Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Via Bellini 25/G, 15100 Alessandria
http://www.di.unipmn.it

Achieving Self-Healing in Autonomic Software Systems: a Case-Based Reasoning Approach

Cosimo Anglano, Stefania Montani
(cosimo.anglano,stefania.montani@unipmn.it)

TECHNICAL REPORT TR-INF-2005-09-03-UNIPMN
(October 2005)

Achieving Self-Healing in Autonomic Software Systems: a Case-Based Reasoning Approach

Cosimo Anglano, Stefania Montani
Dipartimento di Informatica
Università del Piemonte Orientale
Via Bellini 25/g, Alessandria, Italy.
email: {cosimo.anglano, stefania.montani}@unipmn.it

Abstract

Self-healing, one of the four key properties characterizing Autonomic Systems, can enable large-scale software systems delivering complex services on a 24/7 basis to meet their goals without requiring any human intervention. In this paper we present a self-healing methodology for software systems based on Case-Based Reasoning, a reasoning paradigm that exploits the unformalized knowledge embedded into already solved instances of problems, able to take advantage from the fact that in software systems most errors are just re-occurrences of known problems. We demonstrate the practical applicability of our methodology by showing how it can be used to obtain a self-healing software system delivering large-scale Internet services.

1 Introduction

As pointed out by Ganek and Corbi [14], "the computer industry has spent decades creating systems of marvelous and ever-increasing complexity, but today complexity itself is the problem." By looking at today's large-scale networked applications and services, whose growth has been quite substantial in the recent years, we can indeed observe that their inherent complexity, heterogeneity, and dynamism makes inappropriate, if not impossible, the traditional human-centered approach to system administration. 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* [14, 21, 31, 44], 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 *managed elements*, whose behavior is controlled by *autonomic managers* that apply suitable policies in order to automate the process of system management. Autonomic managers behave according to the so-called *autonomic cycle* [21, 31, 44], schematically depicted in Fig. 1, that encompasses four distinct steps. More specifically, an

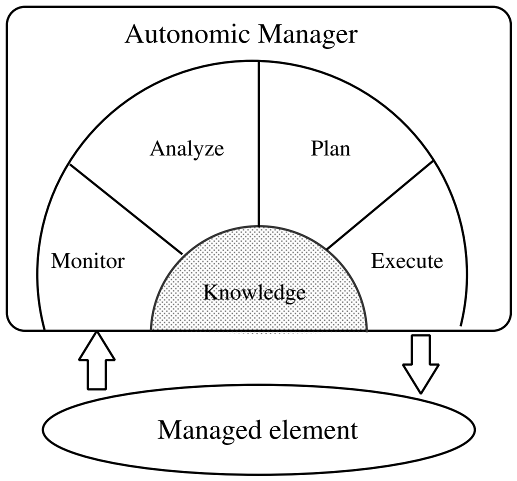

Figure 1: The architecture of an Autonomic Manager, illustrating the *autonomic cycle*.

autonomic manager continuously monitors the corresponding managed element (in the *Monitoring* step) in order to collect information concerning its state and behavior, analyzes this information (in the *Analysis* step) to determine possible deviations from the correct or intended behavior, devises proper corrective actions (in the *Plan* step) to bring the system back into normal behavior, and implements these plans (in the *Execution* step) by exploiting suitable *actuation mechanisms* that must be provided by the managed element.

In order to be able self-managing, an ACS should exhibit the so called *Self-** properties, that is it should be *Self-Configuring* (i.e., able to (re)configure itself under varying and unpredictable conditions), *Self-Optimizing* (i.e., able to detect suboptimal behavior and to optimize itself to improve its execution), *Self-Protecting* (i.e., able to protect itself from both external and internal attacks), and *Self-Healing* (i.e., able to detect and recover from problems and/or failures). Obtaining these properties is the goal of autonomic managers, whose activity is driven by both human-specified policies and knowledge that is acquired and updated during the operation of the system.

The Autonomic Computing paradigm is particularly attractive for large-scale software systems aimed at delivering on-line services on a 24/7 basis, as for instance those described in [9, 29]. The very large size of these systems (that may typically include from hundreds to thousands machines), and the adoption of customized application software and middleware, makes at the same time service failures relatively frequent and human-centered system administration very hard, if not impossible. Consequently, self-managing capabilities, and especially self-healing, represent a very attractive solution for the management problems of these systems. In this paper we focus on the problem of achieving self-healing in (large-scale) software systems, that can be defined as the ability of a system to repair itself after the occurrence of a *fault* (in one or more of its constituent components) that led to an *error*, that in turn caused a *service failure* (that is, a deviation from the correct or intended behavior of a system delivering a service) [1]. As prescribed by the autonomic cycle, a self-healing software system must be able to *monitor* its own behavior, in order to identify service failures, to *analyze* these failures with the aim of diagnosing the faults causing them, to *devise* a suitable repair plan able to fix the faults, and to *execute* the above plans by means of suitable mechanisms.

While monitoring [41, 42] and reconfiguration[12, 17, 26] are certainly important, the ability to correctly diagnose the faults causing a service disruption, and to devise the corresponding repair strategy, is crucial to the achievement of self-healing. As a matter of fact, if a service failure is ascribed to the wrong cause, an ineffective (or even wrong) solution is proposed, with the consequence that other undesired behavior may be introduced into the system. In this paper we address the problem of devising fault diagnosis and remediation techniques that can be used by an autonomic manager of an Autonomic Software System to identify both the causes and the possible solutions of a service failure. More specifically, we propose a methodology supporting autonomous *service failure diagnosis and remediation* (as opposed to fault diagnosis and remediation pursued in traditional approaches) that exploits Case-Based Reasoning (CBR), a reasoning paradigm able to exploit the unformalized knowledge embedded into already solved instances of problems [2, 23].

The rest of the paper is organized as follows. In Section 2 we present the related work, while in Section 3 we briefly review the main features of CBR. In Section 4 we discuss how CBR can be used for service failure diagnosis and remediation in Autonomic Software Systems, and in Section 5 we demonstrate our technique by using the failure and repair data of some real, large-scale Internet Service systems reported in [28]. Finally, Section 6 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 [40] proposes 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 [15], where fault diagnosis is performed by means of a suitable set of models. Brodie *et al.* [10] propose a technique in which recurring software errors are identified by means of the call stack generated by the faulty program, that are stored into a database and are used for fault diagnosis. Littman *et al.* [24] propose *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 represents also the basis for the fault remediation strategy proposed by Arshad *et al.* [5].

The main drawback of these approaches is that they require the availability of formalized and widely recognized background knowledge (henceforth referred to as *structured knowledge*) on 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 requires the availability of a model describing how the various system components interact among them. Unfortunately, significant efforts are 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. Another drawback of these 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, even if it is not (yet) causing any service disruption because it is *dormant fault* [1], or a fault that 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 if, when it occurs, can be masked as well. Furthermore, from the perspective of service delivery, a dormant or a masked fault has little or no importance until it causes a service failure (i.e., it becomes *active* [1]). However, 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 (that 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 (*preventive maintenance* [1]) requires the availability of a model of the system, that brings us back to the problem of structured knowledge mentioned before. Finally, the proposals discussed before either address fault diagnosis or fault remediation, but none of them addresses both issues at the same time.

The CBR-based methodology we present in this paper does not suffer from the problems mentioned above. It is indeed particularly suited to those domains in which a formalized and widely recognized background knowledge is not available. This is often the case when designing an Autonomic Computing System for which structured knowledge is too hard to collect and maintain, or when "retrofitting" self-healing capabilities into existing legacy applications [19]. Moreover, CBR seems particularly appropriate to failure diagnosis and remediation in software systems, as in this domain most errors are re-occurrence of known problems [5, 10, 20, 28, 39, 46], and provides a unique framework in which failure diagnosis and remediation are performed jointly. Nevertheless, rather interestingly, very few proposals of adoption of CBR in this field can be found. To the best of our knowledge, the only proposal resembling ours has been published in [10], where a Case-Based retrieval system for discovering software problems without requiring human intervention is presented. Despite this contribution represents a first concrete step in the direction of relying on CBR for Autonomic Computing, the approach is still quite limited, as it consists in a pure retrieval systems, in which the other steps of the CBR cycle (see section 3) are ignored, and the problem solution is not provided.

3 Case-Based Reasoning

Case-Based Reasoning is a reasoning paradigm able to exploit the unformalized knowledge embedded into already solved instances of problems [2, 23], called *cases*. In some sense, CBR is able to mimic human experts' analogical reasoning, by remembering solutions to similar problems adopted in the past, and by adapting them to the current situation. The problem solving experience gained in the past is explicitly taken into account by storing past cases in a library (the *case base*), and by suitably retrieving them when a new problem has to be dealt with. A case consists of the following information:

- the *problem description*, i.e. a collection of $\langle feature, value \rangle$ pairs able to summarize the problem at hand;
- the *case solution*, describing the solution adopted for solving the corresponding problem;
- the *case outcome*, describing the (positive or negative) result obtained by applying the solution itself.

Two basic possibilities exist for exploiting CBR for complex problems solving, namely:

- *Precedent Case-Based Reasoning*, where the emphasis is on retrieving past cases, and on using past solutions as a justification for the solution of the current problem, with almost no adaptation (e.g. legal reasoning);
- *Case-Based Problem Solving*, where retrieved solutions to previous similar cases need to be adapted to fit the current situation (e.g. planning, design, diagnosis, etc.).

Case-Based Problem Solving is the most general approach, and can be summarized by the following four basic steps, known as the *CBR cycle* [2]:

1. *Retrieve* the most similar case(s) from the case library;
2. *Reuse* them, and more precisely their solutions, to solve the new problem;
3. *Revise* the proposed new solution;
4. *Retain* the current case for future problem solving.

In the above cycle some steps may be missing or may be collapsed. For instance, it is quite common to view the *reuse* and *revise* steps as a single one, or to avoid the *retain* step if the current case is in some sense "covered" by other cases already stored in the library. These steps may be fully automated, although usually some user intervention is needed to perform adaptation and reuse.

A critical aspect affecting the efficiency of CBR is case retrieval, whose computational cost strongly depends on the organization of the case base. Various solutions have been proposed in the literature, ranging from *flat memories*, in which cases are stored sequentially as lists or feature vectors, to more structured organizations (like *shared features nets* and *discrimination nets* [23]). While flat memories are simple to update, but require clever strategies to avoid exhaustive search [32, 36], more complex organizations are harder to organize and maintain, but allow for a faster search.

As anticipated in Section 2, CBR is particularly appealing in situations where the domain knowledge is poor and difficult to explicit (e.g., when a model of the system is not available), since the bottleneck of knowledge acquisition and representation is reduced, as new (unformalized) knowledge is automatically stored in the case base during the normal working process: no additional effort is required to the user and/or to the domain expert. As the case library grows, more and more representative examples can be retrieved, and it will become easier to find a proper solution to the problem at hand. CBR has been applied in several fields, mainly dealing with diagnostic problem solving [4, 22] or planning, and has successfully been exploited also for industrial applications [34, 33]. Moreover, CBR can be easily integrated with other knowledge sources (if available) and with other reasoning paradigms, making the methodology suitable also for of applications with a partially available background knowledge, or with a known problem-diagnosis model. The interest in multi modal approaches involving CBR is recently increasing through different application areas [3, 13], from planning [7] to classification [43] and to diagnosis [25], and from legal [8, 35] to medical decision support [6, 27, 37, 47]. Different reasoning methods can be combined in the same application, or it can be possible to switch among alternative reasoning paradigms. CBR is particularly well suited for integration with Rule Based or Model Based systems[16].

4 A CBR-based Approach to Self-Healing

As discussed in the Introduction, a complete self-healing solution requires that all the steps of the autonomic cycle are carried out. Roughly speaking, there are two possible ways of doing so, namely (a) integrating the autonomic cycle into the system, thus in a certain sense embedding the autonomic manager into the managed system, and (b) "surrounding" the managed system by an external closed control loop (an approach named *externalization* in [15], where it has been proposed for the first time). While the former approach is undoubtedly more general, it requires innovative design and implementation techniques that, at the moment, are not mature enough. Furthermore, it cannot be used with existing applications whose size, complexity, and unavailability of source code may prevent any required modification. For these reasons, an externalization approach appears more appropriate.

Our proposal, schematically illustrated in Fig. 2, uses externalization, as done in Garlan and Schmerl's work [15], but relies on CBR, rather than resorting to a model of the system, so that the need of acquiring and maintaining structured knowledge about the system is avoided. The basic idea of externaliza-

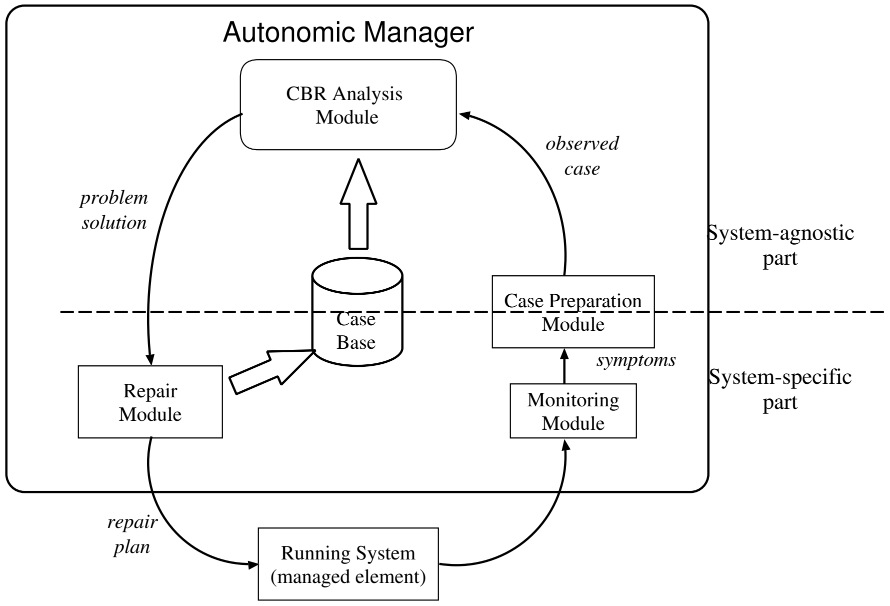

Figure 2: CBR-based Self-Healing

tion [15] is to treat the system as a "black box" surrounded by a set of external modules that form a "closed-loop" controlling the "health" of the system and performing proper repair actions in case of service failures. In our approach this closed loop includes, in addition to the `Running System` (corresponding to the *managed element* in the autonomic computing terminology), four external modules, that jointly act in such a way that self-healing is obtained. In particular, when the `Monitoring` module detects a service failure, it passes the set of *symptoms* of this misbehavior to the `Case Preparation` module, whose purpose is to assemble a case with the proper structure and passes it to the `CBR Analysis` module. The `CBR Analysis` module (which is the core of our architecture), upon receiving an *observed case* from the `Case Preparation` module, finds the solution to the case at hand (or, more precisely, the best solution among those available in the case base), and passes it to the `Repair Module` that, by using suitable system-specific mechanisms, executes the corresponding *repair plan.* Note that during the repair a new case, describing the new solution derived from a previously-stored one, can be generated and stored into the case base.

As shown in Fig. 2, the external modules can be classified either into *System-specific* (i.e., that must be tailored to the specific characteristics of the running system) or *System-agnostic*. While the monitoring and repair activities require the availability and usage either of mechanisms provided by the running system or of adapted third-party components [18] (hence their classification as system-specific), the problem resolution activity (performed by the `CBR Analysis` module) does not rely upon any particular system feature (hence it classification as system-agnostic, although an adaptation strategy would require specific domain knowledge). The `Case Preparation` module has been instead drawn as crossing the dotted line separating system-specific from system-agnostic modules, as it interfaces the system-specific part of the system with the system-agnostic one.

As can be observed by comparing Fig. 2 with Fig. 1 (illustrating the autonomic cycle), our architecture closely matches the behavior of a self-healing autonomic manager. This is the consequence of the fact that the CBR cycle fits very well into the autonomic cycle, since it naturally covers the *Analysis* and *Planning* phases, while the other two phases (i.e., *Monitoring* and *Execution*) are covered by the `Monitoring` and `Repair` modules of our architecture. In particular, the *Retrieve* step of the CBR cycle corresponds to the *Analysis* step of the autonomic cycle, the *Reuse-Revise* steps of CBR, where past retrieved solutions are evaluated (and, if necessary, adapted) correspond to the *Planning* step, and the *Knowledge* used by the Autonomic Manager is contained in the `Case Base`.

In order to behave as a really self-healing system, 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 it is not available when the system is initially put into operation. We therefore envision a *bootstrap* phase, enabling the collection of the initial cases, during which problem solution is performed by humans. However, as the system experiences failures that are solved by human operators, the case base grows to include the information concerning solved problems, so after a relatively limited amount of time problems solution can be performed autonomously. An obvious question that may arise is concerned with the size of the case base, that may potentially become very large if the number of stored cases keeps increasing. However, in these situations it is possible to keep the case base size within reasonable limits by resorting to technique already published in the literature [38], like for instance the definition of suitable "prototypes" able to summarize the information carried by the ground cases they represent, that are stored in place of the cases.

As a final consideration, we note that some aspects of the CBR implementation are strongly domain dependent; typically, the possibility of simply reusing a case solution vs. the need for adaptation and, in this situation, the details of the adaptation strategy, cannot be provided in a general fashion, but need to be tailored to the application under consideration. Nevertheless, the mapping of the CBR steps to the self-healing cycle appears to be general enough to be shared across different contexts.

5 Case Study: Self-Healing in Large-Scale Internet Services

In this Section we illustrate how our approach can be practically used to obtain self-healing in a computing system conceived to deliver large-scale Internet services. In order to make our discussion as realistic as possible, we use as a case study one of the systems described in [28, 29], for which the data concerning some representative service failures and the corresponding repair actions are available.

The system chosen for our discussion is the one named *Online* in [28, 29], whose purpose is to deliver an online service/Internet portal on a 24/7 fashion. The system is spread across two data centers, and includes about 500 machines altogether. Each data center hosts a subsystem using a three-tier architecture, in which a load-balancing tier distributes the load across a set of front-end servers, that in turn rely on a set of back-end servers to carry out their operation. The *Online* computing system uses both Sparc and x86 machines running the Solaris operating system, while the software and the middleware used to provide the service is customized, and is updated very frequently (typically every week).

The sheer size of the system makes the fault frequency relatively high, so proper fault-tolerant techniques and strategies have been incorporated into its design. As shown in [28], these techniques work reasonably well and are able to mask a large fraction of those faults that can be ascribed to node hardware/software problems (although some of them turn into service failures that must be properly handled), but are very often unable to mask those faults that are caused by operators' errors (typically software misconfigurations), that consequently turn into service failures.

In order to illustrate how our self-healing approach works, we consider three service failures (and the corresponding solutions), two of which have been taken from [28], while the third one has not been observed in the actual system, but is very plausible in practice. Starting from these failures, we describe a possible case structure, and we show how a service failure can be autonomously repaired by our self-healing infrastructure. The service failures we consider for our case study are the following:

- *Failure 1* (from [28]): the system was not delivering the email messages to its users. The failure was found to be caused by a software upgrade of the front-end daemon that handles username and alias lookup, that inadvertently changed the format of the string used by that daemon to query the back-end database server that stores usernames and aliases. The daemon continually retried all lookups because they were failing, eventually overloading the back database, causing also the risk of bringing down other services using it. The problem was finally fixed by rolling back the software upgrade and rebooting the database and front-end nodes.

- *Failure 2* (fictitious): the system was, again, not delivering the email messages to its users, but this time no increase in the username/alias lookup frequency occurred. The inspection of the logs of the email server however revealed a significant fraction of discarded messages. The failure was found to be caused by a misconfiguration of the email servers, that set an internal buffer to a too small value. The problem was fixed by increasing the buffer size and by restarting the email service on front-end nodes.

- *Failure 3* (from [28]): users' posting were (sometimes) not showing up on the service's newsgroup. The failure was due to an operator misconfiguration that had caused the newsgroup posting server not to run the daemon performing username/alias lookups. This server requires indeed that the message sender is a valid *Online* user, otherwise the message is silently dropped. The posted messages were being silently dropped since all the user authentication requests were failing because the username lookup daemon was not running. The failure was fixed by correcting the configuration file, and by restarting the newsgroup service on front-end nodes.

Let us now discuss a possible structure for the cases based on the information concerning the failures and their symptoms. As discussed in Section 3, a case consists in the *problem description* (i.e., a collection of <feature,value> pairs), the *case solution* (i.e., a description of the solution adopted to solve the corresponding problem) and the *case outcome* (describing the positive or negative result obtained by applying the solution). Let us start with problem description by giving the list of features used for problem description, reported in Table 1, that correspond to the symptoms of the failures. This list is derived from the

Table 1: Features used to describe cases

Feature	**Possible Values**	**Description**
email deliv.	*Yes*/*No*	indicates whether email messages are received (*Yes*) or not (*No*)
lookup number	*None*/*Low*/ *Normal*/*High*	indicates the number of username/alias lookups per unit of time
disc. msg.	*Yes*/*No*	indicates whether any service log reports discarded messages (*Yes*) or not (*No*)
news msg appear	*Yes*/*No*	indicates whether all posted messages appear in the newsgroup (*Yes*) or not (*No*)
lookup daemon running	*Yes*/*No*	indicates whether the username/alias look daemon is running (*Yes*) or not (*No*)

three service failures discussed above and has been chosen in order to suit the example (of course, in general the cases may have a more complex structure). Each case will contain all the features listed above, although some of them may have not been observed for a specific case, plus two fields corresponding to its solution and outcome. Table 2 reports the case base for our example.

Table 2: The case base corresponding to the three service failures. The NULL value is used to indicate that the corresponding feature has not been observed.

Case desc.	**email deliv.**	**lookup number**	**disc. msg.**	**news msg appear**	**lookup daemon running**	**problem solution**	**case outcome**
case 1	No	High	No	NULL	NULL	rollback to previous software version and reboot front-end nodes	positive
case 2	No	Normal	Yes	NULL	NULL	resize buffer and restart email service	positive
case 3	NULL	None	Yes	No	No	start lookup daemon and restart news service	positive

Let us now describe how our self-healing methodology would solve a new service failure, after it has been observed by the `Monitoring` module. We assume that the `Case Preparation` module (not described here because of its simplicity) formats the new case according to the structure described before, and passes it to the `CBR Analysis` module. For our example, let us assume that the new case has the structure reported in Fig. 3, that corresponds to an

email deliv.	**lookup number**	**disc. msg.**	**news msg appear**	**lookup daemon running**
No	NULL	Yes	NULL	Yes

Figure 3: Case corresponding to the new service failure that is being observed

email messages delivery failure, for which message discards are reported into the servers logs, and the lookup daemon is observed to be running.

Starting from this case, the `CBR Analysis` module retrieves the most similar past case chosen among those stored into the case base. In order to do so, it must have a way of measuring the similarity between cases, that in turn requires the definition of a measure of distance among cases. Generally speaking, the distance $d(c_i, c_j)$ between cases c_i and c_j can be computed as weighted average of the normalized distances among their various features, that is:

$$d(c_i, c_j) = \frac{\sum_{k=1}^{N} w_k \cdot d(c_i(k), c_j(k))}{N} \tag{1}$$

where $d(c_i(k), c_j(k))$ and w_k denote the normalized distance between feature k of cases c_i and c_j, and the weight associated with this feature, respectively.

The features of the cases used in our example may take only either boolean or categorical values, for which two different measures of distance can be used. For boolean features we use to so-called *overlap distance* [45], that is defined as:

$$d(c_i(k), c_j(k)) = \begin{cases} 0 & \text{if } c_i(k) = c_j(k) \\ 1 & \text{otherwise} \end{cases}$$

For categorical features (only *lookup number*, in our case) we use a *similarity table* that explicitly lists the distance among all the pairs of possible values, defined as follows: [1]

$$d(c_i(k), c_j(k)) = \begin{cases} 0 & \text{if } c_i(k) = c_j(k) \\ 0.5 & \text{if } c_i(k) = \mathit{None} \wedge c_j(k) = \mathit{Low} \text{ or viceversa} \\ 0.5 & \text{if } c_i(k) = \mathit{Low} \wedge c_j(k) = \mathit{Normal} \text{ or viceversa} \\ 0.5 & \text{if } c_i(k) = \mathit{Normal} \wedge c_j(k) = \mathit{High} \text{ or viceversa} \\ 0.75 & \text{if } c_i(k) = \mathit{None} \wedge c_j(k) = \mathit{Normal} \text{ or viceversa} \\ 0.75 & \text{if } c_i(k) = \mathit{Low} \wedge c_j(k) = \mathit{High} \text{ or viceversa} \\ 1 & \text{otherwise} \end{cases}$$

By using the above distance measures, and by defining as '1' the distance between the value *NULL* and any other value, the distances between the new case (Fig. 3) and each case included in the case base (Table 2) are those reported in Table 3, from which we can conclude that the closest match to the new case is *case 2*.

Table 3: Distances between the new service failure and the cases in the case base

distance w.r.t.	**email deliv.**	**lookup number**	**disc. msg.**	**news msg appear**	**lookup daemon**	**overall distance**
case 1	0	1	1	1	1	$\frac{4}{5}$
case 2	0	1	0	1	1	$\frac{3}{5}$
case 3	1	1	0	1	1	$\frac{4}{5}$

Once the most similar case has been determined, the corresponding solution can be used to solve the problem corresponding to the new case. As already mentioned in Section 3, sometimes this solution may be readily used to solve the new problem (as in the example discussed in this section), while in other cases it may have to be adapted if it does not solve directly the new problem. Solution adaptation, however, cannot be always automated in a simple way, so in these cases some form of human intervention may be required. A similar consideration applies to the execution of the devised repair plan, that requires the availability of suitable reconfiguration/repair mechanisms, that sometimes may be hard to develop in a completely autonomous manner. In spite of that,

[1] Of course, other distance values are possible, but for our purposes this simple definition is sufficient.

we believe that our system represents a step towards the achievement of a full self-healing behavior. More precisely, we believe that our system fits between *Level 3* (*Predictive*) and *Level 4* (*Adaptive*) of the Ganek and Corbi's scale of autonomicity [14], depending on the amount of human intervention required to carry out the tasks mentioned above. As a final consideration, it is worth to point out that, although the example presented in this section is relatively simple, at the same time it is representative of a large class of real-world situations. Moreover, since the CBR methodology can handle exactly in the same way more complex cases, we believe that our approach can be practically used to achieve self-healing in real-world, large-scale software systems.

6 Conclusions

In this paper we have presented a CBR-based approach for the achievement of self-healing in software systems that, unlike alternative solutions, directly addresses service failures rather than individual component faults, so that unnecessary repair actions are avoided. Moreover, it does not require the availability of structured knowledge like, for instance, models of the behavior of the system, thus easing its applicability to large-scale, complex software systems. The suitability of this approach to real world applications has been exemplified by showing how it can be used to make a large-scale Internet service system able to self-heal. The relative simplicity of the case study chosen to illustrate our methodology is in no way a requirement for the applicability of CBR, that can handle exactly in the same way more complex cases. Therefore, we believe that our methodology represent a concrete step towards the achievement of self-healing for complex, large-scale software systems.

As future work, we plan to apply our methodology in practice by using it to provide self-healing in real-world large-scale systems, like PlanetLab [11] and OurGrid [30]. In order to do this, an implementation of both the system-agnostic and system-specific modules of our infrastructure need to be implemented and integrated.

References

[1] A. Avizienis and J. Laprie and B. Randell and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), January-March 2004.

[2] A. Aamodt and E. Plaza. Case-Based Reasoning: foundational issues, methodological variations and systems approaches. *AI Communications*, 7(1):39–59, 1994.

[3] D. Aha and J. Daniels, editors. *Proc. AAAI Workshop on CBR Integrations.* AAAI Press, 1998.

[4] K.-D. Althoff and S. Wess. Case-Based Knowledge Acquisition, Learning, and Problem Solving in Diagnostic Real World Tasks. In *Proc. of Fifth European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW '91)*, Crieff, Scotland, UK, Jun. 1991.

[5] N. Arshad, D. Heimbigner, and A. Wolf. A Planning Based Approach to Failure Recovery in Distributed Systems. In *Proc. of 2nd ACM Workshop on Self-Healing Systems (WOSS '04)*, Newport Beach, CA, USA, October 2004. ACM Press.

[6] I. Bichindaritz, E. Kansu, and K. Sullivan. Case-based reasoning in care-partner: Gathering evidence for evidence-based medical practice. In B. Smyth and P. Cunningham, editors, *Proc. 4th European Workshop on Case-Based Reasoning*, volume 1488 of *Lecture Notes in Computer Science*, pages 334–345, Dublin, Ireland, September 1998. Springer.

[7] P.P. Bonissone and S. Dutta. Integrating case-based and rule-based reasoning: the possibilistic connection. In *Proc. of 6th Conference on Uncertainty in Artificial Intelligence*, Cambridge, MA, USA, July 1990.

[8] L.K. Branting and B.W. Porter. Rules and precedents as complementary warrants. In *Proc. of 9th National Conference on Artificial Intelligence*, Anaheim, CA, USA, July 1991. AAAI Press.

[9] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4), 2001.

[10] M. Brodie, S. Ma, G. Lohman, T. Syeda-Mahmood, L. Mignet, N. Modani, J. Champlin, and P. Sohn. Quickly finding known software problems via automated symptom matching. In *Proc. of the 2nd International Conference on Autonomic Computing*, Seattle, WA, USA, June 2005.

[11] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communications Review*, 33(3), July 2003.

[12] E. Dashofy, A. van der Hoek, and R. Taylor. Towards Architecture-based Self-Healing Systems. In *Proc. of 1st ACM Workshop on Self-Healing Systems (WOSS '02)*, Charleston, SC, USA, November 2002. ACM Press.

[13] E. Freuder, editor. *Proc. AAAI Spring Symposium on Multi-modal Reasoning.* AAAI Press, 1998.

[14] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[15] D. Garlan and B. Schmerl. Model-based Adaptation for Self-Healing Systems. In *Proc. of 1st ACM Workshop on Self-Healing Systems (WOSS '02)*, Charleston, SC, USA, November 2002. ACM Press.

[16] K.J. Hammond. *Case-Based Planning: viewing planning as a memory task.* Academic Press, 1989.

[17] M. Hawthorne and D. Perry. Architectural Styles for Adaptable Self-Healing Dependable Systems. In *Proc. of 2005 ACM International Conference on Software Engineering (ICSE '05).* ACM Press, May 2005.

[18] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kenesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. In *Proc. of 5^th IEEE International Active Middleware Workshop*, Seattle, WA, USA, June 2003. IEEE CS Press.

[19] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Retrofitting Autonomic Capabilities onto Legacy Systems. Technical Report TR CUCS-026-03, Department of Computer Science, Columbia University, 2003.

[20] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proc. of 18^{th} IEEE Symposium on Reliable Distributed Systems*, Lausanne, Switzerland, October 1999. IEEE CS Press.

[21] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer*, January 2003.

[22] J. Kolodner and R. Kolodner. Using experience in clinical problem solving: introduction and framework. *IEEE Transactions on Systems, Man and Cybernetics*, 17:420–431, 1987.

[23] J.L. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, 1993.

[24] M. Littman, T. Nguyen, and H. Hirsh. Cost-Sensitive Fault Remediation for Autonomic Computing. In *Proc. of IJCAI Workshop on AI and Autonomic Computing: Developing a Research Agenda for Self-Managing Computer Systems*, Acapulco, Mexico, August 2003.

[25] D. Macchion and D. Vo. A hybrid knowledge-based system for technical diagnosis learning and assistance. In S. Wess, K. Althoff, and M. Richter, editors, *Proc. 1st European Workshop on Case-Based Reasoning*, volume 837 of *Lecture Notes in Computer Science*, pages 301–312, Kaiserslautern, Germany, November 1993. Springer.

[26] M. Mikic-Rakic, N. Mehta, and N. Medvidovic. Architectural Style Requirements for Self-Healing Systems. In *Proc. of 1^{st} ACM Workshop on Self-Healing Systems (WOSS '02)*, Charleston, SC, USA, November 2002. ACM Press.

[27] S. Montani, L. Portinale, G. Leonardi, and R. Bellazzi. Case-based retrieval to support the treatment of end stage renal failure patients. in press.

[28] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet services fail, and what can be done about it? In *Proc. of 4^{th} Usenix Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, USA, March 2003.

[29] D. Oppenheimer and D. Patterson. Architecture and Dependability of Large-Scale Internet Services. *IEEE Internet Computing*, September-October 2002.

[30] The ourgrid home page. http://www.ourgrid.org. Accessed on August 2nd, 2005.

[31] M. Parashar and S. Hariri. Autonomic Computing: An Overview. In *Proc. of 2004 Workshop on Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, Mont Saint-Michel, France, September 2004. Springer.

[32] L. Portinale, P. Torasso, and D. Magro. Selecting most adaptable diagnostic solutions through pivoting-based retrieval. In D. Leake and E. Plaza, editors, *Proc. of 2nd International Conference on Case-Based Reasoning*, volume 1266 of *Lecture Notes in Computer Science*, pages 393–402, Providence, RI, USA, July 1997. Springer.

[33] R. Bergmann. *Experience Management: Foundations, Development Methodology, and Internet-Based Applications*, volume 2432 of *Lecture Notes in Computer Science*. Springer, Berlin, 2002.

[34] R. Bergmann and K.-D. Althoff and S. Breen and M. Goker and M. Manago and R. Traphoner and S. Wess. *Developing Industrial Case-Based Reasoning Applications. The INRECA Methodology*, volume 1612 of *Lecture Notes in Artificial Intelligence*. Springer, Berlin, 2003.

[35] E. Rissland and D. Skalak. Combining case-based and rule-based reasoning: A heuristic approach. In N. S. Sridharan, editor, *Proc. of 11th International Joint Conference on Artificial Intelligence*, pages 524–530, 1989.

[36] J.W. Schaaf. Fish and shrink. a next step towards efficient case retrieval in large-scale case bases. In I. F. C. Smith and B. Faltings, editors, *Proc. 3rd European Workshop on Case-Based ReasoningEWCBR*, volume 1168 of *Lecture Notes in Computer Science*, pages 362–376, Lausanne, Switzerland, November 1996. Springer.

[37] R. Schmidt, S. Montani, R. Bellazzi, L. Portinale, and L. Gierl. Case-based reasoning for medical knowledge-based systems. *International Journal of Medical Informatics*, 64(2-3):355–367, 2001.

[38] R. Schmidt, S. Montani, R. Bellazzi, L. Portinale, and L. Gierl. Case-based reasoning for medical knowledge-based systems. *International Journal of Medical Informatics*, 2–3, 2001.

[39] C. Simache, M. Kaaniche, and A. Saidane. Event Log based Dependability Analysis of Windows NT and 2K Systems. In *Proc. of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC '02)*, Tsukuba, Japan, December 2002. IEEE CS Press.

[40] R. Sterrit. Autonomic networks: engineering the self-healing property. *Engineering Applications of Artificial Intelligence*, 17:727–739, October 2004.

[41] R. Sterrit and S. Chung. Personal Autonomic Computing Self-Healing Tool. In *Proc. of 11th International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS '04)*, Brno, Czech Republic, May 2004. IEEE CS Press.

[42] R. Sterrit, D. Gunning, A. Meban, and P. Henning. Exploring Autonomic Options in an Unified Fault Management Architecture through Reflex Reactions via Pulse Monitoring. In *Proc. of 11th International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS '04)*, Brno, Czech Republic, May 2004. IEEE CS Press.

[43] J. Surma and K. Vanhoof. Integration rules and cases for the classification task. In M. Veloso and A. Aamodt, editors, *Proc. 1st Int. Conference on Case-Based Reasoning*, volume 1010 of *Lecture Notes in Computer Science*, pages 325–334, Sesimbra, Portugal, October 1995. Springer.

[44] H. Tianfield and R. Unland. Towards autonomic computing systems. *Engineering Applications of Artificial Intelligence*, 17:689–699, 2004.

[45] D.R. Wilson and T.R. Martinez. Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research*, 6, 1997.

[46] J. Xu, Z. Kalbarczyk, and R. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proc. of the 1999 Pacific Rim International Symposium on Dependable Computing (PRDC '99)*, Hong Kong, China, December 1999. IEEE CS Press.

[47] L.D. Xu. An integrated rule- and case-based approach to AIDS initial assessment. *International Journal of Biomedical Computing*, 40:197–207, 1996.