A Self-Healing Approach for Object-Oriented Applications

A.R. Haydarlou, B.J. Overeinder, and F.M.T. Brazier
Department of Computer Science, Vrije Universiteit Amsterdam
de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
{rezahay,bjo,frances}@cs.vu.nl

Abstract

In this paper, we present our approach and architecture for fault diagnosis and self-healing of interpreted object-oriented applications. By combining aspect-oriented programming, program analysis, artificial intelligence, and machine learning techniques, we advocate that our approach can heal a significant number of failures of real interpreted object-oriented applications.

1. Introduction

Today's increasingly complex systems, composed of a variety of components, operating in large-scale distributed heterogeneous environments, require more and more human skills to install, configure, tune, and maintain. Determining the root cause of software runtime failures in such complex systems is hindered by the lack of appropriate diagnostic feedback, where often system logs are the only information available. Automated support is clearly beneficial.

Ideally such complex systems would be able to recognize and solve a large portion of these errors on their own. To this purpose, these systems would need (1) to be self-aware, (2) to know when and where an error state occurs, (3) to have adequate knowledge to stabilize themselves, (4) to be able to analyze the problem situation, (5) to make healing plans, (6) to suggest various solutions to the system administrator, and (7) to heal themselves without human intervention.

Autonomic computing [8] has been proposed as a way to reduce the cost and complexity of systems, to control their manageability and to achieve the above desired situation. *Self-management* is central to autonomic computing and encompasses four self-* tasks: (1) self-configuring, (2) self-healing, (3) self-optimizing, and (4) self-protecting. Self-management assumes two subsystems: (1) the managed system, the application, containing the business functionality, and (2) the autonomic manager, which monitors the situation and performs the self-* tasks. In this paper, the terms managed system and application are used interchangeably. Self-healing is the self-task addressed in this paper.

The approach presented in this paper combines a number of techniques to successfully diagnose failures and to enable self-healing. The main techniques used are (1) aspect-oriented programming [5], (2) static and dynamic program analysis and model checking [6], and (3) artificial intelligence and machine learning [7]. Aspect-oriented programming is used to instrument two types of sensors (regarding failure and application model information) in the application compiled code. The program analysis techniques utilize the sensor information to determine the root cause of application failures. Finally, artificial intelligence and machine learning techniques use the result of the analysis phase to make healing plans, predict the impact of healing actions, or generate suggestions for healing.

The logs of a number of complex systems in a financial enterprise provided insight in the problems that occur on a daily basis. These logs were spread throughout different files in different formats on different machines, making it time consuming and difficult for skilled personnel to localize program failure, analyze the cause(s), and to come up with a solution. A significant part of the logs was related to a small set of most frequently occurring failures (exceptions), such as *ClassNotFound*, *NullPointer*, *ClassCast*, *Naming*, *FileNotFound*. This paper demonstrates how a significant number of such application failures can be automatically diagnosed and healed using our approach.

The remainder of this paper is organized as follows. Sections 2 and 3 present our approach and architecture. Section 4 describes our experiment showing how code is automatically instrumented. Finally, Section 5 presents related work and discussion, and lays out future work.

2. Approach

The ultimate goal in self-healing systems is to equip current (legacy) interpreted object-oriented distributed applications with a technique with which they can determine by themselves the root cause of their software runtime failures

and make plans or suggestions for healing these failures. To reach this goal, the following design criteria are considered:

1. No application source code is required.
2. Performance overhead should be acceptable.
3. Only software runtime failures that are not caught by the application are handled.
4. For those failures that can not be healed, the cause should be diagnosed.
5. All healing actions should be reported to system administrators.

As stated above, the scope of this paper is self-healing of distributed interpreted object-oriented systems. To make this kind of applications self-aware, the following sensor types are instrumented at the strategic positions in the application that report different information:

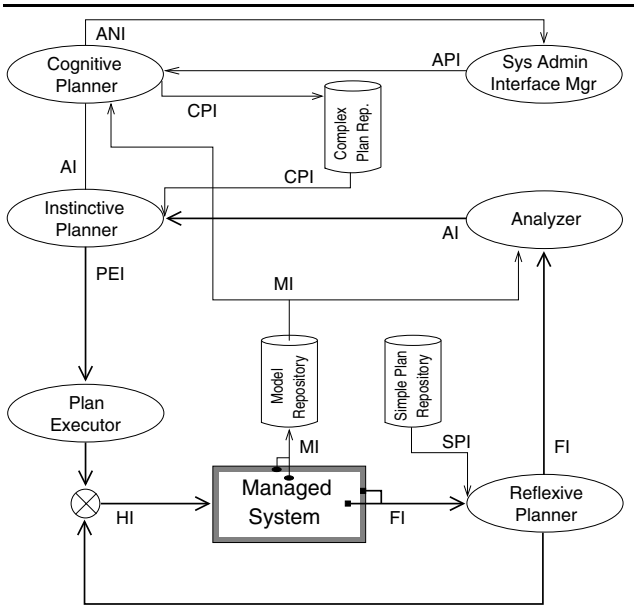
Failure info sensor A sensor of this type is passive and is activated when the application experiences a failure. It retrieves information about the failure context (such as failure type, stack trace, and failure place).

Model info sensor A sensor of this type is active and retrieves information that is needed to keep the autonomic manager's model of the application alive and up-to-date. The application model is based on two abstractions: (1) abstraction of the static structure of the application, and (2) abstraction of traces of the states of the running application.

As a result of exploration of real application logs, both types of sensors are instrumented at the following positions in the application:

Application content Sensors are instrumented in all strategic components (objects) of the application. These components together implement the business logic and the functionality of the application.

Application boundary Sensors are instrumented at the boundary of the application where interactions with other applications (systems) take place. The following interaction points are identified: (1) user interaction: a point where a user can provide input to the application and expects output from the application, (2) operating system interaction: a point where an interaction with an OS service takes place, like an I/O interaction, (3) middleware services interaction: a point where a middleware service is used, like a naming-service, and (4) data source interaction: a point where an application receives its required data from other systems. For the time being, three data source interaction points are distinguished: database interaction is a point where a database connection is set up and queries are sent to the database; web services interaction is a point where the application interacts with other applications using web services; and legacy back-end interaction is a point where a legacy system is approached through some proprietary protocol.



FI = Failure Info, MI = Model Info, HI = Healer Info
SPI = Simple Plan Info, CPI = Complex Plan Info, AI = Analysis Info
API = Admin Policy Info, ANI = Admin Negotiation Info, PEI = Plan Execution Info

Figure 1. Conceptual self-healing architecture.

3. Architecture

Figure 1 shows our self-healing architecture. It contains two feedback loops starting from and ending at the managed system (see the thick arrow lines in Fig. 1). The first feedback loop deals with simple situations and performs very quickly. The second feedback loop deals with complex situations, which should be analyzed, and prepares complex healing plans. The following sections describe the managed system and the modules which compose the autonomic manger and their role in the architecture.

3.1. Managed System

Aspect-oriented programming (AOP) is a new programming technique that allows developers to clearly separate *cross-cutting* concerns (i.e., behavior that cuts across the typical modules). AOP introduces *aspects*, which encapsulate behaviors that affect multiple modules. Aspects can be inserted, changed and removed easily. We consider failure info and model info sensors as aspects and use low-level

AOP techniques to instrument these aspects in the application compiled code (class file and no Java source file—code which is ready to be executed). The result of the instrumentation is the managed system.

We are especially interested in monitoring event transitions, like method call, state change, branch selection (model info) and exception occurrence (failure info) at different application points.

The model information (MI) is captured by the Model Manager and stored remotely in the Model Repository. When some runtime failure at one of the instrumentation points occurs, the failure info sensor collects failure context information (FI) (such as component name, method name, method formal and actual parameters, stack trace and line number) and triggers the Reflexive Planner.

3.2. Reflexive Planner

Some of the most frequently occurring failures, like *ClassNotFound Exception*, have a clear and simple cause and there is no need to perform an analysis step. This type of failure can be healed very quickly using the Reflexive Planner which runs in the same process space as the managed system and plays a central role in the first feedback loop.

The Reflexive Planner is responsible for a pool of specialized reflexive planners which continuously listen to the incoming FIs and handle them utilizing simple plans (SPIs) hosted in Simple Plan Repository which is filled by system administrators.

If a reflexive planner succeeds, it generates healing info (HI) and sends HI to a Failure Healer and reports its action to the Sys Admin Interface Mgr. Otherwise it triggers remotely the Analyzer by routing FI.

3.3. Analyzer

For complex failures the second feedback loop is triggered to determine the root cause of these failures. The Analyzer is responsible for a pool of specialized analyzers which continuously listen to the incoming FIs. Analyzers combine model (MI) and failure (FI) information using static and dynamic program analysis techniques to determine the root cause of failures. As each failure is associated with a specific pre-defined exception type, analyzers are able to base their analysis on the semantics of the exception type.

As a result, analyzers generate analysis info (AI) and send it to the Instinctive Planner. The analysis info contains the following information: location of the root cause (in terms of file, method, statement, interaction point) and context of the root cause (method call-path and current state).

3.4. Instinctive Planner

Most of frequently occurring failures, like *NullPointerException*, all have different causes. The instinctive planner may have known plans to heal them. The Instinctive Planner is responsible for a pool of specialized instinctive planners which continuously listen to the incoming AIs and handle them using complex plans (CPIs) hosted in Complex Plan Repository. Complex plans are applied on known situations and specified by system administrators or Cognitive Planner.

If an instinctive planner succeeds, it generates a plan execution info (PEI) and sends PEI to the Plan Executor. Otherwise instinctive planner triggers the Cognitive Planner by routing AI.

3.5. Cognitive Planner

The Cognitive Planner is the brain of the autonomic manager and is activated when instinctive planners encounter unknown situations. It utilizes artificial intelligence and machine learning techniques to produce new plans (CPI), given the current situation (AI) and system administrators policy.

If the generated new plans need to be confirmed, it provides a list of healing suggestions and negotiates them with the system administrators. Finally the new generated rules are stored in Complex Plan Repository.

3.6. Plan Executor

The Plan Executor is responsible for a pool of specialized plan executors which continuously listen to the incoming PEIs. Each plan executor has specific knowledge to translate the incoming healing plan to the execution implementation details.

Application failures, similar to diseases, have a symptom and a root cause. A distinction is made between components containing the failure symptom (symptom component) and components containing the failure root cause (root cause component). Note that they may physically be the same component.

Plan executors perform two important tasks: (1) stabilizing the panic situation arisen from the failure occurrence, and (2) repairing the failure root cause.

To stabilize the situation, they construct a clone of the symptom component and replace the method body of the symptom method with a new body. The new method body contains all code that has not yet been executed due to the failure. The Failure Healers are then remotely instructed to dynamically load the cloned component, initialize it with the last state of the symptom component and call the symptom method (with the new body) again.

In parallel, the code of the root cause component is repaired if possible. After restarting the application, the repaired code is executed. This process, in fact, simulates the way a developer debugs his/her code. The failure point becomes a breakpoint and the execution continues after correcting the failure code.

4. Experiments

A prototype of the architecture presented has been implemented in Java. Although fault diagnosis and self-healing operations can be deployed at different levels (see also Section 5), in the prototype it is used at the component and code level. In the prototype presented in this section, fault diagnosis is at component and code level, and self-healing repair operations are at code level. To test the applicability of our approach, we have applied the prototype to a small Java application, with two classes `Foo1` and `Foo2` (Fig. 2(a)).

We have injected a wrong code line in one of the methods of the application (Fig. 2(a), class `Foo1`, line 3), which will cause a *NullPointerException* in another method (Fig. 2(a), class `Foo2`, line 4). Normally, *NullPointerException* causes the application to crash if it is not caught by the code. (This type of exception occurs frequently in practice). For simplicity, variable `fsb` in class `Foo1`, line 3, is initialized with `null`, but in practice the value is read from persistent storage (file, database, etc.). So, it is not an obvious programming fault, as the value is not known on beforehand.

Javassist [4] has been used to instrument Java try/catch block at every interesting method body in the class files (compiled code) of the application. By the execution of the instrumented application the *NullPointerException* is caught and the context information about the failure is reported to the autonomic manager (Fig. 2(b), lines 9–13).

Based on the information about the symptom of the failure, the autonomic manager analyzes the problem and determines its root cause (Fig. 2(a), class `Foo2`, line 4). Afterwards a clone of `Foo2` is constructed with a new body for the `bar2` method (Fig. 2(c), line 3–5), and finally the execution is resumed. In fact, we stabilize the panic situation arisen due to the failure.

In addition, the root cause in the original code is repaired properly (Fig. 2(d), line 3). The repair rule is determined by the event (diagnosed root cause) and the healing operation is derived from objectives given by the system administrator. The original class is then replaced with the repaired one, which will be executed the next time if the application is restarted.

```
1 public class Foo1 {
2   public void bar1() {
3     StringBuffer fsb = null;
4     Foo2 foo2 = new Foo2();
5     foo2.bar2(fsb);
6   }
7 }
```

```
1 public class Foo2 {
2   public void bar2(Stringbuffer asb) {
3     System.out.println("before crash");
4     asb.toString();
5     System.out.println("after crash");
6   }
7 }
```

(a)

```
1 public class Foo2 {
2   public void bar2(Stringbuffer asb) {
3     try {
4       System.out.println("before crash");
5       asb.toString();
6       System.out.println("after crash");
7     }
8     catch (Throwable t) {
9       FailureInfo fi = new FailureInfo();
10      fi.setExceptionClass(t);
11      fi.setActualMethodArgs(asb);
12      fi.setFormalMethodArgs(...);
13      FailureReporter.report(fi);
14      FailureHealer.heal();
15      return;
16    }
17  }
18 }
```

(b)

```
1 public class Foo2 {
2   public void bar2(Stringbuffer asb) {
3     asb = new StringBuffer();
4     asb.toString();
5     System.out.println("after crash");
6   }
7 }
```

(c)

```
1 public class Foo1 {
2   public void bar1() {
3     StringBuffer fsb = new StringBuffer();
4     Foo2 foo2 = new Foo2();
5     foo2.bar2(fsb);
6   }
7 }
```

(d)

Figure 2. Example of code instrumentation.

5. Related and Future Work

Our proposed architecture resembles the ABLE Autonomic Agent [1] architecture. The main differences between the two architectures is that in our approach (1) the role of system administrator and analyzer are explicitly modeled, and (2) the cognitive planner and the instinctive planner are related to each other.

TRAP [10] uses AOP and behavioral reflection to provide runtime adaptation to the existing object-oriented applications without modifying their source code. In our approach AOP is used to extract failure and model information from the managed system.

Design by contract (DBC) [9] is a method for developing software in which software modules guarantee certain properties when they request services from each other. DBC is mainly used to warn developers about contract violations. In future work these violations will be used as historical data to help predict the behavior of the managed system and to make healing plans.

Bowring *et al.* [2] show how to use machine learning to automatically classify and predict software behavior based on execution data. They instrument programs to profile event transitions (like method calls, branches) based on test plans and represent the execution profiles as Markov models. From the Markov models (training set), they train classifiers to predict the behavior of unseen program executions.

Brun and Ernst [3] use dynamic invariant detection to generate program properties. They use support vector machine and decision tree learning tools to classify the properties and find fault-revealing properties. The training set is a set of programs with known errors and corrected versions of those programs.

Tschudin and Yamamoto [11] present an approach to create robust code for communication software which continues to operate despite parts of the implementation being knocked out. In their metabolic approach, implementation elements are able to continue to operate and can recover by themselves for restoring full services again.

Our cognitive planner can benefit from these approaches to derive more precise healing actions. Self-healing at code level is very complex, as the system has to derive the intended behavior of the application. If that cannot be determined, interaction with the system administrator is required. For self-healing at component, service, host, or location (host aggregation) level, at which functionality and operational semantics are defined, the automated healing process can potentially be applied to a larger extent.

We are currently working on an ontological framework based on OWL and OWL-S to represent the knowledge regarding managed system's healthy situation. The framework will provide mechanisms to system administrators in order to express the managed system's high-level healthy objectives, and actions to be taken when the health-conditions are violated. The high-level objectives need to be coupled with low-level observations and operations, i.e., the high-level healthy objectives need to be asserted by low-level observations (such as root cause diagnosis) from hosts, services, components, or application code. The self-managing tasks have a level attribute determining the level for the healing operation (OWL-S grounding details on the interoperation between the levels).

Other research currently on the agenda are: (1) to use program analysis to determine the root cause of problems and to study the side effects of system stabilization and system healing actions, (2) to use practical explorations to determine the domain of reflexive, instinctive and cognitive planners, (3) to use machine learning to predict the root cause of multi-causal failures and to generate healing suggestions, and (4) to use agents to implement the modules of the autonomic manager.

Acknowledgements

This research is supported by the NLnet Foundation, http://www.nlnet.nl and Fortis Bank Netherlands, http://www.fortisbank.nl.

References

[1] J. Bigus, D. Schlosnagle, J. Pilgrim, W. Mills, and Diao. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371, 2002.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 195–205, Boston, MA, July 2004.

[3] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 480–490, Edinburgh, Scotland, May 2004.

[4] S. Chiba. Javassist: Java bytecode engineering made simple. *Java Developer's Journal*, 9(1), 2004.

[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, Oct. 2001.

[6] M. Ernst, J. Cockrell, and W. G. D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb. 2001.

[7] B. Finkbeiner, S. Sankaranarayanan, and H. Sipma. Collecting statistics over runtime executions. *Electronic Notes in Theoretical Computer Science*, 70(4):1–19, Dec. 2002.

[8] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[9] G. T. Leavens and Y. Cheon. Design by contract with JML. http://jmlspecs.org, 2004. In draft paper.

[10] S. M. Sadjadi, P. McKinley, R. Stirewalt, and B. Cheng. TRAP: Transparent reflective aspect programming. Technical Report MSU-CSE-03-31, Department of Computer Science, Michigan State University, East Lansing, MI, Nov. 2003.

[11] C. Tschudin and L. Yamamoto. A metabolic approach to protocol resilience. In *Proceedings of the First Workshop on Autonomic Communication (WAC 2004)*, Lecture Notes in Computer Science. Springer, Berlin, Germany, Oct. 2004.