Computing and Informatics, Vol. 29, 2010, 93–115

ACHIEVING COST-EFFECTIVE SOFTWARE RELIABILITY THROUGH SELF-HEALING

Alessandra Gorla, Mauro Pezzè, Jochen Wuttke

University of Lugano
Lugano, Switzerland
e-mail: {gorlaa, mauro.pezze, wuttkej}@usi.ch

Leonardo Mariani, Fabrizio Pastore

University of Milano Bicocca
Milano, Italy
e-mail: {mariani, fabrizio.pastore}@disco.unimib.it

Revised manuscript received 16 October 2009

Abstract. Heterogeneity, mobility, complexity and new application domains raise new software reliability issues that cannot be met cost-effectively only with classic software engineering approaches. Self-healing systems can successfully address these problems, thus increasing software reliability while reducing maintenance costs. Self-healing systems must be able to automatically identify runtime failures, locate faults, and find a way to bring the system back to an acceptable behavior. This paper discusses the challenges underlying the construction of self-healing systems with particular focus on functional failures, and presents a set of techniques to build software systems that can automatically heal such failures. It introduces techniques to automatically derive assertions to effectively detect functional failures, locate the faults underlying the failures, and identify sequences of actions alternative to the failing sequence to bring the system back to an acceptable behavior.

Keywords: Self-healing, autonomic computing, software reliability

Mathematics Subject Classification 2000: 68M15, 68N01

1 INTRODUCTION

Software systems are growing in complexity and size, and new software paradigms support new forms of heterogeneity and dynamic evolution of software applications. Many modern software systems are composed of subsystems that are developed and maintained by different providers, and are dynamically linked to satisfy different needs. Carzaniga et al. use the term *societies of digital systems* to refer to systems that are dynamically composed of heterogeneous multi-layered elements, and identify in the verification and validation of such systems one of the challenges for software engineering [9].

Classic verification and validation approaches try to reveal failures, and remove faults before deployment, and require the availability of the source code and the ability to control the system execution. In many modern software systems, developers do not own all the source codes, and cannot predict all configurations and environment settings that may be responsible of anomalous and faulty behaviors. Moreover, classic maintenance cycles that require human experts to debug software applications and fix faults are expensive and time consuming, and do not meet the reliability requirements of many modern applications.

Consider, for example, a system composed of servers, clients and subsystems that are dynamically bound to produce the desired results, such as complex booking systems composed on many servers and customer devices, or traffic alert systems based on the cooperation of sensors, cars and traffic elements. The different components may be deployed and maintained by different organizations, none of which owns the whole application. In complex booking systems, companies may own server applications, customers may own client applications, and device manufacturers may own the software applications running on different mobile devices. In traffic alert systems, car manufacturers may own the subsystems on the vehicles, while the cities may own the traffic elements and the sensors. The execution conditions depend on dynamically changing configurations and environment conditions that may be difficult or even impossible to predict and reproduce locally both at deployment and maintenance time. The behavior of complex booking systems depends on the current events and the stationary and mobile devices connected to the systems, while the behavior of traffic alert systems depends on the traffic conditions, the devices available on the vehicles and the functioning of the involved sensors. Failures may derive from incompatibilities between inexpensive components, like applications running on mobile devices or systems running on different cars. Faults in small and inexpensive components, like an application running on a mobile device or a car subsystem, that are dynamically bound to the system may be hard to detect and expensive to repair on time. Even short maintenance cycles that require the intervention of human experts may cause the abort of the current requests that may not be valid any more when the systems is repaired.

Failure tracking systems like the ones used by software companies active in the end-user market and the ones recently studied in research laboratories can help developers prioritize fault analysis and corrective maintenance actions, and can mi-

tigate, but not solve consumer problems [27, 12]. If for example the browser or the application running on a mobile device fails in integrating with the booking system, the software release that corrects the fault can eliminate future failures, but may not solve the contingent problems in a timely manner.

Developing fault-free software is and remains a chimera, and some software failures are and will be unavoidable, and will require expensive maintenance cycles. However, it is possible to automatically detect and heal some faults, thus alleviating the problems that stem from software failures, reducing maintenance costs and increasing the reliability of software applications. Research towards techniques and mechanisms to automatically identify failures, diagnose and heal faults is quite recent and takes different names. Horn [20] and Kephart and Chess [23] use the term *autonomic computing* to identify systems that can guarantee some minimal functionality even in unexpected execution conditions, Kramer and Magee use the term *self-managed systems* to indicate systems that can automatically react to unexpected changes in the execution conditions [24], others use the term *self-adaptive systems* to indicate systems that automatically adjust to different execution conditions. In this paper we use the term *self-adaptive systems* to indicate systems that automatically adapt to different execution conditions, and the term *self-healing systems* to indicate systems that can automatically identify failures and diagnose and heal faults, and we focus on functional failures that we define as results that differ from the expected ones.

The problem of recovering from unexpected functional failures has been addressed by fault tolerant mechanisms that have been widely investigated in the domain of safety-critical applications [15, 43]. Most mechanisms and techniques for fault tolerance affect software design, require disciplined programming and impact significantly on the overall software cost. While cost is not a problem in safety critical applications, it represents a strong limitation in many application domains, where they represent a major factor and disciplined design and implementation cannot be fully enforced.

Self-healing systems introduce approaches that can heal some classes of faults at runtime without impacting on design, coding and cost significantly. In this paper, we discuss the problems of building self-healing mechanisms and we illustrate a possible solution by introducing techniques for revealing failures, diagnosing faults and healing them.

2 AUTOMATICALLY HEALING FUNCTIONAL FAULTS

Many self-adaptive approaches, and in particular self-healing systems, share control loops as a core design element, as advocated as the key characteristic of engineering self-adaptive systems by Brun et al. [7]. Figure 1 shows the MAPE-K (monitor-analyze-plan-execute over a knowledge base) cycle as presented in one of the early papers on autonomic computing by Kephart and Chess [23]. The MAPE-K cycle assumes the availability of *sensors* and *effectors* to gather information about the

current behavior and influence the future behavior of the managed resources. It also requires functionality to *monitor* the sensed data, *analyze* the current status, and *plan* corrective actions to be *executed* through the effectors. The core functionalities share *knowledge* about the managed resources and their behavior. The shared knowledge is used to act properly.

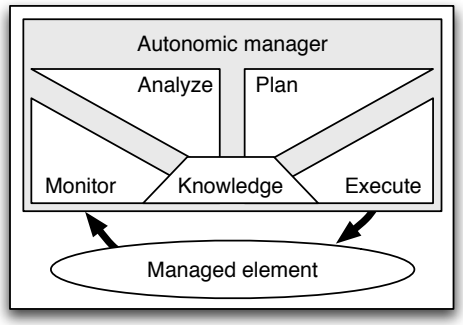


Fig. 1. The MAPE-K self-adaptive cycle as proposed by IBM [23]

In self-healing systems, monitors detect and, when possible, predict failures, analyzers diagnose faults and locate errors, and planners identify the healing strategies that must be executed. Each of these phases presents specific challenges and problems.

The self-healing cycle starts with detecting failures, which are actual behaviors that differ from the expected ones. Automatically detecting failures can be as easy as revealing system crashes or catching exceptions, but timely detection of functional failures can become very difficult. For example, even simple failures like the wrong computation of the cost of a service cannot be reduced to generic events, like crashes or exceptions, but require detailed knowledge of the expected results, like the correct cost of the services. Detecting software problems when the system fails may be too late for executing effective healing actions. For example, recovering from system crashes by restoring the user sessions may be hard or even impossible. Detecting erroneous states before the actual failures can prevent irrecoverable consequences and enable effective healing mechanisms. While many current healing mechanisms rely on simple failure detectors (e.g. [35]), in Section 4 we show how to create complex failure detectors with a mechanism to automatically translate relevant design level properties to effective code level assertions to early detect subtle runtime failures.

Once failures are detected or, even better, predicted, self-healing systems must automatically identify the corresponding faults, that is, code elements whose execution produces erroneous states and consequent system failures. Unfortunately the relation between faults and failures is not straightforward: The same fault can result in many failures, the same failure can be caused by several faults, faults may cause

failures only under specific execution conditions, faults may mask each other, and the effect of faults may result in failures only after long executions, as for instance memory leaks may cause failures long after the execution of the faulty code [34]. This is why locating faults is extremely difficult and time consuming. The recent impressive progress of debugging techniques has facilitated debugging [47], but human expertise still remains a key element of the debugging process. Self-healing approaches require techniques to automatically locate faults, but fortunately may rely on approximate information about fault locations. While developers need to identify the exact faulty statements to correct them, self-healing systems may use information about likely faulty modules to try to exclude them from the computation. Although some classic healing approaches, like checkpoint-and-recovery or reboot-based mechanisms, do not even try to locate the faults, but simply re-execute the code possibly under different conditions to try avoiding the problems [15, 8], locating possibly faulty modules with some precision can enable more effective healing mechanisms. In Section 5 we illustrate a mechanism to identify likely faulty components by comparing failing executions with models of correct behaviors.

To complete the self-healing control loop, self-healing systems must identify and execute suitable healing actions. Correcting the actual faults may be difficult, may require detailed semantic information about the systems, and may ask for specific knowledge of the faults that is not always produced by automatic fault localizers. Often self-healing systems do not fix the faults, which can be a hard task, but try to either avoid executing the faulty statements and thus preventing the failure occurrence, or recover from the effects of failures, which are often easier tasks. Classic healing approaches, like the ones based on checkpoint-and-recovery, reboot and exception handling, require little knowledge about the faults and their localization, but neither targets specific problems (checkpoint-and-recovery and reboot), and either reacts to emerging problems only to the extent the developers can predict them (exception handlers) [14]. New self-healing approaches try to exploit dynamic linking and system redundancy. For instance, several approaches to recover from service failures take advantage of the possibility of dynamically binding to different service implementations [4]. In Section 6 we show how to exploit the intrinsic redundancy of software systems to automatically recover from unexpected failures.

3 RUNNING EXAMPLE

In the next sections we illustrate the approaches to detect failures, diagnose and heal faults referring to a known fault that affected the Tomcat web server for several months, from version 6.0.0 to 6.0.10[1]. The fault concerns the initialization of a factory class: When a web application calls the factory method `JspFactory.getDefaultFactory()` during initialization, the method returns a null value that causes a `NullPointerException`, thus preventing the application startup.

[1] Tomcat Bugzilla bug database: ID 40820, `https://issues.apache.org/bugzilla/show_bug.cgi?id=40820`

Figure 2 shows the sequence of actions that lead to the failure during the Tomcat bootstrap: The method `Catalina.start()` initiates the web server instance, and starts the web applications installed on the server by calling the method `HostConfig.deployApps()`. After deploying the web application, Tomcat calls the method `StandardContext.start()` that loads the class `JspRuntimeContext`, and initializes the listeners of the web application.

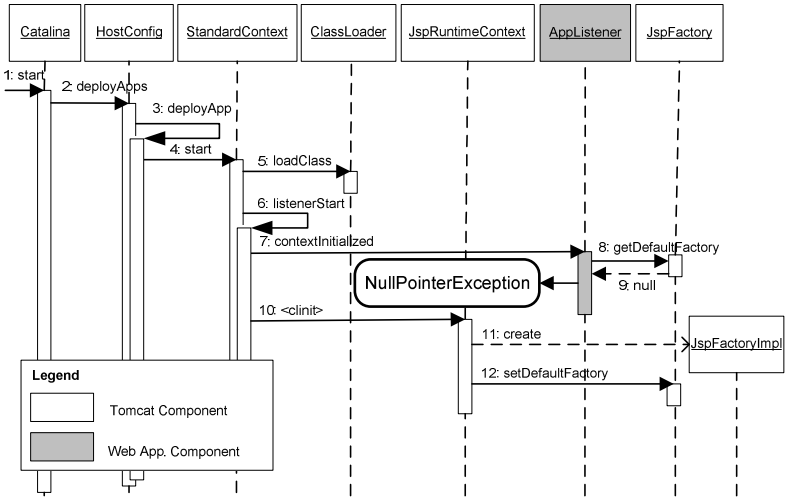


Fig. 2. The failing Tomcat bootstrap sequence

When, during the listener initialization, the web application invokes the method `JspFactory.getDefaultFactory()`, the method does not find an initialized JspFactory and returns null, thus triggering a `NullPointerException`. Tomcat catches the exception, suspends the deployment of the application, and completes the server initialization without starting the web application.

The static initializer of the class `JspRuntimeContext` is in charge of setting the default JspFactory. In Java, static initializers are invoked either just before the first use of the class or after calling the method `Class.forName(String)` to load the class.

During the Tomcat initialization, the class `JspRuntimeContext` is loaded by calling the method `ClassLoader.loadClass(String)`, which does not force the execution of the static initializer. Because of that, the singleton instance of class `JspFactory` is not set to the default value, and thus `getDefaultFactory()` returns a null value if invoked. Tomcat uses the class `JspRuntimeContext` before the termination of the initialization, thus it initializes the class `JspFactory` correctly for future uses.

Tomcat developers fixed the fault by replacing the call to `ClassLoader.loadClass(‘‘org.apache.jasper.compiler.JspRuntimeContext’’)` with a call to `Class.forName(‘‘org.apache.jasper.compiler.JspRuntimeContext’’)`: this

new invocation anticipates the class initialization and prevents the null pointer dereferencing.

4 DETECTING FAILURES

As discussed in Section 2, detecting failures or preferably erroneous states before the actual failures is a critical and hard functionality of self-healing systems. Many design and implementation approaches introduce assertions to monitor system executions, and detect violations of software properties at runtime. Assertions can detect erroneous states early, often before irrecoverable consequences, and have been shown to work well as fine-grained checks of invariants and pre- and post-conditions [38]. However, experiments indicate that it is often difficult to define the right set of assertions to monitor a given property [41]. Moreover, there are many useful design-level properties whose violations manifest as functional failures, and our experiments show that often design-level properties require many checks to be distributed across large parts of the code.

For example, the *Singleton* design pattern that prescribes a structural approach to solve a specific design problem, implicitly introduces the constraint that when the static examplar method is called, it will return a fully initialized instance of the singleton class [17]. The class `JspFactory` in Tomcat is a classical example for a singleton. At the design level, this is easy enough to understand and express, while at the code level, it results in many assertions that are spread widely in the code and are hard to get consistent and correct. In the implementation of Tomcat, the singleton instance does not follow the standard practice of the design pattern, and this implementation decision led to the fault discussed in the previous section. This is a typical example of functional problems due to the semantic gap between the design level and the systems implementation.

In this section, we illustrate an approach to bridge the gap between easily expressible design level properties and hardly manageable code level assertions. The approach provides *property templates* to guide the annotation of design models of the systems with the relevant design properties that are used to automatically generate the code level assertions needed to detect violations of the design properties before system failures [33]. Property templates are based on the observation that in many cases violations of constraints implied by design decisions manifest in similar kinds of systems failures, and encode the information needed to automatically generate monitors for typical design-level properties.

Table 1 illustrates some sample property templates that are identified by a name and a set of parameters, an informal description and some examples. Although property templates can be defined independently from the specific design notation, in this paper, we refer to design specifications expressed as UML specifications, and we represent property templates as stereotypes in a suitable UML profile. Software designers annotate the UML model of the system with stereotypes expressing the desired properties. The stereotypes are mapped onto assertions encapsulated

in aspects and inserted into the system using dynamic load-time weaving. Using these techniques, adding monitors to any system is non-intrusive and can also check properties that involve third-party libraries.

Property	Description	Example Specification
`immutable`	A constrained entity may not change its visible state once it is created.	[...] it is illegal for the calling Thread to attempt to modify the `ServletResponse` object.
`initialized`	A constrained entity must complete all custom initialization before becoming accessible to clients.	The `initialize()` method is called to initialize an uninitialized `PageContext` so that it may be used by a JSP implementation.
`language <L>`	A constrained entity must be a string and must match a regular expression defining the language `L`.	`parseExpression()` prepares an expression for later evaluation.
`unique`	A constrained entity must be unique within its context. If the constrained entity is a relation, tuples in the relation must be unique.	Each tag in a JSP page has a unique `jspId`.
`explicit <I>`	A constrained class must directly implement interface `I`.	

Table 1. Selected classes of constraints for property templates and examples taken from the Java Server Pages specification

Figure 3 shows the excerpt of the Tomcat design with class `JspFactory` annotated with the `<<initialized>>` stereotype. The excerpt includes all the information required to generate the runtime monitor. The `<<initialized>>` stereotype takes an optional parameter that allows developers to specify which method of the annotated class is considered to be the *initializer*. The generated runtime monitor checks that the initializer method has been called at least once before any other method of the class is used.

The code generated for the stereotype consists of two parts: 1) infrastructure classes and callbacks that allow detailed monitoring of the system state, and 2) assertions checking invariants and method pre- and post-conditions over the system state together with the additional information gathered by the monitoring infrastructure.

The code generated for the `<<initialized>>` property does not require additional monitoring infrastructure. Figure 4 shows the most relevant parts of the generated code. The pointcut in lines 5 and 6 specifies the *initializer*, and the corresponding aspect in lines 7 and 8 sets a hidden flag when this method is called. The pointcut in lines 11–13 captures all other methods of the annotated class and the aspect in lines 14–18 checks that the *initializer* has been called. When the property is violated, that is the `initialized` flag is `false`, the fault diagnosis component

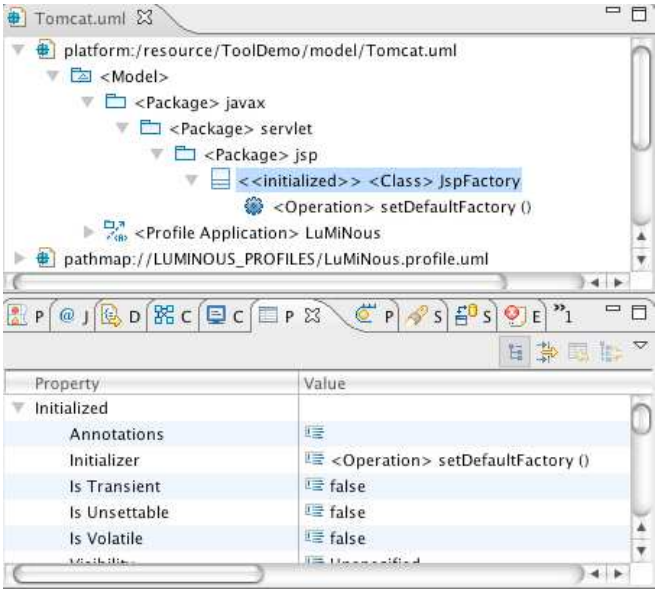


Fig. 3. Excerpt of the UML design of Tomcat annotated with the `<<initialized>>` stereotype

```
1  public aspect JspFactory_Initialized {
2
3    private boolean initialized = false;
4
5    pointcut initMethod():
6      call(static * JspFactory.setDefaultFactory(..));
7    after() returning: initMethod(){
8      initialized=true;
9    }
10
11   pointcut checkedMethods():
12     call(static * JspFactory.*(..))
13     && !cflow(initMethod());
14   before(): checkedMethods() {
15     if (!initialized){
16       //notify fault localization
17     }
18   }
19 }
```

Fig. 4. Excerpt of the code generated for `JspFactory`

is notified. The monitor passes on information about the violated property, the current stack trace, and references to the objects involved in the violation.

5 LOCATING FAULTS

When detecting failures or erroneous states, for example invariant violations, self-healing systems must locate the faults. As discussed in the former sections, detecting faults is difficult because of the complex relation between failures and faults, and because of the unpredictable distance between symptoms of failures and the execution of faulty program elements. Debugging approaches produce accurate results, but require human judgment. Fault localization techniques trade precision for automation: They may identify the faulty elements with some degree of imprecision or uncertainty, but do not need human intervention.

In this section, we illustrate a fault localization approach that locates execution elements that likely led to erroneous states by comparing failing to correct executions. The method relies on models of correct program behavior, and locates possible faults by studying the violations of models during failing executions.

The availability of accurate and detailed models of correct program behavior is crucial for locating faults. We build accurate albeit partial models of correct program behavior automatically by analyzing successful program executions, for example while running a successful system test suite.

When executing successful test cases, we capture interactions between components and we record both the sequences of inter-component method invocations and the data exchanged between components. We record all the outgoing invocations that can be observed when a component service is executed. For instance, when Tomcat executes the method `start()` implemented by `Catalina`, we may record the sequence `Lifecycle.start()`, `CatalinaShutDownHook(Catalina)`, `StandardServer()`, `StandardServer.initialize()`, `StandardServer.await()`.

We record also the data values passed as parameters and the return values. When the data to be recorded are objects, we recursively extract the values stored within their attributes. For example, when monitoring invocations to method `MBeanServer.isRegistered` that accepts an object `ObjectName` as parameter and returns a Boolean value, we record both the return value (for example `returnValue = false`) and the data stored by the attributes of the `ObjectName` object (for example, `ObjectName._canonicalName="Catalina"`, `ObjectName.compat = true`).

We use the information about sequences of method invocations and data exchanged during interactions to infer models that summarize and generalize the observed executions. We generalize observations because concrete values observed during testing may not correspond completely to the values observed in the field, but can be used to identify general properties that should hold for all legal executions. We generate two types of models that capture properties: 1) Boolean properties on parameter and return values that capture properties about the data exchanged by components, and 2) finite state automata that specify sequences of method calls.

We generate Boolean properties on parameters and return values by elaborating the data exchanged during executions with Daikon [16]. For example, Daikon can elaborate the values of the parameters of invocations of method `JspFactory.getDefaultFactory()`, and can automatically generate the properties `returnValue != null` and `returnValue.pool.current < returnValue.pool.max`, which indicate that method `JspFactory.getDefaultFactory()` never returned `null`, and that the current size of its `pool` attribute (defined in field `returnValue.pool.current`) is always less than its maximum allowed size (defined in `returnValue.pool.max`).

We generate finite state automata that represent the observed component interactions with kBehavior [30]. For example, kBehavior can elaborate the sequences of calls observed during the initialization of class `JspRuntimeContext`, and can automatically generate the simple finite state automaton shown in Figure 5, that models the only sequence of interactions observed during the initialization.

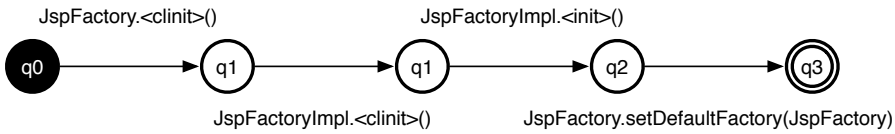


Fig. 5. The finite state automaton for the `JspRuntimeContext` class initializer

These models are neither complete nor consistent, but identify partial properties of correct behaviors that can help us automatically locate potential faults. We locate faults automatically by comparing executions that produce erroneous states to the models built during correct executions. Executions that lead to erroneous states differ from correct executions, and thus violate one or more models of correct behaviors. We assume that model violations are related to the execution of faulty components. Our experiments confirm that executions that produce erroneous states violate several models. For example, Table 2 shows the model violations, hereafter *anomalies*, during the failing initialization of a factory class discussed in Section 3.

ID	Method	Description
A1	StandardContext.start()	Unexpected call to JspFactory.<clinit> in state q28
A2	JspFactory.getDefaultFactory()	Unexpected null value returned
A3	JspRuntimeContext<clinit>()	Missing call to JspFactory<clinit>in state q0
A4	ThreadPool$ControlRunnable.run()	Unexpected call to Logger.getLog() in state q2

Table 2. Model violations (*anomalies*) detected by BCT

In general we may observe many model violations (anomalies) that may not be easy to analyze. We also observe that not all anomalies relate to the execution

of faulty elements, and that relevant anomalies may correspond to different fault locations. We filter irrelevant anomalies by ignoring anomalies that are observed in both successful and failing executions, and thus are likely false positives.

We then aggregate anomalies according to likelihood that they refer to a same run-time problem. We observe that an initial anomaly often causes a cascade of anomalies. For example an exception may violate many interaction and data models, before being handled by the application. Thus, we cluster anomalies according to their mutual distance on the dynamic call tree. The distance between two anomalous events detected during the execution of methods `m1` and `m2` is measured as the minimum number of nodes that need to be traversed to move from the node that corresponds to `m1` to the node that corresponds to `m2` in the dynamic call tree [1] of the failing execution. We then cluster anomalies with the Within Clustering Dispersion algorithm [18]. Each cluster represents a possible explanation of a different run-time problem and thus locate a fault.

We prioritize clusters of anomalies according to their size, based on the observation that complex clusters frequently describe complex and highly unexpected executions incorrectly handled by the system. The nodes of the cluster are ordered according to occurrence of the corresponding events. Each cluster is characterized by at least a root node and a set of links that indicate the cause-effect relation between anomalies. Figure 6 shows the two clusters identified for the Tomcat case study. The bigger cluster includes the methods responsible for the investigated failure, and provides enough information to approximately locate the fault.

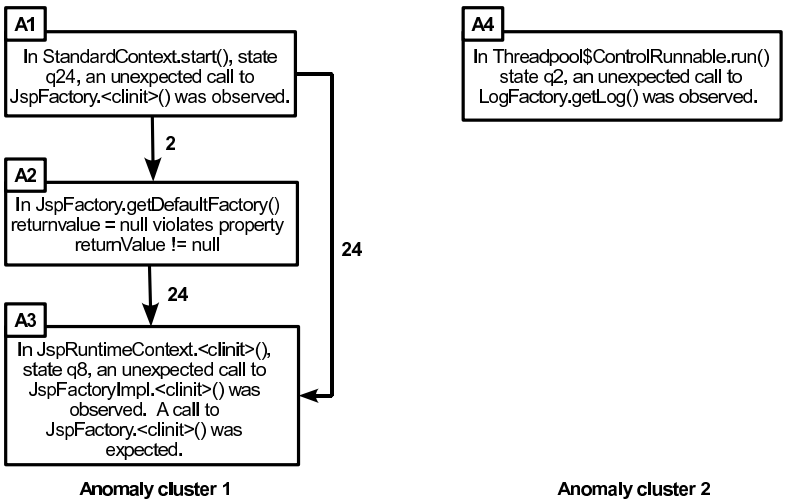


Fig. 6. Clustered anomalies for the Tomcat case study

6 HEALING FAULTS

Once one or more faults have been (partially) located in the code, self-healing systems try to automatically heal them. In the former sections, we discussed the difficulty of finding precise fixes and the limitations of classic debugging approaches that often require human judgement, and we showed how automatic fault localization techniques may locate faults only with some degree of imprecision. This is why self-healing systems often investigate easier alternatives to precise fault fixing, in particular fault avoidance and error recovery. Fault avoidance approaches try to exclude the likely faulty elements from the executions by looking for alternative elements, for example by dynamically binding to new services equivalent to the faulty ones. Error recovery approaches try to solve the problems caused by faults, for example by executing suitable exception handlers. The choices of healing approaches depend on the kind of faults, the precision of the information related to the faults and the type of the applications. For example, exception handlers are simple healing approaches that can deal with predefined classes of faults identified by classic exception mechanisms, but cannot deal with unpredictable failures of unknown nature.

In this section, we illustrate a technique that exploits the implicit redundancy of software systems to automatically identify alternative executions, and exclude faulty components. The technique requires the system to be in a correct state, either because interrupted before executing a faulty component or because rolled back after a faulty execution, and some information about the failing sequence, like the one produced by the fault localization technique discussed in the former section [10].

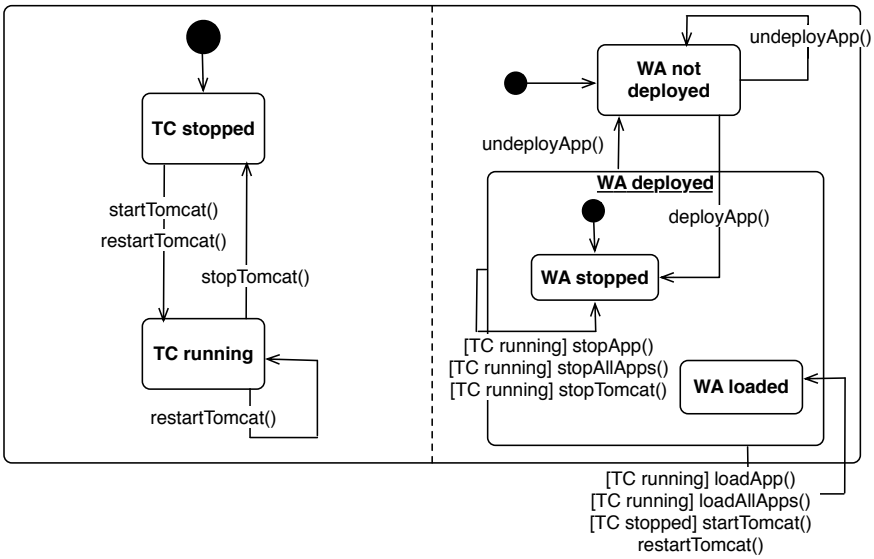
The approach is grounded on the observation that often complex software systems implement similar functionalities in different ways. For example in Tomcat, the operations `startTomcat`, `restartTomcat`, `loadApp(app)` and `loadAllApps()` load deployed applications, and if suitably invoked, they may provide alternative operations when some of them fail partially or completely.[2].

Sequences of operations are equivalent to others if they have the same intended effect of the other operations according to the specifications. We define such sequences of operations as *specification-equivalent*. Since the equivalence holds at the specification level, the sequences of operations may not be equivalent in the actual implementation. When a particular sequence fails due to a fault, there might be another specification-equivalent sequence that succeeds because it is not affected by the fault. We call this particular sequence a *workaround*. Thus a workaround

[2] These operations, as all the operations listed in Figure 7, are macro-operations that map to more complex sequences of actual implemented methods. The operation `startTomcat()`, for example, is a concise way to represent the actual sequence of methods that is executed when Tomcat is started: `Lifecycle.start()`, `CatalinaShutDownHook(Catalina)`, `StandardServer()`, `StandardServer.initialize()`, `StandardServer.await()`

is a sequence of operations that is both specification-equivalent to the sequence of operations that leads to a failure, and does not cause failures during the execution.

We look for workarounds by automatically identifying sequences of operations equivalent to the failing ones, and selecting one that does not fail. We identify equivalent sequences from (partial) specifications of the system. The technique depends on the specification framework. Here we illustrate the approach referring to the Statecharts specifications of the Tomcat Web application loader shown in Figure 7.



`startTomcat()`	Start the servlet container. All deployed Web applications are loaded
`stopTomcat()`	Stop the servlet container. All running Web applications are stopped
`restartTomcat()`	The servlet container is stopped and restarted
`deployApp(app)`	Deploy a Web application
`undeployApp(app)`	Undeploy a Web application
`loadApp(app)`	Load a Web application
`stopApp(app)`	Stop a Web application
`loadAllApps()`	Load all deployed Web applications
`stopAllApps()`	Stop all running Web applications

Fig. 7. The Statecharts specification of the Tomcat Web applications loader

In Statecharts specifications, sequences of operations correspond to paths between states. We consider two sequences of operations to be equivalent if the paths both start from the same state and terminate in the same state. For example in the Statecharts of Figure 7, the paths `deployApp(),startTomcat()` and `startTomcat(),deployApp(),loadApp()` start from the AND-state ⟨`TC stopped, WA not deployed`⟩ and terminate in the AND-state ⟨`TC running, WA loaded`⟩,

and are thus equivalent. Even in simple specifications, there may be many equivalent sequences. Table 3 lists some sample equivalent sequences for some states of the Statecharts shown in Figure 7. Equivalent sequences can be inferred automatically from a (partial) state based specification.

⟨**Tomcat stopped, Web Application not deployed**⟩		
startTomcat()	≡	restartTomcat()
startTomcat()	≡	startTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡	startTomcat(), restartTomcat()
startTomcat()	≡	startTomcat(), restartTomcat(), stopTomcat(), startTomcat()
restartTomcat()	≡	startTomcat()
restartTomcat()	≡	startTomcat(), stopTomcat(), startTomcat()
deployApp()	≡	deployApp(), undeployApp(), deployApp()
deployApp(), startTomcat()	≡	**startTomcat(), deployApp(), loadApp()**
...		...

⟨**Tomcat running, Web Application not deployed**⟩		
restartTomcat()	≡	stopTomcat(), startTomcat()
restartTomcat()	≡	stopTomcat(), restartTomcat()
restartTomcat()	≡	restartTomcat(), stopTomcat(), startTomcat()
deployApp()	≡	deployApp(), undeployApp(), deployApp()
deployApp()	≡	deployApp(), loadApp(), stopApp()
deployApp(), loadApp()	≡	deployApp(), stopApp(), loadApp()
...		...

⟨**Tomcat stopped, Web Application deployed**⟩		
startTomcat()	≡	restartTomcat()
startTomcat()	≡	startTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡	startTomcat(), restartTomcat()
startTomcat()	≡	startTomcat(), restartTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡	**startTomcat(), loadApp()**
startTomcat()	≡	startTomcat(), loadAllApps()
...		...

⟨**Tomcat running, Web Application deployed**⟨		
restartTomcat()	≡	stopTomcat(), startTomcat()
restartTomcat()	≡	stopTomcat(), restartTomcat()
restartTomcat()	≡	restartTomcat(), stopTomcat(), startTomcat()
loadApp()	≡	stopApp(), loadApp()
stopApp()	≡	stopAllApp()
stopApp()	≡	loadAllApp(), stopAllApp()
...		...

Table 3. Some equivalent sequences that can be inferred from the Statechart in Figure 7

Not all specification-equivalent sequences are valid workarounds. Many may execute the same or other faulty modules and thus lead to failures. Randomly executing equivalent-sequences may be extremely inefficient, thus we need effective ways to identify valid workarounds. In general, we cannot identify workarounds without information on the actual code execution, but we can prioritize specification-equivalent sequences to identify the sequences that are most likely valid workarounds. Priority policies may be based on the occurrences of modules that are marked as likely faulty, differences with respect to the failing sequence, length of the sequence and history information about the occurrence of subsequences in successful executions. The sequences shown in bold in Figure 7 are valid workarounds for the problem with

the Tomcat Web application loader illustrated in Section 3. They are all given high priority by simple algorithms based on the length of the sequences and the information about faulty modules produced by the technique illustrated in the previous section.

7 RELATED WORK

Autonomic and self-managed systems have been widely studied in the last years. The interested readers can refer to the paper by Huebscher and McCann for a recent survey on autonomic computing [21].

In this paper, we advocate a self-healing cycle composed of techniques to detect failures, locate and heal faults and we illustrate the feasibility of each phase through example techniques. In this section, we survey the main techniques that support the three main phases of the self-healing cycle.

Detecting failures is the main objective of testing and analysis techniques, but most testing and analysis research focuses on test case generation and code analysis, and assumes suitable oracles exist, while automatic failure detection relies on runtime monitoring and checks [34].

Runtime monitoring covers a wide range of applications, for example monitoring service-level agreements or structural and architectural constraints of systems. Wang and Shen describe an approach to detect violations of constraints intrinsic in UML class diagrams [42]. They instrument Java programs with calls to a monitoring infrastructure that uses assertion-like statements to check if the running system maintains invariants like association multiplicities. Stirewalt and Rugaber present an approach to *enforce* OCL invariants at runtime [40]. They use specified invariants to generate wrappers around the classes that are referred to in the invariants. These wrappers notify all classes involved in the constraints about changes in values, and suitably update the related objects.

Many researchers have explored the usefulness of assertions as runtime checkers [31, 38]. The obvious value added to software systems by assertions encouraged the development of several industrial strength assertion languages (for example JML and Spec# [5, 26]). While the usefulness of well-designed code assertions is generally undisputed, these assertions are typically created by developers while they implement systems. Voas and Miller address the question at which locations in the code assertions are most effective [41]. Their results indicate that intuitive solutions, which developers may come up with, are not always optimal. Assertion languages are well suited to express assertions at specific program locations that represent implementation decisions, but do not easily capture design-level decisions that may involve several program locations. In this paper we introduce a technique to deal effectively with software requirements that constrain the whole system, and involve several design components.

Locating faults is the main objective of debugging techniques, but most debugging techniques assume the ability to repeatedly execute the faulty system in

a controlled testing environment and ultimately rely on human judgement [47]. The self-healing cycle must rely on fully automatic approaches and cannot assume full repeatability of system executions. Recently, some research effort has been devoted to the development of automated and semi-automated fault localization techniques that focus on the identification of either suspicious code blocks or faulty behaviors.

Techniques that focus on the identification of faulty code blocks use coverage information in order to identify the instructions that likely caused the failures [37, 22]. These techniques typically return a list of possible faulty statements as result, the returned list is sorted according to the likelihood of a statement to be faulty and the list must be inspected by the developers. Such techniques considerably reduce the effort required to locate faults, but are not completely automated yet.

Techniques that focus on the identification of anomalous behaviors, like the one proposed in this paper, use automatically inferred models in order to identify model violations at runtime and present such violations as possible explanations for the failure cause [19, 46, 44].

Behavioral models are inferred using data recorded during monitored executions at testing time. There exist many techniques for dynamic model inference, even if not all of them focus on the usage of such models to identify anomalies: Daikon derives Boolean expressions describing relations between program variables [16]; Several approaches derive finite state automata that identify constraints on the ordering of events [6, 13, 29]. The empirical nature of the inferred models produces many false positives that reduce the efficacy of such techniques. In this paper we discussed a technique to filter false positives and cluster related anomalies based on the distance in the dynamic call tree.

Healing faults and avoiding failures that may derive from faults have been core goals of research in fault tolerant systems. Classic fault handling approaches rely on design diversity principles, which lead to the development of several independently designed and implemented versions of the same components. Classic approaches include N-version programming [3], recovery-blocks [36] and self-checking programming [25]. N-version programming, recovery-blocks, and self-checking programming mechanisms have been extended to different domains, in particular to Web services [28].

Some recent approaches to automatic fault healing rely on the use of *registries* and *wrappers*. Registries augment applications with lists of failures and corresponding recovery actions [4, 32]. Wrappers capture the interactions between components to solve integration problems [11, 39]. Both registries and wrappers can effectively handle failures at runtime under the assumption that these elements have been designed to cope with the erroneous conditions that led to the failures.

Many approaches have been proposed to cope with non-deterministic failures. *Checkpoint and recovery* techniques periodically save consistent states to be used as safe roll backs [15]. When a failure occurs, the system is brought back to the latest consistent state, and the following operations are re-executed. These approaches solve temporary problems that may have been caused by accidental, transient conditions in the environment. Qin et al. improved checkpoint and recovery by partially

re-executing failing programs under modified environment conditions to recover from deterministic as well as non deterministic faults [35]. Similarly Candea et al. propose micro-reboots as an efficient way to cope with non-deterministic failures, in particular memory leaks [8]. The intuition that brought Candea et al. to propose micro-reboots were already present in previous approaches of software rejuvenation that rely on the observation that some software systems fail due to *age*, and that proper system reinitializations can avoid such failures [43].

Recently both Weimer et al. and Arcuri et al. investigated *genetic programming* as a way to automatically fix software faults [45, 2]. When the software system fails, the run-time framework automatically generates a population of variants of the original faulty program. Genetic algorithms evolve the initial population guided by the results of test cases that select a new *correct* version of the program.

8 CONCLUSIONS

Modern software systems are growing in complexity and importance. Classic testing and analysis techniques are still extremely useful, but cannot cope with many new problems that derive from specific configuration and environment conditions that can change dynamically at run-time. In this paper, we indicate self-healing approaches as a way to cope with new run-time problems in a cost-effective way. We introduced the self-healing cycle as an instance of the MAPE-K autonomic cycle. We indicated techniques to detect failures, locate and heal faults, and we illustrated them with a common example, a well known Tomcat problem.

The design and implementation of self healing cycles open additional problems related to the check for consistency of the healing solutions. The design of sophisticated self-healing systems requires new approaches to protect the system from new hazards that may be induced by unplanned actions of healing systems and unexpected interactions of different healing mechanisms that may act on different parts of the same open system.

REFERENCES

[1] Ammons, G.—Ball, T.—Larus, J. R.: Exploiting Hardware Performance Counters With Flow and Context Sensitive Profiling. In: Proceedings of the Conference on Programming Language Design and Implementation, Las Vegas, NV, USA, 1997, pp. 85–96.

[2] Arcuri, A.—Yao, X.: A Novel Co-Evolutionary Approach to Automatic Software Bug Fixing. In: CEC '08: Proceedings of the IEEE Congress on Evolutionary Computation, 2008, pp. 162–168.

[3] Avizienis, A: The N-Version Approach to Fault-Tolerant Software. IEEE Transactions on Software Engineering, Vol. 11, 1985, No. 12, pp. 1491–1501.

[4] Baresi, L.—Guinea, S.—Pasquale, L.: Self-Healing BPEL Processes with Dynamo and the JBoss Rule Engine. In: ESSPE '07: International Workshop on Engi-

neering of Software Services for Pervasive Environments, New York, NY, USA, 2007, pp. 11–20.

[5] Barnett, M.—Leino, K. R. M.—Schulte, W.: The Spec# Programming System: An Overview. In: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Systems, CASSIS 2004, 2004, pp. 49–69.

[6] Biermann, A.—Feldman, J.: On the Synthesis of Finite State Machines from Samples of Their Behavior. IEEE Transactions on Computer, Vol. 21, June 1972, pp. 592–597.

[7] Brun, Y.—di Marzo Serugendo, G.—Gacek, C.—Giese, H.—Kienle, H.—Litoiu, M.—Müller, H.—Pezzè, M.—Shaw, M.: Engineering Self-Adaptive Systems Through Feedback Loops. In: Software Engineering for Self-Adaptive Systems, 2009, pp. 48–70.

[8] Candea, G.—Kiciman, E.—Zhang, S.—Keyani, P.—Fox, A.: JAGR: An Autonomous Self-Recovering Application Server. In: Active Middleware Services, IEEE Computer Society, 2003, pp. 168–178.

[9] Carzaniga, A.—Denaro, G.—Pezzè, M.—Estublier, J.—Wolf, A.: Toward Deeply Adaptive Societies of Digital Systems. In: ICSE '09 Companion: Proceedings of the 31st International Conference on Software Engineering, Vancouver, CA, 2009.

[10] Carzaniga, A.—Gorla, A.—Pezzè, M.: Healing Web Applications Through Automatic Workarounds. International Journal on Software Tools for Technology Transfer, Vol. 10, December 2008, No. 6, pp. 493–502.

[11] Chang, H.—Mariani, L.—Pezzè, M.: In-Field Healing of Integration Problems with COTS Components. In: ICSE '09: Proceeding of the 31st International Conference on Software Engineering, Vancouver, CA, 2009, pp. 166–176.

[12] Clause, J.—Orso, A.: A Technique for Enabling and Supporting Debugging of Field Failures. In: ICSE '07: Proceedings of the 29th IEEE and ACM SIGSOFT International Conference on Software Engineering, Minneapolis, MN, USA, 2007, pp. 261–270.

[13] Cook, J. E.—Wolf, A. L.: Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology, Vol. 7, 1998, No. 3, pp. 215–249.

[14] Cristian, F.: Exception Handling and Software Fault Tolerance. IEEE Transactions on Computer, Vol. 31, 1982, No. 6, pp. 531–540.

[15] Elnozahy, M.—Alvisi, L.—Wang, Y.—Johnson, D. B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Computing Surveys, Vol. 34, 2002, No. 3, pp. 375–408.

[16] Ernst, M. D.—Cockrell, J.—Griswold, W. G.—Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE Transactions on Software Engineering, Vol. 27, February 2001, No. 2, pp. 99–123.

[17] Gamma, E.—Helm, R.—Johnson, R.—Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994.

[18] Gordon A.: Classification. Chapman and Hall/CRC, 2nd edition, 1999.

[19] Hangal, S.—Lam, M. S.: Tracking Down Software Bugs Using Automatic Anomaly Detection. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, Orlando, FL, USA, 2002, pp. 291–301.

[20] Horn P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, International Business Machines, 2001.

[21] Huebscher, M. C.—McCann, J. A.: A Survey of Autonomic Computing – Degrees, Models, and Applications. ACM Computing Surveys, Vol. 40, 2008, No. 3, pp. 1–28.

[22] Jones, J. A.—Harrold, M. J.—Stasko, J. T.: Visualization of Test Information to Assist Fault Localization. In: ICSE '02: Proceedings of the Internation Conference on Software Engineering, Orlando, FL, USA, 2002, pp. 467–477.

[23] Kephart, J. O.—Chess, D. M.: The Vision of Autonomic Computing. IEEE Computer, Vol. 36, 2003, No. 1, pp. 41–50.

[24] Kramer, J.—Magee, J.: Self-Managed Systems: An Architectural Challenge. In: Future of Software Engineering, FOSE '07, 2007, pp. 259–268.

[25] Laprie, J.-C.—Béounes, C.—Kanoun, K.: Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. IEEE Computer, Vol. 23, 1990, No. 7, pp. 39–51.

[26] Leavens, G. T.—Baker, A. L.—Ruby, C.: JML: A Notation for Detailed Design. In: Haim Kiloc, Bernhard Rumpe, and Ian Simmonds, editors, Behavioral Specifications of Businesses and Systems, Chapter 12, Kluwer, 1999, pp. 175–188.

[27] Li, P. L.—Ni, M.—Xue, S.—Mullally, J. P.—Garzia, M.—Khambatti, M.: Reliability Assessment of Mass-Market Software: Insights from Windows Vista®. In: ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering, Washington, DC, USA, 2008, pp. 265–270.

[28] Looker, N.—Munro, M.—Xu, J.: Increasing Web Service Dependability Through Consensus Voting. In: COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference, COMPSAC '05, Volume 2, Washington, DC, USA, 2005, pp. 66–69.

[29] Lorenzoli, D.—Mariani, L.—Pezzè, M.: Automatic Generation of Software Behavioral Models. In: ICSE '08: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 2008, pp. 501–510.

[30] Mariani, L.—Pezzè, M.: Dynamic Detection of COTS Components Incompatibility. IEEE Software, Vol. 24, 2007, No. 5, pp. 76–85.

[31] Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 1988.

[32] Modafferi, S.—Mussi, E.—Pernici, B.: SH-BPEL: A Self-Healing Plug-In for WS-BPEL Engines. In: MW4SOC '06: Proceedings of the 1st Workshop on Middleware for Service Oriented Computing, New York, NY, USA, 2006, pp. 48–53.

[33] Pezzè, M.—Wuttke, J.: Automatic Generation of Runtime Failure Detectors from Property Templates. In: Betty H. C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, Software Engineering for Self-Adaptive Systems, Springer Verlag, 2009, pp. 229–264.

[34] Pezzè, M.—Young, M.: Software Test and Analysis: Process, Principles and Techniques. John Wiley and Sons, 2008.

[35] Qin, F.—Tucek, J.—Zhou, Y.—Sundaresan, J.: Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. ACM Transactions on Computer Systems, Vol. 25, 2007, No. 3, pp. 1–33.

[36] Randell, B.: System Structure for Software Fault Tolerance. In: Proceedings of the International Conference on Reliable Software, New York, NY, USA, 1975, pp. 437–449.

[37] Renieris, M.—Reiss, S. P.: Fault Localization With Nearest Neighbor Queries. In: Proceedings of the Internation Conference on Automated Software Engineering, 2003, pp. 30–39.

[38] Rosenblum, D. S.: A Practical Approach to Programming With Assertions. IEEE Transactions on Software Engineering, Vol. 21, 1995, No. 1, pp. 19–31.

[39] Salles, F.—Rodriguez, M.—Fabre, J.-C.—Arlat, J.: Metakernels and Fault Containment Wrappers. In: International Symposium on Fault-Tolerant Computing, Los Alamitos, CA, USA, 1999, pp. 22–29.

[40] Stirewalt, K.—Rugaber, S.: Automated Invariant Maintenance via OCL Compilation. In: 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2005, 2005, pp. 616–632.

[41] Voas, J. M.—Miller, K. W.: Putting Assertions in Their Place. In: Proceeding of the 5th International Symposium on Software Reliability Engineering (ISSRE), 1994, pp. 152–157.

[42] Wang, K.—Shen, W.: Runtime Checking of UML Association-Related Constraints. In: Proceedings of the 5th International Workshop on Dynamic Analysis, 2007, p. 3.

[43] Wang, Y.-M.—Huang, Y.—Vo, K.-P.—Chung, P.-Y.—Kintala, C.: Checkpointing and Its Applications. In: FTCS '95: Proceedings of the 25th International Symposium on Fault-Tolerant Computing, Washington, DC, USA, 1995, pp. 22–31.

[44] Wasylkowski, A.—Zeller, A.—Lindig, C.: Detecting Object Usage Anomalies. In: Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 2007, pp. 35–44.

[45] Weimer, W.—Le Goues, C.—Nguyen, T.—Forrest, S.: Automatically Finding Patches Using Genetic Programming. In: ICSE '09: Proceeding of the 31st International Conference on Software Engineering, Vancouver, CA, 2009, pp. 364–374.

[46] Yilmaz, C.—Paradkar, A.—Williams, C.: Time Will Tell: Fault Localization Using Time Spectra. In: ICSE '08: Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 2008, pp. 81–90.

[47] Zeller, A.—Hildebrandt, R.: Simplifying and Isolating Failure-Inducing Input. IEEE Transactions on Software Engineering, Vol. 28, 2002, No. 2, pp. 183–200.

Alessandra Gorla is a Ph. D. candidate in computer science at the University of Lugano (Switzerland). She holds a Bachelor and M. Sc. degree in computer science, both from the University of Milano Bicocca (Italy). Her research interests include software testing and analysis, and self-healing systems.

Mauro Pezzè is a Professor of computer science at the University of Milano Bicocca (Italy) and at the University of Lugano (Switzerland). He is associate editor of ACM Transactions on Software Engineering and Methodology and member of the Steering Committee of the ACM International Conference on Software Testing and Analysis.

Jochen Wuttke is a Ph. D. candidate in computer science at the University of Lugano (Switzerland). He holds an M. Sc. degree in computer science from the University of Munich (Germany) and an MBA from the Kent Business School at Canterbury, UK. His research interests include software quality, autonomic systems, and formal approaches to software engineering.

Leonardo Mariani is a researcher at the University of Milano Bicocca (Italy). His research interests include software engineering, testing and analysis of component-based systems, dynamic analysis, design and development of self-healing solutions, test and analysis of service-based applications, and design and development of autonomous and adaptive systems.

Fabrizio Pastore is a Ph. D. candidate in informatics at the University of Milano Bicocca (Italy). His research interests include dynamic analysis and self-healing systems. He is currently working on the development of a self-healing framework as part of the EU funded project Shadows.