Appl Intell
DOI 10.1007/s10489-007-0047-1

Achieving self-healing in service delivery software systems by means of case-based reasoning

Stefania Montani · Cosimo Anglano

Received: 7 December 2006 / Accepted: 6 April 2007

Abstract Self-healing, i.e. the capability of a system to autonomously detect failures and recover from them, is a very attractive property that may enable large-scale software systems, aimed at delivering services on a 24/7 fashion, to meet their goals with little or no human intervention. Achieving self-healing requires the elicitation and maintenance of domain knowledge in the form of ⟨service failure diagnosis, repair plan⟩ patterns, a task which can be overwhelming. Case-Based Reasoning (CBR) is a lazy learning paradigm that largely reduces this kind of knowledge acquisition bottleneck. Moreover, the application of CBR for failure diagnosis and remediation in software systems appears to be very suitable, as in this domain most errors are re-occurrences of known problems. In this paper, we describe a CBR approach for providing large-scale, distributed software systems with self-healing capabilities, and demonstrate the practical applicability of our methodology by means of some experimental results on a real world application.

1 Introduction

The inherent complexity, heterogeneity, and dynamism of today's large-scale networked applications and services makes inappropriate, if not impossible, the traditional human-centered approach to system administration [1]. As a result, the attention of the industrial and academic communities has been driven towards novel solutions allowing the design and the implementation of self-managing computer systems.

The *Autonomic Computing paradigm* [1, 2], inspired by the human autonomic nervous system, has been recently proposed as an approach for the development of computer and software systems and applications that can manage themselves in accordance with high-level guidance from humans. An Autonomic Computing Systems (ACS) is composed of (one or more) *managed elements*, whose behavior is controlled by an *autonomic manager*, which applies suitable policies in order to automate the process of system management. In particular, in order to be able to self-manage, an ACS needs to exhibit the so called *Self-** properties:

- *Self-configuration*, that is the ability of the ACS to (re)configure itself to react to varying and unpredictable conditions in its operational environment;
- *Self-optimization*, that is the ability of the ACS to detect and to optimize suboptimal behaviors;
- *Self-protection*, that is the ability of the ACS to protect itself from both external and internal attacks;
- *Self-healing*, that is the ability of the ACS to detect problems and/or failures and to recover from them.

Self-healing, in particular, is extremely attractive for large-scale software systems, comprising a suite of applications and the computation/communication infrastructure on which they are executed, aimed at delivering services on a 24/7 basis (henceforth referred to as *Service Delivery Systems* (SDS)), as for instance those described in [3, 4]. The very large size of SDS (that may typically include from hundreds to thousands of machines), and the adoption of customized application software and middleware, makes service failures (i.e., failures in delivering their intended services) relatively frequent, because of the occurrence of

S. Montani (✉) · C. Anglano
Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria, Italy
e-mail: stefania.montani@unipmn.it

C. Anglano
e-mail: cosimo.anglano@unipmn.it

one or more faults in the applications they use, in the software/hardware infrastructure exploited to execute these applications, or in both. Devising suitable self-healing solutions, able to properly address all these faulty scenarios, is a very relevant objective, that is giving birth to a significant number of scientific investigations [5–10].

According to the Autonomic Computing paradigm, a self-healing system must be able to:

1. *monitor* its own behavior in order to detect service delivery failures;
2. *analyze* these failures in order to diagnose the faults causing them;
3. *plan* proper fault remediation strategies, and
4. *execute* these plans in order to restore the normal behavior of the system.

These four steps give birth to the so-called *self-healing cycle*, that is continuously performed by the autonomic manager controlling a managed element.

An existing SDS (i.e., the managed element in the Autonomic Computing terminology) may be rendered able to self-heal by integrating it with a *self-healing infrastructure* (i.e., the autonomic manager) which, as shown in Fig. 1, controls its behavior by logically or physically "surrounding" it with a closed control loop. In particular, the analyze and plan activities (grouped by the *self-healing engine* box in the figure) constitute the core phases of the process.

There are two possibilities for achieving such a closed-loop control. The first one consists of engineering the self-healing behavior directly into the components of the system, for instance by using an environment like Autonomia [11], that provides the necessary mechanisms and tools to perform the integration. The second alternative, known in the literature as *externalization* [8], consists instead of using a self-healing infrastructure totally external to the software system that interfaces with it. Externalization is particularly suited for retrofitting self-healing capabilities into existing systems since, being the self-healing infrastructure external to the system itself, it requires little or no modifications to existing applications. Moreover, and perhaps more importantly, externalization allows the development of generic self-healing infrastructures that can be used with any SDS, provided that suitable interfacing mechanisms are available. Therefore, in this paper we will follow this second approach.

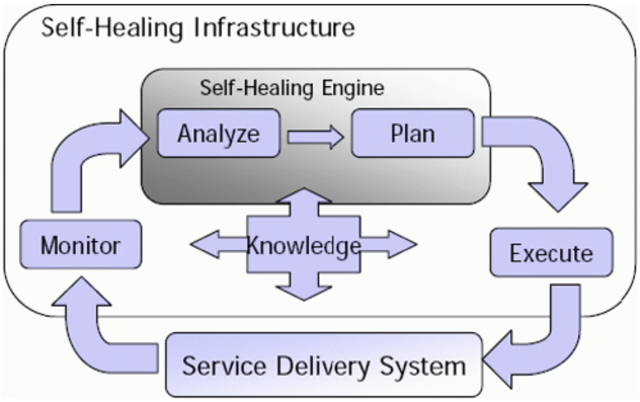


Fig. 1 A self-healing software system

Providing an SDS with self-healing capabilities requires the availability of specific *knowledge* (see Fig. 1) about the ⟨service failure diagnosis, repair plan⟩ patterns that may apply to the system at hand. Formalizing this kind of knowledge, for instance by means of rules or models, is a difficult and time consuming task, and a periodic revision of the knowledge base is required to always keep it up to date. These requirements are clearly in conflict with the goal of making the system as much as possible independent of human intervention.

The Case-Based Reasoning (CBR) methodology [12] is known to be well suited for those domains where formalized and widely recognized background knowledge is not available. CBR is a problem solving paradigm that utilizes the specific knowledge of previously experienced situations, called cases. It basically operates by *retrieving* past cases that are similar to the current one and by *reusing* past successful solutions after, if necessary, *revising* (i.e., *adapting*) them; the current case can then be *retained* and put into the system knowledge base, called the *case library* or *case base*. The retrieve, reuse-revise and retain procedures are known as the steps of the *CBR cycle* [12] (see top part of Fig. 2 in Sect. 3 for an implementation).

CBR thus allows one to build a knowledge base of past cases, which represent an "implicit" form of knowledge, that can be reused in present problems. By the term "implicit" knowledge we refer to an unstructured, operative type of knowledge, which directly stores the ⟨problem, solution⟩ patterns that have occurred in time "as they are", without any effort in the direction of extracting generalized or more abstract information (e.g., of eliciting rules or models, referred to as "explicit" or "structured" knowledge henceforth) from them.

Representing a real-world situation as a case is often straightforward: given a set of meaningful features for the application domain, it is sufficient to identify the value they assume in the situation at hand; typically, a case also stores information about the solution applied and, sometimes, about the outcome obtained. Due to the simplicity of this process, in many real world examples the knowledge acquisition bottleneck can be significantly reduced in comparison with the exploitation of other reasoning methodologies. New knowledge is also automatically stored in the case base during the normal working process; as the case library grows, more and more representative examples are collected, and it becomes easier to find a proper solution to a new problem by means of this paradigm. Furthermore, CBR seems particularly suited for failure diagnosis and remediation in software systems, in general, and in SDS in particular, as in this domain most errors are re-occurrences of known problems [6, 7, 13]; the methodology also provides a unique framework in which failure diagnosis and remediation are performed jointly.

In this paper we describe a CBR approach to support self-healing in (possibly large and distributed) SDS, in which case-based retrieval supports service failure diagnosis, while the reuse-revise step supports remediation. Case-based learning takes place to derive general knowledge from the case library content, and to control the case library growth. This contribution extends the ideas described in [5, 14], and further tests them by means of a thorough experimental evaluation, that has been carried out on a testbed implemented by means of Cavy, a tool we developed in order to support the deployment and operation of testbeds tailored to self-healing infrastructures, which will be introduced later in the paper.

The rest of the paper is organized as follows. In Section 2 we present related work about self-healing achievements in software systems. Section 3 details the CBR approach to self-healing, while Section 4 provides some experimental results, obtained on a testbed built using Cavy. Finally, Section 5 concludes the paper and outlines future research work.

2 Related work

Various approaches to fault diagnosis and remediation have been proposed in the literature. Sterrit [10] describes an approach to fault diagnosis based on *event correlation*, where various *symptoms* of system malfunctions (represented by alarms triggered by the various system components that are collected during the monitoring phase) are correlated in order to determine the (set of) fault(s) that have occurred. An alternative approach is proposed by Garlan and Schmerl [8], where fault diagnosis is performed by means of a suitable set of models. Joshi et al. [15] use Bayesian estimation and Markov decision processes to probabilistically diagnose faults, and use the results to generate recovery actions. Littman [9] proposes *cost-sensitive fault remediation*, a planning-based technique aimed at determining the most cost-effective system reconfiguration able to bring the system back to full functionality. Planning is also the basis for the fault remediation strategy proposed by Arshad [6]. Other very interesting applications of model-based diagnosis to automated software debugging and hardware design are described in [16–18]. Finally, in the same area we also mention the WS-DIAMOND project (http://wsdiamond.di.unito.it), where model-based diagnosis, planning and configuration techniques are resorted to in order to handle self-healing for Web services.

However, the main drawback of these approaches is that they require the availability of formalized and widely recognized background knowledge (i.e., structured knowledge) about the structure and/or the behavior of the system. For instance, planning-based techniques require a description of the domain, the states, and the correct configurations of the system, while event correlation and model-based diagnosis require the availability of a model describing how the various system components interact with each other. Despite the fact that structured knowledge may sometimes be available in practice, this is not always the case. Moreover, unfortunately, as we already observed, a significant effort is usually required to build, maintain, and use structured knowledge, with the consequence that its applicability to large-size systems, exhibiting complex behaviors and interactions among their components, may be problematic, at least in some situations.

CBR appears to be a suitable methodology for reducing the knowledge elicitation bottleneck, and its exploitation would enable one to be as general as possible, and to face also situations in which structured knowledge cannot be easily relied upon. Nevertheless, to the best of our knowledge, the only investigation in this direction has been published in [7], where a case-based retrieval system for discovering software problems without requiring human intervention is presented. This work, however, is quite limited, as it consists of a pure retrieval systems, in which the other steps of the CBR cycle are ignored, and problem solution is not provided. Case-based retrieval has also been resorted to in [19] for fault prediction (but not for fault remediation).

Another drawback of the literature approaches is their "fault orientation", that is they are triggered by individual component faults. Consequently, they attempt to correct a fault as soon as it is diagnosed (*preventive maintenance* [20]), even if it is not (yet) causing any service disruption because it is *dormant* [20], or it has been masked by the fault-tolerance techniques embedded into the system. Devising a repair plan for a fault that can be masked by the system is a waste of resources, and the same holds true for a dormant fault that, when it occurs, can be masked as well. Furthermore, from the perspective of service delivery, a dormant or masked fault has little or no importance until it causes a service failure (i.e., it becomes *active* [20]). While pro-active repair of dormant faults can be important in physical systems, for software systems it is much less important, as an unnoticed bug or a misconfiguration (which are, by definition, dormant faults) may never turn into an active fault causing a service failure. For instance, an unnoticed bug may be corrected as a side effect of a software update performed to fix another problem. Moreover, the correction of dormant or masked faults requires the availability of a model of the system, which brings us back to the problem of acquiring structured knowledge mentioned before. In our approach, self-healing will thus be activated just by service failures.

Finally, the proposals discussed before typically either address fault diagnosis or fault remediation, but they do not address both issues at the same time. Our work therefore appears to be a significant contribution in the recent literature panorama.

3 A CBR approach to self-healing

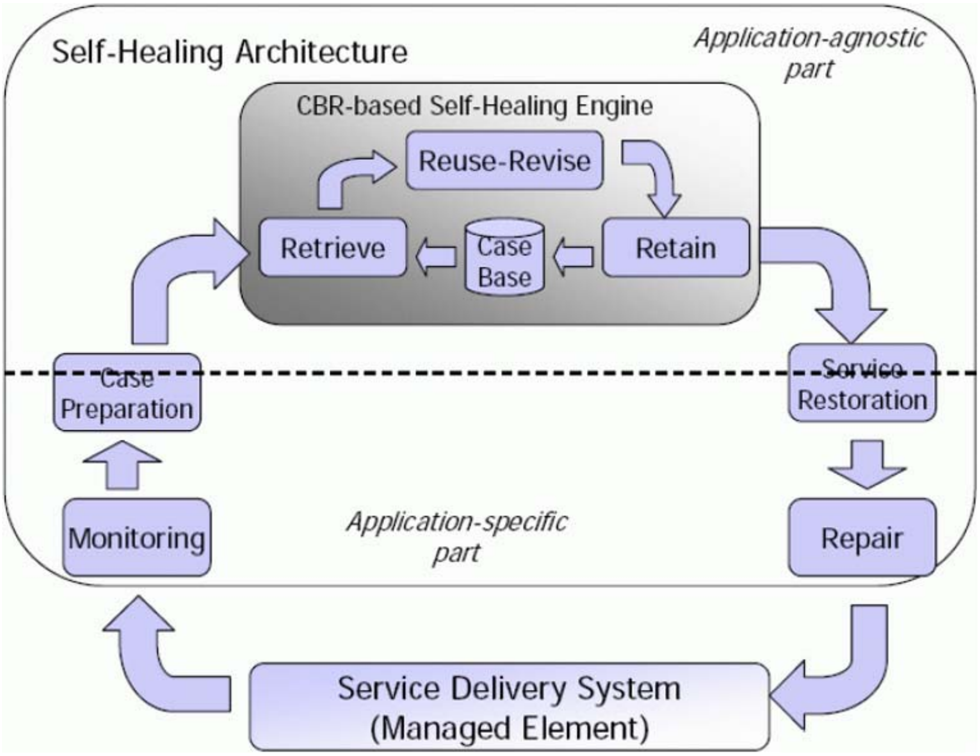
As anticipated in the Introduction, our self-healing approach is based on externalization, by which the SDS that has to be made able to self-heal is surrounded by an external self-healing infrastructure, carrying out the four steps of the self-healing cycle. More specifically, we propose a *self-healing architecture* (schematically depicted in Fig. 2), in which the SDS (i.e., the managed element) is treated as a "black box", surrounded by a set of external modules that form a closed-loop controlling the "health" of the system itself, and performing proper repair actions in case of service failures. As indicated in Fig. 2, our architecture relies on a CBR-based *self-healing engine*. The choice of CBR is motivated by the observation that the CBR cycle fits very well into the self-healing cycle, since it naturally covers the *analysis* and *planning* phases by means of the *retrieval* and *reuse-revise* steps, respectively; moreover, the *knowledge* used by the autonomic manager is contained in the case base, properly maintained by applying the *retain* step policies. As shown in Fig. 2, the CBR-based self-healing engine has an internal structure made up of three modules, each one implementing a step of the CBR cycle, namely:

- the *Retrieve* module, which retrieves past cases similar to the query one, thus implicitly providing a set of possible diagnoses for the current problem;
- the *Reuse-revise* module, which establishes what is (are) the candidate(s) solution(s) to the input problem, given the list of retrieved ones;
- the *Retain* module, which defines some policies about the opportunity of maintaining the current ⟨problem, solution⟩ pattern in the case base, possibly after a proper summarization strategy has been applied.

The other two phases of the autonomic cycle (i.e., *monitoring* and *execution*) are covered by a few additional specific modules (see Fig. 2), namely:

- the *Monitoring* module, which detects a service failure by properly probing the services provided by the SDS;
- the *Case Preparation* module, which collects the set of lower level symptoms (i.e., faults) that may have led to the detected service failure, and maps them to the case structure. In this way, it properly defines a new case, which can be used as a query case by the CBR self-healing engine;
- the *Service Restoration* module, that converts the solution(s) into a (set of) repair plan(s);
- the *Repair* module, which finally executes the repair plan corresponding to the selected solution, by using suitable application-specific mechanisms.

Fig. 2 CBR approach to self-healing


Self-Healing Architecture
Application-agnostic part
CBR-based Self-Healing Engine
Reuse-Revise
Retrieve
Case Base
Retain
Case Preparation
Service Restoration
Monitoring
Application-specific part
Repair
Service Delivery System (Managed Element)

As shown in Fig. 2, the external modules can be classified either into *application-specific*, i.e., that must be tailored to the specific characteristics of the SDS, or *application-agnostic*. While the monitoring and repair activities require the availability and usage either of knowledge and mechanisms provided by the running system or of adapted third-party components [21] (hence their classification as application-specific), the problem resolution activity (performed by the CBR modules) does not rely upon any particular system feature (although an adaptation strategy in the revise phase could require specific domain knowledge). The *Case Preparation* and the *Service Restoration* modules do not fall in just one of the two categories, since they act as an interface between the two layers.

In the remainder of this Section we will discuss the methodological choices taken in the design of our CBR-based self-healing infrastructure (i.e., the application-agnostic part), after a discussion of the hypotheses on which this design is based. The presentation of the application-specific modules is instead postponed until Sect. 4, where also an example is used to illustrate how the whole self-healing cycle described in Fig. 2 operates in a real case.

3.1 Assumptions

In this work we focus on SDS systems which verify the following assumptions:

- **Assumption 1:** the application of a (possibly incorrect) solution generates neither a new fault, nor a change in the system inner state (e.g., a change in some parameter configuration). Therefore, it is never required to completely revise the right repair action to deal with a new and/or unpredicted situation, introduced by an incorrect action already taken by mistake; it is also not required to explicitly "undo" the incorrect action itself. On the other hand, the same fault might generate different service failures (which rely on the same resource, made unavailable by the fault itself);
- **Assumption 2:** explicit time deadlines for producing the case solution are not fixed;
- **Assumption 3:** cases do not exhibit multiple solutions (i.e., the case solution is always univocally defined). Nevertheless, observe that our adaptation strategy would easily cope with an extension removing this assumption (see Section 3.2.2);
- **Assumption 4:** *transient faults* (i.e., faults that appear only once, and never again) do not take place; in fact, it seems to us a minor and reasonable simplification to ignore faults which, by definition, do not show up again after the first time;
- **Assumption 5:** *intermittent faults* (i.e., faults that appear and disappear in a totally unpredictable fashion) will not take place as well; this is a strong assumption, on the other hand, but intermittent faults represent a very complex problem, which is still seen as an open research issue in the literature: therefore, by now we will not try to cope with a problem for which a recognized solution/management strategy has not been indicated yet.

The class of applications for which these assumptions hold, although not including all the possible SDS systems, is still large enough to include many real world examples of Web-based applications used to deliver a variety of services accessible through the Internet. Therefore, our approach can be profitably used "as is" for a reasonably large class of relevant applications. Relaxing some of these assumptions to broaden the class of targeted SDS will be part of our future work.

3.2 Methodological choices in the CBR system

As depicted in Fig. 2, our CBR-based self-healing engine includes three components, each one performing one or more steps of the CBR cycle. The development of these components required us to take a few methodological decisions, that are discussed below.

3.2.1 Case representation and retrieval

In SDS, each case represents a service failure episode, whose features are the corresponding symptoms (i.e., faults) discovered by the Case Preparation module. On the other hand, the case solution is composed by the restoration procedure applied in that situation.

In order to retrieve the most similar (i.e., less distant) cases with respect to the input one, a measure of distance in the features space has to be provided. Generally speaking, the distance between two cases can be computed as a (weighted) average of the normalized distances between their various features (where weights can be properly set to state that some features are more "important" for retrieval relatively to the others). The simplest approach requires to set all the weights to 1 (as we did in our experiments), but more discriminating weighting policies can be used as well [22], and will be part of our future investigation.

Various metrics can be relied upon to calculate the distance; we are currently using the heterogeneous euclidean-overlap metric (HEOM) [23], a distance metric able to treat both symbolic and numeric variables, and to cope with the problem of missing data. If f is a feature, HEOM is defined as follows:

$$\mathrm{HEOM} = \sqrt{\sum_f d_f(x,y)^2}$$

where $d_f(x,y) = 1$, if x or y are missing; $d_f(x,y) = overlap(x,y)$ if f is a symbolic feature, (i.e., 0 if $x = y$, 1 otherwise); $d_f(x,y) = \frac{|x-y|}{range_f}$ if f is a linear feature.

In case of symbolic features, the *overlap* function can be generalized with the use of a similarity table.

3.2.2 Reuse-revise

We have currently adopted a very simple form of *adaptation*, which requires to exploit all the pertinent past solutions (i.e., remediation strategies) for a given case, by applying all of them in sequence, until the service is restored (and all the remaining unapplied solutions can then be discarded). This policy also fits the not infrequent situation in which the query case is a combination of two or more past ones (i.e., two different service failures have taken place simultaneously, or the same service failure has been caused by two faults, whose symptoms are described as feature values).

The order in which solutions should be applied, and the possible need to "undo" the effect of an already tested repair plan before applying another, are application-dependent aspects, that should be provided as an initial guideline by the human experts. As a default, we simply order the retrieved cases on the basis of their distance from the input case, and we reuse up to all the solutions, without revising them. Actually, the application of one solution after the other (with no "undo" mechanisms) can be seen as a very simple revision strategy, which is indeed suitable for applications which fit *Assumption 1* in Sect. 3.1, i.e., for which no side-effect is foreseen after the implementation of a non-working solution. How to deal with situations in which such hypothesis does not hold will be object of our future investigation.

On the other hand, if a certain case admits multiple solutions (see *Assumption 3* in Sect. 3.1) for the service failure it embeds, then this adaptation strategy allows one to solve the problem by means of the first (i.e., less distant case's) pertinent retrieved solution, among the existing alternative ones.

Finally, observe that, if the system cannot retrieve any useful solution from its case base, the situation is handled by asking a direct intervention by human experts (see however the discussion in Sect. 3.2.3), a possibility we are allowed to take because we have not fixed a time deadline to repair the service failure (see *Assumption 2* in Sect. 3.1).

3.2.3 Retain and case-based learning

In order to behave as a system really able to self-heal, our infrastructure must be able to work as much as possible in an autonomous way, i.e., without human intervention. However, this can happen only if a case base containing enough instances of solved cases is available, but in general this does not hold true when the system is initially put into operation. We therefore envision a *bootstrap phase*, enabling the collection of the initial cases, during which the problem solution is provided by humans.

Background knowledge is needed also when the case base does not contain examples similar to the query case (due to the presence of *competence gaps* within the case base itself). In this situation, human intervention could be required on the fly.

However, it is worth noting that:

1. competence gaps tend to be reduced while the system is being used, since CBR is a lazy learning paradigm which automatically collects new knowledge in an implicit form;
2. if a reasonable amount of explicit knowledge is available in the beginning, it might be somehow integrated with the CBR system (especially in the revision phase), in order to improve the remediation strategies definition (see also Sect. 5).

As far as more and more cases are stored in the case base, it may be needed to properly summarize the collected cases, to control the case base growth. In particular, we have adopted a strategy that derives suitable "prototypes" (see for instance [24, 25]), able to summarize the information carried by the ground cases they represent, that are stored in place of the cases themselves.

Prototypes are a generalization from single to clustered typical cases. The main purposes of such a generalization knowledge are to:

- structure the case base;
- decrease the storage amount by erasing redundant cases;
- guide and speed-up the retrieval process.

In particular, the periodic reorganization of the case base to learn or update the prototype definitions, by taking into account the new acquired cases, automatically allows one to delete the redundant or useless ground cases. An evaluation of each newly acquired case is therefore not needed with this strategy.

Moreover, as observed in the Introduction, in domains with rather weak domain theories (like the SDS one), CBR can provide an easy way of acquiring knowledge. Anyway, gathering new cases may implicitly improve the system competence, but does not elicit the intrinsic knowledge of the stored cases. To learn the structured knowledge contained in cases, a generalization step is required. At least in domains where experts use to reason by exploiting prototypical and exceptional cases, the creation of prototypes seems to be an adequate learning technique. In these domains, prototypes can be seen as a form of knowledge that fills the gap between specific cases and general rules.

To some extend, it is thus possible that generalized (i.e., structured) knowledge about the system is learnt from the case base: in the applications where this goal is reached, it will be obviously possible to take advantage of the newly elicited knowledge (e.g., in the form of prototypes) through other reasoning techniques (e.g., rule-based reasoning or model-based reasoning), or to move in the direction of defining a multi-modal reasoning system, in which CBR will be just one of the methodologies relied upon (see also Sect. 5).

Finally, prototypes can at least partly help to solve the adaptation problem. The idea to cope with the adaptation task by generalizing, works for diagnostic tasks where abstracted typical cases represent diagnoses, and additional specific features of former single cases can be neglected. On such generalized cases the definition of an adaptation strategy becomes easier, since the specific details of ground cases leave space to a more general kind of knowledge. If a hierarchy of abstracted cases exists, adaptation can be seen as a top down search to find the most specific case that fits for the current problem. We will investigate this strategy in a deeper fashion in the future.

How prototypes can be extracted in a concrete situation is described in Sect. 4.

4 Experimental results

In order to validate our approach, as well as to test its effectiveness in providing self-healing behavior in SDSs, we implemented a prototype of the CBR-based self-healing engine (using Java as the programming language and MySQL to store the case base), and interfaced it with an SDS testbed. This testbed was deployed and operated by means of Cavy, a software system we designed and implemented as part of our self-healing project (see Sect. 4.1), and included a real distributed application delivering an EBay-like auction service. The resulting self-healing SDS was used to perform an extensive experimentation, in which service failures were caused (by injecting individual components with faults), detected, diagnosed and repaired autonomously. Our results indicate that the CBR-based self-healing engine we developed is able to provide suitable repair plans for the detected failures, and that the self-healing infrastructure built (by means of Cavy) after the architecture depicted in Fig. 2 can be effectively used to concretely implement the externalized approach to self-healing.

4.1 The Cavy tool

Cavy is a software system allowing the configuration, deployment, and operation of Service Delivery testbeds comprising a set of machines on which the various components of the applications delivering the services are executed. Cavy adds to these machines and application processes a set of modules that (a) inject individual testbed components with faults, (b) diagnose the resulting service failures, (c) interact with a self-healing engine in order to obtain a repair plan fixing these failures, and (d) perform the corresponding repair actions. More precisely, as schematically depicted in Fig. 3, in addition to the various machines (depicted as boxes) and the processes of the application delivering the services (depicted as circles labeled with AP), Cavy's architecture encompasses a set of *Fault Managers* (one for each machine of the testbed, and represented as rhomboids labeled as FM), a *Fault Injector*, a *Service Monitor*, a *Fault Diagnoser*, a *Fault Repairer*, and an *Integrator*. A Cavy testbed is driven by the *Fault Injector*, that at random points in time *injects* one (several) machine(s)/application process(es) with a (set of) fault(s). These faults usually induce the inability of the application to deliver one (or more) services in its intended fashion, that is they cause a service failure. The *Service Monitor* module continuously monitors the services delivered by the application, and in case of failures triggers the operation of the other modules of the testbed in order to obtain a (set of) solution(s) from the self-healing engine.

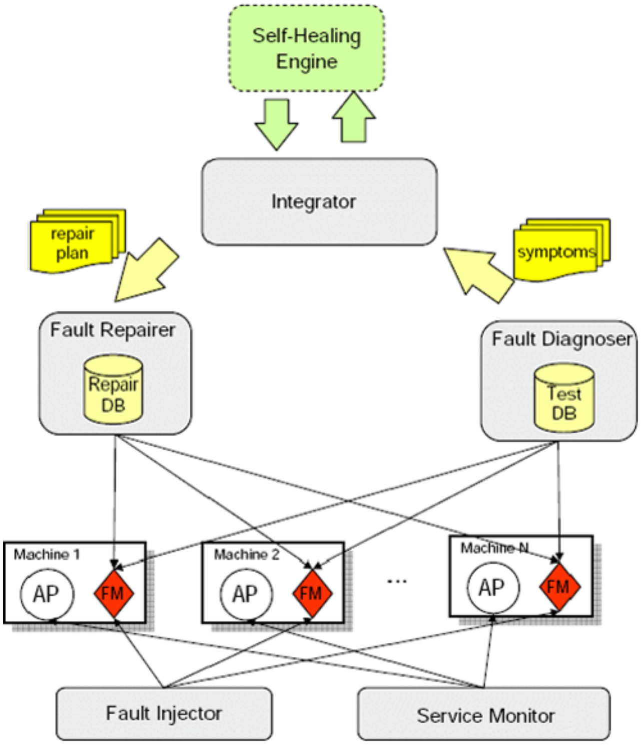


Fig. 3 The architecture of Cavy

Let us explain in more detail the operations performed by each testbed component, and how these components interact among them.

4.1.1 The fault managers

The basic components of a Cavy testbed are the Fault Managers, that are executed on all the machines of the testbed. Each Fault Manager is in charge of injecting the machine on which it runs (and/or the application component(s) running on that machine) with faults, as well as of repairing them.

Fault injection is performed by using user-level scripts (possibly requiring the superuser privileges) that induce faults by performing actions like machine reboots, network interface card disconnections, disk or file systems disconnections, file deletion/corruption, modifications to configuration files to emulate errors due to human operators, and suspensions/terminations of processes to emulate software crashes.

Each Fault Manager holds a database containing, for each fault, a textual description (e.g., network interface down or Web server crash) and the names of the scripts or executable programs that will cause and fix the fault. Cavy supports executable programs or scripts in any language directly executable on the machine on which the Fault Managers are run. For instance, a Unix machine can be injected with a fault like "disconnect the machine from the network" by executing the 'ifconfig eth0 down' command, that can be contained into a shell script, while the corresponding repair action corresponds to the execution of the 'ifconfig eth0 up' command. More complex fault or repair actions can be specified using a mix of executable programs and shell scripts.

Each Fault Manager continuously waits for a command to be sent either by the *Fault Injector*, the *Fault Repairer*, or the *Fault Diagnoser*. When a fail command is received from the *Fault Injector*, the Fault Manager searches its database for the specified fault, and executes the corresponding fault script. Analogously, when a repair command is received from the *Fault Repairer*, the Fault Manager finds in its database the name of the script fixing the specified fault, and executes it. This functionality of the Fault Manager thus corresponds to the one of the Repair module in our self-healing conceptual architecture (see Fig. 2). Finally, the Fault Manager can also be queried by the *Fault Diagnoser* for the list of *active faults* (i.e., faults that still have to be repaired).

4.1.2 The fault injector

As already anticipated, the operations of the testbed are driven by the Fault Injector that has the purpose of injecting the components of the testbed with faults. The Fault Injector uses a database containing information on the various faults it can produce, initialized at startup time by separately querying the various Fault Managers of the testbed. The Fault Injector operates as follows: (a) it randomly selects one of the faults in its database, (b) sends the corresponding fault command to the proper Fault Manager, (c) waits for a random amount of time (chosen according to a user-specified distribution) before injecting the machine/application at hand with the next one. If the *multi-failure* mode of operation is used, the Fault Injector injects the machine/application with the next fault immediately after the waiting time expires. If, conversely, the *single-failure* mode is chosen, the Fault Injector after waking up waits until the current service failure has been fixed.

4.1.3 The service monitor

The Service Monitor (corresponding to the Monitoring module of our self-healing conceptual architecture depicted in Fig. 2) continuously monitors the SDS in order to identify possible failures in the delivery of services. The Service Monitor runs asynchronously with respect to the other modules of the testbed, and polls (at a user-defined fixed frequency) the various services delivered by the application run on the testbed. When a failure is detected, the Service Monitor sends a notification message (containing the identity of the detected failure) to the *Fault Diagnoser*, and suspends itself until the failure has been fixed (see later sections).

The Service Monitor consists of a generic engine, that does not depend on the specific application being monitored, on which suitable methods aimed at testing the specific application services can be plugged in. Therefore, only these plugins need to be provided for a particular application run on the testbed.

4.1.4 The fault diagnoser

The Fault Diagnoser is the component of Cavy which is in charge of identifying the faults that have given rise to a specific service failure. As discussed before, the Fault Diagnoser is triggered by a message sent by the Service Monitor. Upon receiving this message, the Fault Diagnoser queries all the Fault Managers of the testbed for the list of active faults (that represent the *symptoms* of the detected failure), assembles a message containing this list, and sends it to the *Integrator*. The Fault Diagnoser corresponds to the application-specific part of the Case Preparation module of our self-healing conceptual architecture of Fig. 2.

4.1.5 The integrator

The Integrator is the Cavy module that corresponds to the application-agnostic parts of both the Case Preparation and Service Restoration modules of our self-healing architecture. In particular, it interfaces the Fault Diagnoser with the self-healing engine by receiving the list of diagnosed faults and by assembling from it the case (corresponding to the detected service failure) that is given as input to the self-healing engine. Furthermore, it interfaces the self-healing engine with the Fault Repairer by receiving from the former a list of solution cases (sorted in increasing order with respect to the distance from the input case), and by preparing and sending to the latter, for each of them, the corresponding *repair plan* (consisting of the set of faults that must be repaired in order to fix the detected service failure).

4.1.6 The fault repairer

The Fault Repairer is the component of Cavy that is in charge of carrying out the (set of) repair plan(s) devised by the self-healing engine, and corresponds to the application-specific part of the Service Restoration module of our self-healing architecture. The Fault Repairer executes the above plans in sequence (recall that they are sorted according to their distance from the input case) by sending the corresponding repair commands to the proper Fault Managers. As already observed, the Fault Managers are in charge of implementing the repair plan by converting it into the proper (application-specific) script commands. After each plan has been executed, the Fault Repairer tests the failed service in order to check whether it has been fixed or not. If the outcome is positive, it informs the Integrator, that retains the solution for the input case, while if it is negative the next plan in the sequence is executed.

4.2 The experimental testbed

In order to experiment with our self-healing methodology, we have concretely implemented the self-healing architecture of Fig. 2 by realizing a prototype CBR-based self-healing engine (called CaBaReT), and by coupling it with a Cavy-based testbed. CaBaReT consists of a Java-based front end, that interfaces itself with a relational DBMS (MySQL in the current implementation) managing the case base. Upon receiving a case corresponding to a service failure coming from an external source (the Cavy's *Integrator* module, in our case), CaBaReT sorts the cases in its case base according to their distance from the input case, and sends their solutions to an external module (again, the *Integrator*). We have used Cavy and CaBaReT to build a self-healing version of *Rubis* [26], an application delivering an auction service (modeled after EBay). Among the various configuration options Rubis allows one to choose, we picked the two-tier PHP-based one. The first tier consists of a machine, running an Apache Web server, providing a portal service using which customers may interact with the auction system (to perform actions like registering, selling items and bidding for them). The second tier consists instead of a machine, running a MySQL DBMS, that stores all the information concerning users, items, and bids (in three separate tables, named after the kind of information they store). Both the machines we used for our testbed were equipped with Intel Pentium IV processors and 512 MB of RAM, and ran the Linux operating system. It is worth to point out that Rubis, in spite of its apparently simple architecture, can be considered representative of a wide class of Web-based applications used to deliver a variety of services accessible through the Internet, that verify the assumptions in Sect. 3.1. Therefore, the results we obtained in our experiments with Rubis should hold also for the entire class of applications it represents.

In order to implement the Monitoring module of our prototype self-healing architecture, we wrote a set of plugins, using the C++ language and the cURL libraries [27], that test the various services delivered by Rubis. More specifically, the services we took into account are: Register (register a user), Browse (register an item before selling it, or browse items—by category or by region), Sell (sell an item), and Home (go to the initial Rubis page).

The Register activity requires to access the User table, while Browse requires to read/write the Item table, and Sell requires to access the Bid table. On the other hand, Home generates a static page, which does not need to access the database. Home is therefore a simpler request, while all the other services can be grouped in a single category (that we will call Cat1 in the experiments) characterized by the need to interact both with the Web server and the DBMS.

In this scenario, the following reasons for the occurrence of a service failure can be identified:

- Apache configuration problems took place (e.g., the wrong Web server port has been set, or the wrong permissions have been assigned to the documents directories);
- Apache network problems took place (i.e., the network connection of the machine on which Apache is running is down);
- it is necessary to restart Apache (since the Web server is down, or some configuration change has been operated);
- MySQL configuration problems took place (e.g., the wrong MySQL port has been set);
- MySQL network problems took place (i.e., the network connection of the machine running MySQL is down);
- it is necessary to restart MySQL (since the DBMS is down, or some configuration change has been operated);
- the sufficient privileges to access a database table (namely User or Item or Bid) have not been granted to the user.

Globally, we have thus identified 6 faults, plus 3 permission problems on specific database tables, that may affect the Rubis services. Therefore, we have built a case base in CaBaReT, implemented as a single DMBS table (but more complex database structures, such as a different table for every category of cases, sharing common features, could be easily adopted in case of need), where every case is described by the following features:

- **Failure** (a symbolic feature describing the observed service failure);
- **User** (a boolean feature stating whether the permissions to access the User table have not been correctly granted);
- **Item** (as above, for the Item table);
- **Bid** (as above, for the Bid table);
- **Apache Cf** (a boolean feature, explaining whether any configuration mistake for the Web server has been introduced);

- **Apache Net** (a boolean feature, stating whether the Apache network is down);
- **Apache** (a boolean feature, stating whether Apache is down);
- **MySQL Cf** (as for Apache Cf, but referring to the database);
- **MySQL Net** (as for Apache Net);
- **MySQL** (as for Apache);
- **Solution** (a symbolic feature, that will not be used to calculate case distance, explaining the repair plan - still not in terms of the script to be executed, since this translation will be operated by the Integrator and by the proper Fault Manager after the CBR self-healing architecture has provided its answer).

Referring to the Rubis application, and to this case structure, in the next Section we will describe our experimental results.

4.3 Experiments

In this section, we will describe the initial phase of our experiments in a step by step fashion; this will allow us to illustrate how the conceptual self-healing cycle presented in Fig. 2 operates. We will then summarize the results of our additional tests, and draw some conclusions.

We started by building an initial case base (bootstrap phase), in which four specific cases (reported in Table 1) where inserted by a human expert.

The system was then put into operation, and a new case was generated, due to the injection of the Rubis application with a fault. In particular, the Fault Injector randomly selected an Apache configuration problem, and sent the corresponding fail command to the Fault Manager running on the Apache machine, that injected it with the fault by using the corresponding script. After a short while, the Service Monitor of the *Cavy* testbed identified the Sell service failure, and triggered the Fault Diagnoser, that probed the Fault Managers of the two testbed machines and sent the list of faults it found to the Integrator. The Integrator then assembled the case reported in Table 2, and sent it to CaBaReT. Upon receiving this input case, CaBaReT computed the distances between it and all the cases in the case base, by means of the Retrieve module, as discussed in Sect. 3.2. The resulting distances, computed by giving equal weights to all the features, are reported in Table 3.

The less distant cases with respect with the new one are the third and the fourth (corresponding to Sell and Home service failures respectively). The Reuse-revise module applied both solutions, in the retrieval order. In particular the solution to the third case was useless, but the one to the fourth case (i.e., Fix Apache Config), implemented by the Apache Fault Manager, driven by the Integrator and by the Fault Repairer, fixed the new case problem. Observe that, despite the fact that the service failure experienced in the new case was different from some of the ones already stored in the table, by retrieving the most similar cases and by applying their solutions it was possible to correctly solve the case itself (in accordance with the second part of *Assumption 1* in Sect. 3.1). The new case was then kept by the Retain module (actually,

Table 1 Contents of the case base in the beginning

Failure	User	Item	Bid	Apache Cf	Apache Net	Apache	MySQL Cf	MySQL Net	MySQL	Sol.
Register	no	no	no	no	no	no	no	no	yes	Restart MySql
Browse	no	no	no	no	no	yes	no	no	no	Restart Apache
Sell	no	no	no	no	no	no	no	yes	no	Bring Db Net up
Home	no	no	yes	yes	no	no	no	no	no	Fix Apache Config

Table 2 The first automatically generated case

Failure	User	Item	Bid	Apache Cf	Apache Net	Apache	MySQL Cf	MySQL Net	MySQL
Sell	no	no	no	yes	no	no	no	no	no

Table 3 Distances between the first service failure and the four cases in the case base

Failure	User	Item	Bid	Apache Cf	Apache Net	Apache	MySQL Cf	MySQL Net	MySQL	All
1	0	0	0	1	0	0	0	0	1	3/10
1	0	0	0	1	0	1	0	0	0	3/10
0	0	0	0	1	0	0	0	1	0	2/10
1	0	0	1	0	0	0	0	0	0	2/10

Table 4 The second automatically generated case

Failure	User	Item	Bid	Apache Cf	Apache Net	Apache	MySQL Cf	MySQL Net	MySQL
Sell	no	no	no	no	no	yes	no	yes	no

Table 5 Distances between the second service failure and the five cases in the case base

Failure	User	Item	Bid	Apache Cf	Apache Net	Apache	MySQL Cf	MySQL Net	MySQL	All
1	0	0	0	0	0	1	0	1	1	4/10
1	0	0	0	0	0	0	0	1	0	2/10
0	0	0	0	0	0	1	0	0	0	1/10
1	0	0	1	1	0	1	0	1	0	5/10
0	0	0	0	1	0	1	0	1	0	3/10

Table 6 The third automatically generated case

Failure	User	Item	Bid	Apache Cf	Apache Net	Apache	MySQL Cf	MySQL Net	MySQL
Browse	no	yes	no	no	no	no	no	no	no

we initially applied a policy following which all new solved cases were retained).

The second failure that was generated in our experiment regarded again the Sell service, but this time it resulted from the simultaneous generation of two individual faults (namely, Apache and MySQL Net), that gave rise to the case reported in Table 4. As for the first service failure, the self-healing engine computed the distances of this case with respect to the cases in the case base (that now included also the first case, which had been retained), that are reported in Table 5. As a consequence of the fact that two distinct faults had been registered simultaneously in the input case, the application of the solutions of both the third and the second cases (which are the most similar ones) was required in order to fix the problem.

As a further step, the case in Table 6 was generated, and this time none of the existing cases (including the two newly solved and retained ones) proved to be able to fix the problem; human intervention was required, by manually editing the solution of the new case, i.e., to correct the permissions for the Item table.

The experiments continued in this fashion for about 3 days, and required only a few additional manual interventions (less than 10). At the end, 1016 cases were generated, most of which including multiple service failures, and all were automatically solved by CaBaReT.

At this point of the experiment, we evaluated the possibility of reducing the memory occupancy of the case base, by understanding if some more general knowledge could be learnt from the acquired cases, enabling the deletion of some very specific cases, adequately described by such a higher level information.

In particular, we applied the C4.5 algorithm [28] to learn a decision tree that, after we labeled all the database cases with their solution (same solution = same class), was able to select the most important features characterizing a class (i.e.,

Table 7 Contents of the prototypes case base

Failure	User	Item	Bid	Apache Cf	Apache Net	Apache	MySQL Cf	MySQL Net	MySQL	Sol.
NULL	NULL	NULL	NULL	yes	no	no	no	no	no	Fix Apache Config
NULL	NULL	NULL	NULL	no	yes	no	no	no	no	Bring Apache Net Up
NULL	NULL	NULL	NULL	no	no	yes	no	no	no	Restart Apache
Cat1	NULL	NULL	NULL	no	no	no	yes	no	no	Fix MySQL Config
Cat1	NULL	NULL	NULL	no	no	no	no	yes	no	Bring Db Net Up
Cat1	NULL	NULL	NULL	no	no	no	no	no	yes	Restart MySql
Register	yes	NULL	NULL	no	no	no	no	no	no	Fix User Perm.
Browse	NULL	yes	NULL	no	no	no	no	no	no	Fix Item Perm.
Sell	NULL	NULL	yes	no	no	no	no	no	no	Fix Bid Perm.

a prototype, see Sect. 3.2.3). More specifically, we used C4.5 release 8, available in source form from [29], that was compiled for a Linux platform. In order to use it for our purposes, we prepared a data file (using the format prescribed by C4.5) from a dump of the case base contents and containing the 1016 input cases, and ran C4.5 against it. The time taken by C4.5 to complete its execution on a machine equipped with a 1.7 GHz Intel M processor and running Linux was 19 seconds. At the end of its execution, C4.5 generated a tree having size 333, that was reduced to 74 after the C4.5 pruning operation was performed. The results produced by C4.5 indicated that the order of importance of the various features for classification was (in a decreasing fashion) **MySQL Cf**, **MySQL Net**, **MySQL**, **Apache**, **Apache Cf**, and **Apache Net**, while features **User**, **Item**, and **Bid** proved to be unimportant, except for very specific situations (see below).

Furthermore, the analysis of the pruned tree revealed us that those cases characterized by the simultaneous presence of multiple faults could be solved by applying, one after the other, the solutions of the individual faults. This corresponds to say that, in practice, the most complex cases are just a linear combinations of the simplest ones, whose solutions in turn can be mapped in a 1:1 way to the corresponding individual faults.

The results of this generalization procedure and of these considerations, expressed in the form of a database of prototypes, are listed in Table 7. Each of the 9 prototypes summarizes a set of ground cases, sharing the same values in a significant set of features (i.e., some faults/permissions strongly characterizing a specific solution). The values of the features considered as irrelevant for classification by C4.5 were set to NULL in the prototypes.

As it can be observed from the table, each of the Apache-related faults can independently cause each of the service failures, while the MySQL related ones can also independently cause each of the service failures in Cat1. Moreover, Register is affected by the User permission feature, while Browse is affected by the Item permission feature and Sell is affected by the Bid permission one.

In order to assess the ability of the above prototypes to correctly identify proper solutions to the occurred service failures, we performed a set of experiments in which more than 100 new failures were induced in the system and had to be solved by CaBaReT using a case base containing only the prototypes and not performing any retain. In all these experiments CaBaReT was able to provide a working repair plan, consisting of the sequential application of one or more solutions to individual faults, without requiring any human intervention. In the worst case, all 9 cases were retrieved, the first 8 solutions were applied without completely solving the problem, and only the 9th one finally proved to be the required one. The approach of retrieving all cases and of applying all solutions was feasible, since retrieval time was very low for the prototypes database (thanks to its limited size), and in the Rubis application the adoption of a wrong solution did not alter the system state (as prescribed by *Assumption 1* in Sect. 3.1). In the future, we will work at more complex solution adaptation and combination strategies, that might be more suitable for applications for which this simplification does not hold.

5 Conclusions and future work

In this paper we have presented a CBR approach for the achievement of self-healing in SDS that, unlike alternative solutions, avoids unnecessary repair actions. The repair procedure is indeed triggered by service failures rather than by individual component faults. Moreover, it does not require the availability of structured knowledge, such as models of the system behavior, thus easing its applicability to large-scale, complex software systems.

The suitability of this approach has been demonstrated by some tests conducted on Rubis, an on-line auction service application, running on a distributed architecture. For this purpose, we developed a proof-of-concept implementation of our CBR-based self-healing engine, and interfaced it with an ad hoc testbed we built using Cavy, a tool supporting the deployment and operation of testbeds tailored to self-healing infrastructures. Cavy allows to easily define testbeds with various characteristics. The results we obtained were quite encouraging, and showed both the feasibility of our approach, and its effectiveness in providing self-healing in real-world applications. It is worth to point out that the application we used can be considered as representative of a wide class of Web-based applications exploited to deliver a variety of services accessible through the Internet. Therefore, our results should hold also for the entire class of applications that Rubis represents (i.e., that satisfy the assumptions in Sect. 3.1).

In the future we plan to exploit Cavy for verifying our CBR approach to self-healing on additional real world applications. In particular, we plan to test the applicability of the current implementation to classes of problems that do not satisfy all the assumptions in Sect. 3.1, for instance in situations in which the application of an incorrect solution introduces new faults, or somehow modifies the system inner state, so that a proper "undo" policy, or solution selection mechanism, must be devised.

We will also analyze the possibility of resorting to prototypes to cope with the adaptation task by generalizing; such a policy may lead to the definition of a more suitable adaptation strategy with respect to the currently implemented one, for complex examples that do not fit all our working hypotheses (see Sect. 3.1).

Moreover, we also plan to test the advantages of our approach in applications in which some formalized background knowledge is or becomes available. Actually, CBR can be easily combined with other knowledge sources and with other reasoning paradigms, and is particularly well suited for integration with rule-based or model-based systems [30]. The interest in multi-modal approaches involving CBR is recently increasing through different application areas [31, 32], from planning [33] to classification [34] and to diagnosis [35], and from legal [36, 37] to medical decision support [24, 38, 39]. Our goal will be to demonstrate the further advantages of relying on two different methodologies, by tightly coupling them, or alternatively by just switching between one and the other, when the aim is to provide a software system with autonomic diagnosis and remediation capabilities.

References

1. Ganek AG, Corbi TA (2003) The dawning of the autonomic computing era. IBM Syst J 42(1):5–18
2. Kephart JO, Chess DM (2003) The vision of autonomic computing. IEEE Comput 36:41–50
3. Brewer E (2001) Lessons from giant-scale services. IEEE Internet Comput 5(4):46–55
4. Oppenheimer D, Patterson D (2002) Architecture and dependability of large-scale Internet services. IEEE Internet Comput 41–49
5. Anglano C, Montani S (2005) Achieving self-healing in autonomic software systems: a case-based reasoning approach. In: Czap H, Unland R, Branki C, Tianfield H (eds) Proceedings of the international conference on self-organization and adaptation of multi-agent and grid systems (SOAS), Glasgow, December 2005, IOS, Amsterdam, pp 267–281
6. Arshad N, Heimbigner D, Wolf A (2004) A planning based approach to failure recovery in distributed systems. In: Proceedings of 2nd ACM workshop on self-healing systems (WOSS '04), Newport Beach, CA, USA, October 2004. ACM, New York
7. Brodie M, Ma S, Lohman G, Syeda-Mahmood T, Mignet L, Modani N, Champlin J, Sohn P (2005) Quickly finding known software problems via automated symptom matching. In: Proceedings of the 2nd international conference on autonomic computing, Seattle, WA, USA, June 2005
8. Garlan D, Schmerl B (2002) Model-based adaptation for self-healing systems. In: Proceedings of 1st ACM workshop on self-healing systems (WOSS '02), Charleston, SC, USA, November 2002. ACM, New York
9. Littman M, Nguyen T, Hirsh H (2003) Cost-sensitive fault remediation for autonomic computing. In: Proceedings of IJCAI

workshop on AI and autonomic computing: developing a research agenda for self-managing computer systems, Acapulco, Mexico, August 2003

10. Sterrit R (2004) Autonomic networks: engineering the self-healing property. Eng Appl Artif Intell 17:727–739
11. Dong X, Hariri S, Xue L, Chen H, Zhang M, Pavuluri S, Rao S (2003) Autonomia: an autonomic computing environment. In: Proceedings of the 2003 international conference on performance, computing, and communications. IEEE Computer Society, Los Alamitos
12. Aamodt A, Plaza E (1994) Case-based reasoning: foundational issues, methodological variations and systems approaches. AI Commun 7:39–59
13. Oppenheimer D, Ganapathi A, Patterson D (2003) Why do Internet services fail, and what can be done about it? In: Proceedings of 4th usenix symposium on Internet technologies and systems (USITS '03), Seattle, WA, USA, March 2003
14. Montani S, Anglano C (2006) Case-based reasoning for autonomous service failure diagnosis and remediation in software systems. In: Roth-Berghofer T.R. et al (eds) Proceedings of the European conference on case-based reasoning (ECCBR) 2006. Lecture notes in artificial intelligence, vol 4106. Springer, Berlin, pp 489–503
15. Joshi KR, Hiltunen MA, Sanders WH, Schlichting RD (2005) Automatic model-driver recovery in distributed systems. In: Proceedings of 24th IEEE symposium on reliable distributed systems (SRDS 05). IEEE, New York
16. Console L, Friedrich G, Theseider-Dupre D (1993) Model-based diagnosis meets error diagnosis in logic programming. In: Proceedings of the IJCAI, Chambery, France, pp 1494–1499
17. Wotawa F (2002) On the relationship between model-based debugging and program slicing. Artif Intell 135:125–143
18. Friedrich G, Stumptner M, Wotawa F (1999) Model-based diagnosis of hardware design. Artif Intell 111:3–39
19. Koshgoftaar TM, Seliya N, Sunsaresh N (2006) An empirical study of predicting software faults with case-based reasoning. Soft Qual J 14:85–111
20. Avizienis A, Laprie J, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 1(1):11–33
21. Kaiser G, Parekh J, Gross P, Valetto G (2003) Kenesthetics extreme an external infrastructure for monitoring distributed legacy systems. In: Proceedings of 5th IEEE international active middleware workshop, Seattle, WA, USA, June 2003. IEEE Computer Society, Los Alamitos
22. Nunez H, Sanchez-Marre M (2004) Instance-based learning techniques of unsupervised features weighting do not perform so badly! In: Lopez de Mantaras R, Saitta L (eds) Proceedings of the European conference on artificial intelligence (ECAI). IOS, Amsterdam, pp 102–108
23. Wilson DR, Martinez TR (1997) Improved heterogeneous distance functions. J Artif Intell Res 6:1–34
24. Schmidt R, Montani S, Bellazzi R, Portinale L, Gierl L (2001) Case-based reasoning for medical knowledge-based systems. Int J Med Inf 64(2-3):355–367
25. Gierl L, Stengel-Rutkowski S (1994) Integrating consultation and semi-automatic knowledge acquisition in a prototype-based architecture: Experiences with dysmorphic syndromes. Artif Intell Med 6:29–49
26. The Rubis Project Home Page. http://rubis.objectweb.org
27. The cURL and libcurl Home Page. http://curl.haxx.se
28. Quinlan JR (1993) C4.5: Programs for machine learning. Morgan, San Mateo
29. The C4.5 Distribution Page. http://www2.cs.uregina.ca/~dbd/cs831/notes/ml/dtrees/c4.5/c4.5r8.tar.gz
30. Hammond KJ (1989) Case-based planning: viewing planning as a memory task. Academic, New York
31. Aha D, Daniels J (eds) (1998) Proceedings of the AAAI workshop on CBR integrations. AAAI, Menlo Park
32. Freuder E (ed) (1998) Proceedings of the AAAI spring symposium on multi-modal reasoning. AAAI, Menlo Park
33. Bonissone PP, Dutta S (1990) Integrating case-based and rule-based reasoning: the possibilistic connection. In: Proceedings of 6th conference on uncertainty in artificial intelligence, Cambridge, MA, USA, July 1990
34. Surma J, Vanhoof K (1995) Integration rules and cases for the classification task. In: Veloso M, Aamodt A (eds) Proceedings of the 1st international conference on case-based reasoning, Lecture notes in computer science, vol 1010. Sesimbra, Portugal, October 1995. Springer, Berlin, pp 325–334
35. Macchion D, Vo D (1993) A hybrid knowledge-based system for technical diagnosis learning and assistance. In: Wess S, Althoff K, Richter M (eds) Proceedings of the 1st European workshop on case-based reasoning, Kaiserslautern, Germany, November 1993. Lecture notes in computer science, vol 837. Springer, Berlin, pp 301–312
36. Branting LK, Porter BW (1991) Rules and precedents as complementary warrants. In: Proceedings of 9th national conference on artificial intelligence, Anaheim, CA, USA, July 1991. AAAI, Menlo Park
37. Rissland E, Skalak D (1989) Combining case-based and rule-based reasoning: a heuristic approach. In: Sridharan NS (ed) Proceedings of 11th international joint conference on artificial intelligence, pp 524–530
38. Bichindaritz I, Kansu E, Sullivan K (1998) Case-based reasoning in care-partner: Gathering evidence for evidence-based medical practice. In: Smyth B, Cunningham P (eds) Proceedings 4th European workshop on case-based reasoning, Dublin, Ireland, September 1998. Lecture notes in computer science, vol 1488. Springer, Berlin, pp 334–345
39. Xu LD (1996) An integrated rule- and case-based approach to AIDS initial assessment. Int J Biomed Comput 40:197–207