to the point satisfying the given formal specification.

In ABF, GP abstracts the target program as a tree, in which each node is a function with inputs represented as children of that very node. A population of program variations is held at each generation, where individuals are chosen to fill the next population accordingly to a special *problem-specific fitness function*. *Crossover* and *mutation operators* are then applied on the programs to generate new offspring. The goal is to pass all the provided unit tests. Therefore, ABF relies on GP to modify the program in order to pass the failed unit tests, but also takes precautions that the new modifications do not compromise the former functionalities. Although ABF provides a high-level of automation, the developers still have to inspect the output of the ABF system. The problem is that even all the test cases passed the check that does not necessarily mean that all the errors are fixed. In fact, the system might evolve to a partially patched program that is able to
pass the formerly failed tests, but it might still have errors.

Moreover, a special form of post-processing is required to facilitate the human perception of the code changes made by the ABF system. Besides fixing errors (or bugs), GP might change some parts of the code without changing its semantics. For example, a constant number like “4” might be replaced by expressions like “2+2” or “0+100−96”. This type of changes might complicate the human inspection of the source code after ABF has been applied.

**Automatic Patches with GP.** GP maintains a population of individual programs. Computational analogs of biological mutation and crossover produce program variants. Each variant's suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. A significant impediment for an evolutionary algorithm like GP is the potentially infinite-size search space it must sample to find a correct program. To address this problem, in [129] are proposed two key innovations:

1. A restriction of the GP algorithm is proposed to produce only changes that are based on structures located in other parts of the target program. For example, it is hypothesized that a program that is missing important functionality (e.g., a null check) will be able to copy and adapt it from another location of the same program.

2. New constraints over the genetic operations of mutation and crossover operators are proposed to operate only on the region of the program that is relevant to an error, i.e., the portions of the program that are on the execution path producing that error.

Combining these innovations, [129] demonstrates automatically generated repairs for ten C programs totaling 63,000 lines of code. GP is used to maintain a population of variants of those programs. Each variant is represented as an abstract syntax tree paired with a weighted program path. Further, variants are modified by using crossover and mutation genetic operations. Each modification produces a new abstract syntax tree and weighted program path. The fitness of each variant is evaluated by compiling the abstract syntax tree and running it on the test cases. Its final fitness is a weighted sum of the positive and negative test cases it passes. The process stops when an evolved program variant passes all the test cases. To fix the irrelevant changes produced by GP, it is used special tree-structured difference algorithms and delta-debugging techniques in a post-processing step.

In another research venue [89], the same researchers propose a GP approach to automate the task of repairing program bugs in existing software. In this approach, programs are evolved and evaluated until one is found that retains the functionality of the original program and fixes the error that occurred. The source code of the program is processed
first, to produce a path containing traces of execution procedures. This helps developers obtain a negative execution path containing the list of executed statements, when an error occurs. Next, the GP algorithm creates new programs by modifying the original code with more bias toward statements that occurred during the negative execution path. Further, additional tests are incorporated into the fitness function to retain the functionalities of the program.

**Automatic Data Structure Repair.** The data structure repair approach [81] is a mutation technique that uses given structural integrity constraints for key data structures to monitor their correctness during the execution of a program. If a constraint violation is detected, repair performs mutations on the data structures, i.e., corrupt program state, and transforms it into another state, which satisfies the desired constraints. The primary goal of data structure repair is to transform an erroneous state into an acceptable state. Therefore, the mutations performed by repair actions provide a basis of debugging faults in code (assuming the errors are due to bugs). A key challenge to embodying this insight into a mechanical technique arises due to the difference in the concrete level of the program states and the abstract level of the program code: repair actions apply to concrete data structures that exist at runtime, whereas debugging applies to code.

A broad class of systems monitors ongoing physical or information processes and presents summarized results to human users. The data structures in these systems typically reflect a sliding window of observations and predictions centered around the current time. Most data structure properties are transient, sliding through the system as it continually rebuilds its data structures to reflect its ongoing movement through time. In this context, any data structure anomalies will eventually be flushed out of the system as long as it continues to operate. **Automatic data structure repair** [33] is a mechanism that can enhance the ability of such systems to execute through errors and eventually move back to a completely correct execution.

In [33] is proposed an approach to automatic data structure repair that involves two data structure views:

1. a **concrete view** of the level of the bits in memory;

2. an **abstract view** of the level of relations between abstract objects.

The abstract view facilitates both the specification of high-level data structure constraints (especially constraints involving linked data structures) and the reasoning required to repair any inconsistencies. Each specification contains a set of model definition rules and a set of consistency constraints. Given these rules and constraints, a proposed tool [33] automatically generates algorithms that build a model, inspect that model and the data
structures to find violations of the constraints, and repair any such violations. The repair algorithm operates as follows:

1. **Inconsistency detection** - evaluate the constraints in the context of the current data structures to find consistency violations.

2. **Disjunctive normal form** - convert each violated constraint into disjunctive normal form; i.e., a disjunction of conjunctions of basic propositions. Each basic proposition has a repair action that will make the proposition true. For the constraint to hold, all of the basic propositions in at least one of the conjunctions must hold.

3. **Repair** - repeatedly select a violated constraint, choose one of the conjunctions in that constraint’s normal form, then apply repair actions to all of the basic propositions in that conjunction that are false. Note that the repair actions for one constraint may cause another constraint to become violated.

4. **Eliminate infinite repair loops** - to ensure that the repair process terminates, pre-analyze the set of constraints to ensure the absence of cyclic repair chains that might result in infinite repair loops. If a specification contains cyclic repair chains, attempt to prune conjunctions to eliminate the cycles.

The proposed tool has been used to repair inconsistencies in four applications: an air-traffic control system, a simplified Linux file system, an interactive game, and Microsoft Word files.

**Juzi - an Approach to Data Structure Repair.** A variety of tools [33, 34] have been developed to repair structurally complex data that do not satisfy the desired structural integrity constraints at run-time. Conventional use of such tools has been to enable continued execution of programs in case of otherwise fatal data structure corruption. Juzi [37] is a tool for automatic repair of complex data structures. Juzi takes as inputs both a Java class representing a data structure and a predicate method that specifies the structural integrity constraints. Further, the tool instruments its inputs and generates a new Java class that behaves similarly to the original class, but can automatically repair itself when the structural integrity constraints are violated. Juzi implements a novel repair algorithm. This algorithm considers a structure that violates its integrity constraints to perform a systematic search based on symbolic execution to repair that structure. Thus, the structure mutates in such a way that satisfies the given constraints. Experiments on structures ranging from library classes to standalone applications, show that Juzi repairs complex structures while enabling programs to recover from erroneous executions caused by data structure corruptions.
4.2.3 Event Filtering Techniques

Event filtering techniques are usually related to software security and vulnerability. They consist of automatically creating and detecting signatures or patterns for malicious attacks such as control hijacking and code injection. Then these signatures are used for a filtering check, so that such attacks cannot break through in the system anymore.

**PASAN.** PASAN [113] is an approach to security and protection in networking applications. In general, it provides a mechanism for incrementing the source code of target systems with additional instructions to detect “control hijacking” and to record enough run-time information from which can be derived both the attack signature and patch for a detected attack. PASAN uses the recorded run-time information to reconstruct the data and control dependencies that actually take place at run time. Because all known control-hijacking attacks corrupt the target address of certain instructions [113], after detecting such an attack, PASAN:

1. finds the corrupted target address;
2. identifies the basic block in which the compromised target address is last modified;
3. identifies the data structure that is being overflowed and its size;
4. creates a bound check to prevent the overflowed data structure from being overflowed in the future.

The resulting bound check and where it should be inserted constitute a software patch that permanently fixes the vulnerability. The current PASAN prototype focuses only on stack-based buffer overflow attacks, which overflow a function’s local array/buffer and eventually modify that function’s return address.

**FLIPS.** FLIPS [78] provides an approach to detection and neutralization of code-injection attacks. The methodology is based on instruction set randomization to detect and create the attack signature. FLIPS uses a selective transactional emulator that executes the vulnerable functions in a sand-boxed environment, and unrolls the memory updates made by the attacker. As a result, the target system may continue normally operate once the attack packets are removed. FLIPS incorporates three major components: an anomaly-based classifier, a signature-based filtering scheme, and a supervision framework that employs instruction set randomization (ISR). ISR is the process of creating a unique execution environment to effectively negate the success of code-injection attacks. This unique environment is created by performing some reversible transformation on the instruction set, where the transformation is driven by a random key for each executable.
The binary is then decoded during runtime with the appropriate key. Note that the randomization of an instruction set requires that the execution environment possess the ability to de-randomize or decode the binary instruction stream during runtime.

Figure 4.3 shows the FLIPS methodology. As shown, FLIPS receives feedback from the supervision framework that monitors the execution of a vulnerable application. If the emulator tries to execute injected code, it catches the fault and notifies the classifier and the filter implementing the FLIPS filtering scheme. It can then terminate and restart the process, or simulate an error return from the current function.

**ShieldGen.** ShieldGen [27] is a mechanism that automatically generates data patches for unknown vulnerabilities subjects of zero-day attacks. A zero-day attack is a computer threat that takes advantage of computer system vulnerabilities that do not currently have a solution. Usually, a zero-day attack takes advantage of a problem (e.g., software bug) not known by the users or undisclosed by the software developers, and thus, before a patch has been released. It is named “zero day” because it occurs before the first day the vulnerability is known. The ShieldGen approach leverages knowledge of the data format of potential attacks to generate new potential attack instances (called probes) and uses a special zero-day detector as an oracle to guide the search of vulnerability signatures. In particular, it uses knowledge such as the protocol format used by a network-based attack or the file format used by a file-based attack.

Figure 4.4 shows the methodology employed by ShieldGen. The latter employs a zero-day attack detector (shown as oracle) to detect zero-day attacks. Next, information of a detected attack is sent to a special data analyzer that recognizes the attack based on its data format. Next, the probe generator & analyzer module produces a data patch.
and constructs new potential attack instances (or probes), based on the data format information. Further, those probes are sent to the oracle (or zero-day attack detector) to check whether they are identical to the real attack. Feedback from the oracle guides the probe generator in the construction of new probes, to discard attack-specific parts in the original attack data as well as retain the inherent, vulnerability-specific part.

ShieldGen considers a fast, patch-level protection in the form of a data patch rather than the more traditional software patch. A data patch serves as a policy for the data analyzer, and is based on the vulnerability or the software flaw that needs to be protected. The data analyzer uses the data patch to identify parts of the input data to cleanse as it is being consumed.

### 4.2.4 Learning and Probabilistic Approaches

This subsection presents learning and probabilistic approaches to automatic repair and bug fixing. In general, such techniques learn from past executions where bugs have been fixed. Thus, applied fixes are stored and can be retrieved and applied again or used when necessary to infer other possible fixes.

**Exterminator.** Exterminator [90] is a bug-fixing system that automatically corrects heap-based memory errors in C and C++ applications without intervention from the programmer. Programs written in C and C++ can easily introduce memory errors, including buffer overflows and dangling pointers. Usually, such errors are notoriously costly to repair. The problem is that tracking down their location in the source code is difficult, even when the full memory state of the program is available. Moreover, correcting such errors is still a challenging task too. Practice has shown, that even for critical security-sensitive bugs, the average time between initial reports and the issuance of a patch is
nearly one month [90].

Exterminator exploits randomization to pinpoint errors with high precision. From this information, Exterminator derives runtime patches that fix these errors both in current and subsequent executions. In addition, Exterminator enables collaborative bug correction by merging patches generated by multiple users. The tool does not require source code nor programmer intervention. It relies on a probabilistic error-discovery mechanism and when an error is discovered, the current heap image is recorded. Further, special probabilistic error-isolation algorithm works on the heap image record to locate the source and size of buffer overflows and dangling-pointer errors. If a buffer overflow is located, Exterminator determines the memory-allocation site of the overflowed object (or variable), and the size of the overflow. In case of a dangling-pointer error, it determines both the allocation and deletion sites of the dangled object. Next, Exterminator corrects the errors by generating runtime patches applied by special correcting allocator. The latter prevents overflows by padding objects, and prevents dangling-pointer errors by deferring object de-allocations.

Exterminator can operate in three distinct modes:

1. **iterative mode** - requires user interaction and runs over the user input to the system;
2. **replicated mode** - corrects errors on-the-fly;
3. **cumulative mode** - corrects errors across multiple runs of the same application.

**BugFix.** BugFix [63] is a debugging tool that automatically generates special suggestions (called bug-x suggestions) helping developers quickly find appropriate fixes. To generate bug-x suggestions, BugFix uses a machine-learning approach that considers knowledge gained from previous bugs that have already been identified and fixed. BugFix requires as input a faulty program and a corresponding test suite containing at least one failing test case. Further it computes and reports a prioritized list of bug-x suggestions for a given debugging situation at a program statement suspected of being faulty. A debugging situation can be considered as a characterization of the particular static and dynamic details of a suspicious statement that is under debugging. In general, a bug-fix suggestion is a textual description of how to modify a given statement such that a bug in the statement is likely to be fixed. An actual fix performed by a developer is textually represented by a bug-fix description. The tool is built upon concepts from machine learning, which allows the tool to learn about new debugging situations and their corresponding bug fixes that are encountered over time.

**ClearView.** In many software systems, availability is a strict requirement. In such systems, it is imperative to eliminate denial of service, i.e., the application should provide
service even in the face of errors. ClearView [94] is a tool providing a mechanism for automatically correcting errors in software systems developed to be highly available. In general, it can automatically correct previously unknown errors in commercial off-the-shelf software systems by patching those at runtime without requiring restart or perturbation of execution. The tool requires no human interaction or intervention. It works on binary code without access to source code or debugging information.

Figure 4.5 presents the architecture of ClearView. As shown, it consists of components responsible for five distinct functions:

1. **Learning**: ClearView observes the application’s behavior during normal executions to build a model that characterizes those normal executions. The model is a collection of properties (or invariants) presented over observed values of registers and memory locations. Each invariant is ensured to be satisfied during the observed normal executions.

2. **Monitoring**: ClearView classifies each execution as **normal** or **failed**, by using a set of monitors that detect failures. For each failed execution, a monitor indicates the location in the binary code where is detected the failure, and prevents negative consequences by terminating the application. ClearView might incorporate arbitrary monitors. The initial implementation relies on two monitors: Heap Guard detecting out-of-bounds memory writes, and Determina Memory Firewall detecting illegal control flow transfers.

3. **Correlated Invariant Identification**: When a monitor first detects a failure, ClearView installs patches that check previously learned invariants close to the location of the failure. Those invariant-checking patches are not intended to correct errors or eliminate the failure. Instead, they find a set of **correlated invariants** that characterize normal and failed executions. A **correlated invariant** is satisfied during normal executions but violated during failed executions. This typically happens in repeated or replayed attacks.

4. **Candidate Repair Generation**: For each correlated invariant, ClearView generates a set of candidate-repair patches that enforce that invariant. Some of these patches
change the values of registers and memory locations to reestablish the invariant whenever it is violated. Others change the flow of control to enforce observed control flow invariants. The hypothesis is that some errors violate invariants, that enforcing violated invariants can correct the effects of these errors, and that correcting these effects can change the execution of the application to eliminate the corresponding failure. The goal is to find a patch that corrects the execution after the first error has occurred but before any effects have propagated far enough to make a failure inevitable. To accomplish this goal, ClearView finds and corrects errors that occur early enough in the execution to make the execution feasible via the invariant-enforcement mechanism.

5. **Candidate Repair Evaluation**: A candidate repair patch might have no effect, or even a negative effect, on the patched application. Thus, ClearView evaluates patches by continuously observing patched applications as they run. It ranks each patch based on whether the application fails or crashes when the patch is in place. This helps ClearView minimize the likelihood of negative effects by applying the most highly ranked patch. The fact that patches affect the execution only when a correlated invariant is violated also tends to minimize the possibility that they will negatively affect normal executions.

ClearView is designed around the concept of learning from failure and success. This helps the quality of ClearView’s patches to improve over time, similarly to a biological immune system. Moreover, developers may use ClearView to find and eliminate a bug in the source code responsible for a failure. ClearView supports this by providing information about the failure, specifically the location where it is detected, the correlated invariants, the strategy that each candidate repair patch used to enforce the invariant, and information about the effectiveness of each patch.

### 4.2.5 Model-Checking Based Techniques

This section presents techniques based on model checking in order to infer possible fix for a fault.

In [127] the authors present AutoFix-E, an automatic code fixing approach based on contracts. This approach considers contract violations as failures and calls existing functions whose postcondition fulfills the violated contract. Therefore the initial violated contract now holds and the execution can be pursued.

AutoFix-E works as illustrated in Figure 4.6. It considers Eiffel classes and generates test cases for these classes, based on the AutoTest tool (see e.g. [85]). Then predicates of a class (i.e. combinations of functions with no arguments and returning a boolean value) are used to classify object states of the program. Then an analysis of the predicates is
performed using the Daikon tool (see e.g. [38]) which provides fault profiles, i.e. the set of invariants predicate of the system that hold in the test cases that pass the test but do not hold in the failing ones. Then the authors use the Pachika tool (see e.g. [29]) in order to generate behavior models from the error-free runs. Fix candidates are the created from a set of fix templates (or fix schema) and the behavior models. Finally, these possible fix candidates are validated using test cases again. The fix candidates for which all the tests are successful are retained.

In [128], the authors propose an approach for automatically generating patches that are included into bug reports. Their motivation lies in the fact that bug-finding tools usually generate numerous alarms about possible bugs and that the ratio between the number of alarms produced and the number of corresponding bugs that are fixed should increase. In order to achieve this, the authors propose to add possible patches to bug reports as a means for increasing the likeliness of developers addressing these bugs.

The approach considered in [128] to automatically generate patches relies on model-checking techniques and focuses on bugs that violate predefined safety-policies. When a policy is analysed and its violation is detected, these analyses can produce a backtrace (i.e. counter-example) that violates this policy. From this backtrace, the authors employ a technique which generated a new trace that will fulfill the policy. They consider a state machines derived from the violated safety-policy. Variations of the violating backtrace is model-checked against this state machine. These variations correspond to deletions and additions of method calls to the violating backtrace so that the safety-policy is fulfilled by the resulting trace. Finally, the solution is mapped back to the source code and a textual patch is then added to the corresponding bug report. These patches may provide an appropriate solution or at least help locating the cause of the bug. Such patches can then be used as an alternative way of understanding what went wrong in the code.

4.2.6 Other Approaches

[26] describes StackGuard, an addition to gcc (i.e. a C compiler) which avoids buffer overflow vulnerabilities. Buffer overflow attacks usually consist of overflowing a buffer so that code can be added to the execution stack after the buffer has been filled and then
returning the address where that code lies. StackGuards prevents this type of attacks to perform any damage by preventing the execution of a compiled function to return a change address (by detecting the address change) or by preventing the write to the function return address. StackGuard allows for both approaches.

In [77], the authors present AutoPaG, an approach for facilitating patch generation for system vulnerability issues. AutoPaG addresses primary buffer overflow vulnerabilities as well as boundary condition errors. When an exploit is executed, AutoPaG is able to detect out-of-bound violations. Then using data flow analysis techniques, the system is able to locate the program statement that introduces the vulnerability. Finally AutoPaG takes inspiration of the manual process of generating this type of patch and automatically replaces statements of the form “strcpy(p, argv[1])” by statements of the form “strncpy(p, argv[1], 4)”, which avoid buffer overflow vulnerability.

4.2.7 Summary and Conclusion

Table 4.1 summarizes the different works on automatic repair presented in this section. These works are categorized according to the technique they use (e.g. Rollback, Genetic Programming and Mutation, etc) and the table also points out the tools and approaches that these work use or define. Finally, Table 4.1 indicates for these works what type of failures their approach makes it possible to correct.

First Table 4.1 shows that a non neglectable amount of works has been conducted in the area of “Automatic Repair” for software system. It is also important to note that although some of these works can be quite theoretical (e.g. [127]), tools are often provided. Finally, the types of faults that are targeted by these works is presented in the last column of Table 4.1. It shows that a lot of the proposed techniques deal with security issues and vulnerability fixes. Another type of faults is related to requirement violation, whether these requirements hold on the data or the invariants of the software. Finally, some works propose approaches that can be applied to general faults. These approaches usually rely on some knowledge regarding previously encountered and fixed faults.
## D6.1: State-of-the-art in Self-Healing and Patch Generation

<table>
<thead>
<tr>
<th>Platform Technology</th>
<th>Targeted Application Technology</th>
<th>Approaches and Tools</th>
<th>Fault Type</th>
<th>Technology</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx</td>
<td>General</td>
<td>Rollback</td>
<td>Data constraints violation</td>
<td>n/a</td>
<td>[95]</td>
</tr>
<tr>
<td>ASSURE</td>
<td>General</td>
<td>Genetic Programming and Mutation</td>
<td>Any fault revealed by a test case</td>
<td>n/a</td>
<td>[33, 81]</td>
</tr>
<tr>
<td>AbF</td>
<td>General</td>
<td>Event Filtering</td>
<td>Vulnerability violation</td>
<td>n/a</td>
<td>[129]</td>
</tr>
<tr>
<td>Juzi</td>
<td>General</td>
<td>Learning and Probabilistic</td>
<td>Buffer overflow attack</td>
<td>C</td>
<td>[110]</td>
</tr>
<tr>
<td>SHIELDGen</td>
<td>General</td>
<td>Learning and Probabilistic</td>
<td>Buffer overflow attack</td>
<td>n/a</td>
<td>[37]</td>
</tr>
<tr>
<td>PASAN</td>
<td>General</td>
<td>Model-Checking based</td>
<td>Buffer overflow attack</td>
<td>n/a</td>
<td>[27]</td>
</tr>
<tr>
<td>FLIPS</td>
<td>General</td>
<td>Others</td>
<td>Buffer overflow attack</td>
<td>C</td>
<td>[113]</td>
</tr>
<tr>
<td>BugFix</td>
<td>General</td>
<td>Others</td>
<td>Buffer overflow attack</td>
<td>n/a</td>
<td>[78]</td>
</tr>
<tr>
<td>ClearView</td>
<td>General</td>
<td>Others</td>
<td>Buffer overflow attack</td>
<td>n/a</td>
<td>[63]</td>
</tr>
<tr>
<td>Exterminator</td>
<td>General</td>
<td>Others</td>
<td>Safety policy violation</td>
<td>C</td>
<td>[94]</td>
</tr>
<tr>
<td>AutoFix-E</td>
<td>Contract</td>
<td>Others</td>
<td>Buffer overflow attack</td>
<td>C</td>
<td>[90]</td>
</tr>
<tr>
<td>StackGuard</td>
<td>Safety policy violation</td>
<td>Others</td>
<td>Buffer overflow attack</td>
<td>C</td>
<td>[127]</td>
</tr>
<tr>
<td>AutoPaG</td>
<td>StackGuard</td>
<td>Others</td>
<td>Buffer overflow attack</td>
<td>Linux</td>
<td>[28]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(out of bound)</td>
<td></td>
<td>[77]</td>
</tr>
</tbody>
</table>

Table 4.1: Summary on Automatic Repair techniques
5 Conclusion

This document introduces self-healing approaches as well as other research areas that are very related to this concept, such as fault-tolerance, Automatic Diagnosis and Automatic Repair. The latter contains works related to automatic patch generation.

The authors of [108] point out that self-healing is still a relatively immature field and that the class of faults tackled by this field remains quite narrow. This document shows that some of these faults are vulnerability attacks, concurrency and functionality issues as well as performance issues. It also shows that although initially inspired from fault-tolerance approaches, based on redundancy, the self-healing community is now moving away from these techniques and considering others such as model checking, machine learning, workarounds, etc.

Moreover, in parallel to this field, a lot of work have been conducted in the past recent years in the area of automatic diagnosis and automatic repair/bug fixing. This document presents a current view of these fields. It shows that quite a lot of attention has been brought to vulnerability issues and that there is still a long way to go before automatic solution for fixing bugs introduced by developers, but only discovered at runtime, can be delivered.

Regarding the challenges that these areas face, as pointed out in Section 2.1.4, developers and users intentions play a very important role in determining if part of a program implementation corresponds to a bug. One important challenge for self-healing and automatic diagnosis and bug fixing consists of introducing knowledge about developers and users intention into the system itself. Of course, this knowledge would need to be represented in a way that allows for automatic processing and reasoning.

It is therefore reasonable to assume that humans will be involved in the maintenance process for some more time as fixing bugs seems to require understanding and creativity that machines cannot mimic very well yet. Therefore at a shorter term, another important challenge resides in acquiring means for quickly and automatically removing symptoms of a fault from a system, without introducing new ones. This idea is related to the one of fault masking (see e.g. [73, 17] ) that does not correct faults in a system but removes its symptoms, giving help and time for the developers to fix the actual root cause of the issue.
Bibliography


