Towards Autonomic Computing: Injecting
Self-Organizing and Self-Healing
Properties into Java Programs

M.J. Oudshoorn, M.M. Fuad, D. Deb
Department of Computer Science
Montana State University
Bozeman
MT 59717-3880
USA

Abstract. Autonomic computing is a grand challenge in computing. It aims to produce distributed software that has the properties of self-healing, self-organizing, self-protecting and self-optimizing. This paper focuses on the challenge of injecting only self-healing properties into existing programs that targets the Java Virtual Machine. The byte code is analyzed and additional code is injected to automatically recover from failure such as network or processor failure. A framework to provide autonomic computing support is presented and evaluated to determine its suitability for a fully fledged autonomic computing system.

Keywords: Autonomic computing, software engineering, distributed systems.

1. Introduction

As software systems increase in size and complexity as larger and larger problems are tackled, programmers turn to technology such as parallel and distributed systems to help them achieve reasonable throughput and deliver results in a reasonable timeframe. As this occurs, programmers focus less and less on the problem at hand and more on more on the complexity associated with the technology they are trying to embrace. Unfortunately many programmers lack the skill of the interest to develop such distributed systems. Equally as bad, many forget to pay attention to issues such as code migration and load balancing which is essential to achieve their goal of improved throughput on multi-user systems.

Ideally, programmers would like to focus on the application on hand and less on the underlying complexity of the technology they are employing. They care about the quality and correctness of the results and the total execution time, but they do not want the responsibility of managing the entire distribution and execution process. This was described as the “All Care and No Responsibility” principle of software development in [18, 19]. The basic concept was termed autonomic computing by IBM’s senior vice president of research, Paul Horn, in a March 2001 keynote address to the National Academy of Engineers [12, 15].

The principle notion is that computing systems should provide support to programmers by being self-managing. A set of policies guides the behavior of the system
as it executes. This is achieved through the application having the following properties where the precise behavior is determined by the policies associated with the application:

- **self-organizing**: The application automatically configures itself across the available resources without the need for additional programmer intervention. It takes into account the resources required by the system and the resources available to determine the best allocation of processes across the available resources.

- **self-protecting**: The system defends itself against malicious attacks or cascading failure. The system tries to anticipate and prevent system-wide errors that are likely to have a detrimental effect on the accuracy of the result or the performance of the application.

- **self-optimizing**: The components constituting the system are constantly seeking ways of improving their performance to deliver better quality results and improve throughput.

- **self-healing**: The system monitors its own health and automatically diagnoses and repairs failures (software and hardware).

This support should be provided without the need to programmer intervention or consideration other than the generation of policies to guide the execution process. The provision of all of these features is an unresolved problem [23] – there are no truly autonomic computing systems in existence at this time. This paper examines an architecture that attempts to deliver support for autonomic computing through the provision of self-organizing and self-healing properties. We refer to this system as ACE (Autonomic Computing Environment). This paper focuses on Java byte code as the primary code source. This permits the provision of autonomic computing support for legacy systems for which the source code is not available, as well as current systems for which the code is available. It permits support for Java and for any other language that currently exists (or will in the future) which targets the Java virtual machine. The goals of the system proposed in this paper include:

- **Autonomic environment**: The underlying system should be autonomic, i.e. it should have elements to self-configure, self-heal, self-optimize and self-protect. We believe that, only providing an autonomic environment to the application programmer is not enough as long as building programs in such system is made simpler. If programming in such system requires advanced knowledge of the workings of the underlying autonomic environment, then the goals of autonomic computing is not fulfilled. Users should program as usual with minimal constraints and the underlying system should transform the code in a way that could be self-managed autonomically.

- **Automatic distribution**: An average programmer wishing to utilize idle machines in the network does not want to take the responsibility for the physical distribution and coordination of objects. They are concerned more with the throughput and total execution time of the whole application. Therefore, programmers should need to do little (or nothing) to ensure the distribution of objects across the distributed environment. The system will automatically determine object dependability and allocate resources accordingly and coordinate the objects transparently at runtime. The programmer can influence the system’s automatic decision by providing user specific policies. Whether the user do that or not, the system should hide the rest of the distribution and coordina-
tion process from the programmer. Programmers should have the impression that the objects are all executing on a single machine, although in reality they may be executing somewhere else in the network.

- **Autonomize at the byte code level:** The goal is to inject autonomic support into non-autonomous system, whose source code is no longer available. Therefore the byte code must be analyzed and the autonomic functionality should be inserted in such a manner that it is separated from the service functionality of the legacy system.

- **Ease of use:** The system will require minimal interaction from the programmer to achieve improved productivity and self-management. Problems encountered due to distribution should be hidden from the user and dealt with automatically in the background. In doing so, the system should not jeopardize the user application or the integrity of the results. The system will provide an easy to operate user interface so that naive computer users could easily interact with the system and run their applications. On average, it should reduce the work and complexity associated with managing a large system. The system will be able to better respond to sudden changes in the environment and adjust its own settings appropriately.

Section 2 examines previous work in autonomic systems. Section 3 of this paper presents the general architecture that is being developed. Section 4 examines support for self-organization and Section 5 looks at self-healing properties. Finally, Section 6 evaluates the work that has been presented and Section 7 concludes the paper.

### 2. Existing Work

Most of the autonomic systems to date are actually prototypes or provide a limited amount of required functionality [1, 8, 20] of an autonomic system. The most important aspect of the new system is that it introduces new metaphors or provides a completely new approach to autonomic computing that adds additional complexity and a steep learning curve for the programmer. The goal of this work is to make the resultant system simple to use, by making the underlying autonomic framework transparent. None of the following systems match this goal.

The Unity system [1] is a platform designed to help autonomic elements interact with each other and their environment. It uses goal-driven self-assembly to configure itself. However, the utility function it uses for self-assembly assumes that one can quantify the utility of different choices. The Unity system does not address the question about how complex it is for application programmers to use this prototype. There is no discussion of programming in such an autonomic system. Along with providing a runtime environment for autonomic elements, our goal is to provide programmer simple to use interfaces to program in such a system.

Autonomia [8] is a ‘proof-of-concept’ prototype software development environment that provides application developers with tools for specifying and implementing autonomic requirements in a distributed application. The goal of Autonomia is to automate deployment of mobile agents that have self-manageable attributes. Autonomia only address self-configuration and self-healing properties of autonomic systems. Users of Autonomia have to follow a well-defined library and a predefined Application Service Template to create their programs. So users are exposed to the underlying system and need to know specific interfaces explicitly to program.
AutoMate [20] develops an autonomic composition engine to calculate a composition plan of components based on dynamically defined objectives and constraints that describe how a given high-level task can be achieved by using available basic Grid services. AutoMate provides a set of tools for programming framework, autonomic composition, coordination middleware etc. However, AutoMate does not address the complexity of integrating autonomic functionality into applications. It introduces many new metaphors and paradigms that learning to program such an autonomic system is complex and does not deliver the benefits sought through autonomic computing.

There are other systems that attempt to provide support for legacy systems. Since the work reported in this paper focuses on the Java byte code it is capable of adding autonomic support for legacy Java systems also. Relatively little research has been done to integrate autonomic functionalities into legacy code and, typically, the options explored are not transparent to the user and, in fact, require the user to have an extended knowledge of the legacy code.

Haydarlou et. al. [11] presents an approach and a conceptual architecture for fault diagnosis and self healing of interpreted object oriented application. Their approach is to equip current and legacy interpreted object oriented code with their proposed technique so that the application can self heal itself and could also try to solve the root cause that initiated that fault. Although we share the similar goals and approach as them, our approach is extensive and completely transparent to the user. The authors do not provide details of their code injection methodologies and do not address different issues that arise during such code injection. The authors propose a conceptual architecture for fault diagnosis; however, they do not elaborate how the learning algorithm actually learns new fault scenarios. The authors experimented with their proposed technique with a very basic application, for which there is no need to save any state information for restarting it after a fault. With the technique provided, their approach will not work for complicated programs, having nested try-catch block, local variable interactions, conditional branches, etc. The authors also do not provide any code inflation or execution time information for their proposed approach.

The work by Schanne et. al. [24] attempts to inject autonomic functionality to existing object oriented code. The authors used the standard proxy/wrapper architecture to inject the added functionality, namely self-organization and self-optimization. The authors used static reflection to gain the structure of a class and then change the methods according to their needs. One of the major drawbacks of their approach is that user needs to supply the pre-processor some meta-information about the code, such as, pre and post-conditions and invariants of methods. So the assumption is that the user knows about the code and could express such meta-information about the methods. This assumption is not realistic for legacy code, as exploring the meta-information for such code is nearly impossible. Users cannot get this required meta-information by just examining the byte code, they need the source code for that, which in real life, may not be available for a legacy code. Also, as they rewrite only a portion of the byte code of the original class (either through mutators or directly), all calls to the original class should be synchronized to maintain the consistency between the proxy and the original class.

Abbas et.al [1] builds an infrastructure that offers adaptation, evolution and autonomic management support to existing systems. Their technique is based on dynamically linked libraries and therefore restricts itself to GNU C library. Their approach autonimize the existing code for run time diagnosis purpose by inserting dynamic linker hooks in the existing code, so that at run time, corresponding libraries are loaded
for diagnosis purpose. Since their approach is tied to C, it cannot be used in a platform agnostic environment.

Kinesthetic eXtreme [14] provides a completely new approach for making legacy system autonomic. They use DASADA standards [2] for probes and gauge technique to monitor and adapt an existing system. The existing code is retrofitted with probes and all probes in the system sends runtime information to a centralized array of gauges where overall system status is determined from that information and any feedback to the system is relayed back to the corresponding components through the embedding probes. According to the vision of autonomic system [12], each of the subcomponents in the system should self-manage and therefore as a whole the system would become self-managed also. However, this approach collects status information from the whole system at first and then tries to change the behavior of the whole system. Since probes from all the sub-component are feeding data to the gauges constantly, there would be significant communication traffic. The system provides self-healing capabilities in the whole system level and not in the single component level.

Orso et. al. [17] presents a technique to dynamically update a running Java application. They use proxy classes to rewrite their code and allow substituting, adding and deleting of classes at run time. Their technique operates by first statically modifying the application code (by class renaming and code rewriting) to enable its dynamic updating and then perform the hot swapping of classes at run-time, when a new version of a class is available. Although this technique provides a novel way of reconfiguring an existing distributed system, it does not follow the autonomic paradigm and it needs direct user intervention to do the hot-swapping.

The work by Griffith et. al. [9] presents a technique to add self healing capabilities to the Common Language Runtime, explicitly for the .NET platform. They provide a framework that allows a repair engine to dynamically attach and detach to/from a managed application for self-healing purposes. Since .NET platform provide more control to user on accessing runtime program and environment parameters, this technique will not work within a Java virtual machine because of its restricted access to runtime environment.

Our approach differs in the way that we want to relieve the application programmers from the burden of such complex and ever changing metaphors. We believe that, programming an autonomic system should be made easy and transparent to the user, otherwise the goal of autonomic computing is sacrificed.

3. General Architecture

It is recognized that any programming framework for building an autonomic system should have the following minimum requirements:

- **Task**: A task in the system is the minimum entity of execution, which may comprise of one or more inter-dependent objects. In this paper, task and partition are used interchangeably.

- **Task management**: Creation of task should be safe, uniform and dynamic. Once created, the system should maintain the entity during its lifetime.

- **Communication**: The underlying communication mechanism should conform to uniform syntax and semantics and should follow any open source protocol to fulfill the basic requirements set by autonomic computing principles [12,
15]. It should provide secure access control and should hold the integrity of the messages in case of any intrusion.

- **Naming**: The naming scheme of task and nodes should be implementation independent and platform agnostic. It should be dynamically generated and should reflect some property of the entity that it is representing.

This research takes a two-pronged approach to accomplishing the stated objectives. First, a static code analyzer builds an object graph from the user supplied byte code. Once it generates the object graph, it partitions the graph according to the underlying system configuration, communication requirements or any user supplied policy. The underlying system comprises a platform-agnostic language and the associated pre-processor for comprehensive byte-code to byte-code translation, so that the resultant transformation produces an autonomic version of the user code and provides support for self-organization (at least in the initial distribution). The transformed program will be based on self-contained concurrent objects communicating through any standard communication protocol and will incorporate salient features from existing middleware technologies. The pre-processor takes a platform-independent application implementation and produces a customized distributed code. Then it communicates with the underlying system for various runtime parameters and generates a deployment scheme detailing the aspects such as object placement, target node, communication protocol etc.

In conjunction with the development of the pre-processor which transforms the byte code, an autonomic framework is built that performs automatic application partitioning and placement based on site-specific application placement policies, capabilities, and current system load. The resource management service in the pre-processor is responsible for placement decision-making: it generates initial deployment schemes for an application; based on the information collected by the system monitoring service. Once distributed, the underlying autonomic system gains the control of the objects and manage the user program afterwards. As a result the underlying framework is adaptive, because it adapts the user application to various platforms and protocols on the network and to unpredictable runtime conditions. Figure 1 shows the system’s flow of operation.

### 3.1. Policies

The important aspect of any autonomic system is for the user to specify the behavior of the system in high level and broadly scoped directives. This will be accomplished by the use of policies. A policy is a representation of desired behaviors or constraints on behaviors [28]. The system will allow two forms of policies to cover most of the possible contexts. The first form of policy would be if-then-else policies. These policies will have nearly no abstraction and will describe actions depending on different situations. The second form of policies will be more abstract and will be composed of several if-then-else policies. Adaptation of the Autonomic Computing Policy Language (ACPL) [2] is desirable as it provides a user friendly form of policy definition, policy management and different tools and API to work with policies. The user in the system will be given interfaces and templates to define their own policies and all policies will be deposited in a hierarchical policy depositor. The static analyzer and the partitioner will provide basic policies, comprise of resource allocation, computation and communication directives derived from the analyzed code. Users can add or modify those system generated policies and could supervise the deployment.
3.2. Behavior of autonomic elements

Autonomic elements are the heart of the system. They encapsulate the program partition (a portion of the user program) as its managed element and interact with the environment by using standard autonomic metaphors. Some of the autonomic elements in the system will be given some higher level administrative authority. These managerial autonomic elements will either manage system registry and policy depository or will act as the user interface for program partitioning and transformation, monitoring or the source or destination of program input and output. However, all the autonomic elements in the system have the same properties and they could act in any of the above roles if they are instructed to. To house those extra services, we extended the notion of autonomic elements with an autonomic manager and several control interfaces. As shown in Figure 2, the autonomic manager assumes the role (either a standard or managerial behavior) that is being determined by the environment.

There are multiple interfaces for the different services to be described, discovered and supplied. For instance, the service interface gives other autonomic elements to reach to an agreement related to delivering different services to each other. The policy interface will provide a way to transfer and modify policies between different autonomic elements. Monitoring interface will provide methods to monitor its internal activities and status information. The deployment interface will provide methods through...
which managed elements will be allocated, deployed or restarted. Separating the functional aspects of autonomic management from the management of autonomic element makes the overall software architecture more modular and easy to extend in the future.

Autonomic elements need to be in a mutual service relationship to interact with each other. If they do not have any service relationship, they could only access certain methods of each other’s control interfaces. This is for security purposes, so that the most delicate and secure operations between autonomic elements could be initiated by each other and not by any third party.

4. Self-Organization

Analyzing and representing software in terms of its internal dependencies is important in order to provide the self-managing capabilities because this is actually the system’s view of the run time structure of a program. In our approach, the underlying system controls the partitioning and analyzes the dependencies and therefore is informative about everything. Once partitioned, Autonomic Elements (AEs) are responsible for taking any optimization and migration decisions for the managed element and a program’s total view in terms of a graph is helpful to make such initiatives. For instance, at runtime, due to load or other factors an AE may decide to migrate its managed element to a less loaded machine and there may be several machines available to handle that load. As each AE has the complete structure of the object dependency graph and the information about which partitions are managed by which AEs (obtained via monitoring services and interaction among AEs), the optimal replacement AE can be found which is in closest proximity with other AEs managing other partitions that have active communication with the partition to be migrated. Therefore, a dependency graph is not only important for initial object placement, but also affects the runtime decisions, made by the system, towards system wide maximized resource utilization policy or any other user policy.

To construct such a graph, it is necessary to determine two pieces of information, namely: 1) the resources (i.e. computation time, memory, disk space, network etc.) consumed by each object and 2) the dependencies (directionality and weight) among the objects which is caused by the interactions among them. Therefore it is necessary to construct a weighted directed graph \( G = (V, E) \), where each node \( v \in V \) represents an object and the edge \( (u, v) \in E \) resembles the communication between objects \( u \) and \( v \). The computational weight \( w(v) \) represents the amount of computation takes place at object \( v \) and the communication weight \( w(u, v) \) captures the amount of communication between objects \( u \) and \( v \). Constructing such a dependency graph statically is a challenging task. There are several Java automatic partitioning tools [26, 27], however, they only detect interaction at the class level, therefore performs partitioning at class granularity and limits the opportunity to exploit object level concurrency. To our knowledge, Spiegel’s Pangaea [25] is the only system that performs analysis at the object level.
Spiegel’s algorithm statically analyzes the Java source code and produces a directed graph where each node and edge represents run time instances and relations (create, use and reference) among them respectively. We use a modified and extended version of this algorithm in our system to deduce object dependency graph.

While Spiegel’s algorithm provides important insight about object dependency graph construction, it is not sufficient for our purpose. For instance, the original algorithm simply produces a directed graph. In contrast, we are interested in a weighted directed graph to effectively extract the computation and communication requirements of objects as well as the relations among them. Moreover, instead of having a general use relation, our target is to further categorize it as read-only and write based on whether the data members of an object are simply accessed or modified during use.

There are also significant differences in algorithmic aspects and implementation strategies. The original implementation assumes the presence of source code, while we are performing the analysis at the byte code level. One additional advantage of our approach is that it allows us to access system classes that are accessible at the byte code level. Spiegel’s algorithm finds the set of types, objects, methods etc. by examining the code at the syntactic level. In contrast, our study uses standard compiler analysis and an efficient intermediate representation (IR). Another study [7] also implemented a modified version of Spiegel’s algorithm at the byte code level. However, like [25], they also deduced a dependency graph without computation and communication significances and used a different IR. Determining the type of a run time object is critical in Java due to polymorphism, inheritance and dynamic binding. Spiegel’s algorithm did not use any standard type inference mechanism to resolve dynamic dispatch and therefore the set of types each reference variable may point to at runtime includes all subtypes. Consequently the resultant object graph has unnecessary edges. In contrast, we use standard techniques like call graph and points-to analysis to resolve dynamic dispatch. Consequently, the analysis presented in this paper produces a more compact graph which is less expensive to perform further analysis on and in that way larger applications can be handled. In the following sections, we first briefly describe the original algorithm and then we discuss the issues related to our implementation. Readers are advised to read [25] for an extensive description of the original algorithm.

4.1. Original algorithm

Spiegel defines set of Java types as classes (non primitive), interfaces and arrays used in the program. He further splits up a Java type as a static type (comprising the static fields and methods of the class), and a dynamic type (the non-static members). For a static type, precisely one static object exists at run-time, while for a dynamic type, the number of instances depends on the enclosing program statements where the allocation takes place. An allocation may appear inside a control structure or loop and as a consequence the number of instances may not be known exactly. In such cases, an indefinite object that summarizes all instances together is inserted into the object graph, otherwise concrete object nodes are used.

The original algorithm works as follows: at first, the set of types the program consists of is computed by syntactically identifying the type closure of the program. Then the class relation graph (CRG) is constructed which captures relationships at the type level. Whenever, a statement belongs to the context of type \( A \) calls a method or accesses a field of type \( B \), an usage edge \((A, B)\) is added to the CRG. Data flow information such as export or import relations take place when new types propagate from one
type to another through field access or method calls. An export edge \((A, B, C)\) is added between type \(A\) and \(B\) when type \(A\) owns a reference of type \(C\) and passes it to type \(B\). Similarly an import edge \((A, B, C)\) is added when type \(A\) owns a reference of type \(B\), from which it receives a reference of type \(C\).

The algorithm then computes the object graph (OG) consisting of run time class instances along with create and reference relations among them. For each allocation (new) statement, create and reference edges are added between the class instances where the allocation takes place and the newly created instance. After the object population has been computed, the algorithm then iterates over all object triplets and use the data flow information from CRG to propagate references within OG until a fixed point is reached. Finally, the algorithm adds a usage edge between objects in OG if they already have a reference relation in OG and their corresponding types have usage relation in CRG.

4.2. ACE implementation

The ACE static analysis of Java byte code is built on top of the Jimple [22] representation, which is part of the Soot framework. The Soot framework is a set of Java APIs for manipulating and optimizing Java byte code. We analyze complete application, therefore by using Soot we first read all class files that are required by the application starting with the main method and recursively loading all class files used in each newly loaded class. As each class is read, it is converted into Jimple IR, suitable for our analysis and transformations. Jimple is a typed, stackless and compact three-address code representation of byte code. Jimple only involves 19 kinds of instruction and as a result is much easier to manipulate compared to stack oriented byte code representation that involves 201 different instructions. Figure 3 shows the Java source code of the example program used to generate the byte code discussed below. In this program, course \(c2\) is instantiated only if the user supplied boolean variable \(math\) is true i.e. course \(Math100\) is compulsory for each student. Class \(St\_Student\) is not part of the original application and is produced by the ACE system during a preprocessing stage as a result of separating static and dynamic types. In this case all the static members of class \(St\_Student\) were actually part of class \(Student\) in the original program.

Soot also provides the necessary support for generating a call graph and performing points-to analysis based on Jimple code. In Java, all instance methods are invoked using virtual call. The actual method invoked depends on the run-time type of the object receiving the method call and often termed a receiver object. The call graph approximates the set of target methods that could possibly be invoked at each call site (method invocation) in the program. On the other hand, points-to analysis makes the call graph more compact and precise by limiting the number of target methods invoked in a call site. Soot’s SPARK [16] points-to analysis engine is used to computes the set of run-time types pointed-to by each program variable. Using the set of receiver objects at each call site, for each type, the methods that will be invoked are actually identified.

We observe that, once we obtain the call graph, it captures all program type interactions if we consider that the only way object \(a\) of type \(A\) can access an object \(b\) of type \(B\) is by invoking a method defined in type \(B\). Identifying interactions that occurs whenever an object \(a\) directly accesses a data field of \(b\) requires additional passes through the Jimple code and manipulation of some other data structures beside the call graph which is time consuming and introduces additional complexity. Therefore, during preprocessing, ACE generates additional accessor and mutator methods at byte
code level that allow the load and store accesses to data fields via getXXX() and setXXX() methods and replace all direct field accesses in the original program by respective method call statements. Thus we perform our analyses at the level of method granularity that is another major distinction from Spiegel's approach.

One important additional advantage of this transformation is that, not only methods, but also fields can be then accessed remotely by a RPC style mechanism as implemented in JavaParty [21]. However, this approach cannot be applied to system classes as they are not modifiable.

4.2.1. Method database.

An important structure we utilize in our implementation is the Method Database (MD) that records information about each method and helps us determining the computational and communicational weights. The MD is constructed by inspecting the Jimple representation of each method used in the program. Each such method is analyzed to obtain information such as 1) whether it involves reading/writing of the fields of the class containing the method 2) amount of resource (CPU, memory) consumed by the method and 3) amount of communication needed to invoke the method. To avoid ambiguity due to polymorphism and method overloading, each method in the MD is represented by a unique method signature as follows:

$className: returnType methodName(paramType1, paramType2......paramTypen)>

```java
public class Example {
    public static void main(String[] argvs) {
        Student s1 = new Student("James", 76);
        Student s2 = new Student("Jill", 57);
        Course c1 = new Course("CS150");
        c1.addStudent(s1);
        System.out.println(c1.getName()+".* + c1.findGpa());
        if(math) {
            Course c2 = new Course("Math100");
            for(int i = 0; i<St_Student.getAllStudent().size(); i++) {
                c2.addStudent((Student)St_Student.getAllStudent().elementAt(i));
                System.out.println(c2.getName()+".* + c2.findGpa());
            }
        }
    }

    public class Course {
        private String courseName;
        Vector registeredStudent;
        public Course(String n) {
            courseName = n;
            registeredStudent = new Vector();
        }
        public String getName() {return courseName; }
        void addStudent(Student s){ registeredStudent.add(s); }
        public Student findGpa(){
            Student hGpaHolder; ...; return hGpaHolder;
        }
    }

    public class Student {
        private String studentName;
        private double gpa;
        public Student(String name, double gpa) {
            studentName = name;
            this.gpa = gpa;
            St_Student.addToAllStudent(this);
        }
        public String toString() {
            ...
        }
    }

    public class St_Student {
        static Vector allStudent = new Vector();
        static void addToAllStudent(Student s)\{allStudent.add(s); \}
        static Vector getAllStudent() { return allStudent; }
    }
}
```

Figure 3. Example Java source code.
Each method is categorized as read/write by recursively examining the load/store field accesses in method’s context. Approximating the computation time for each method is difficult mainly because of the presence of conditional statements and dependability on both input parameters and hardware platforms. In this study, the computational weight of each method is estimated in terms of number of Jimple instructions needs to be executed (ins_cost). Figure 4 shows a Java method and its Jimple representation, where categorization of each Jimple statement is also included. Each of 19 different kinds of Jimple instruction is then weighted according to the associated cost and the final estimate for a method is generated based on these weights. In the case of control structures, the system assumes that each branch of an if-else is taken 50% of the time and loops are executed for a configurable number of times [3]. The architecture-specific execution costs associated with ins_cost can then be computed by arch_cost, provided in units of msec per Jimple instruction. To obtain the arch_cost, the Jimple code is benchmarked on each type of resource. The computation time obtained in this way is roughly approximated and parameters can be tuned over time to provide a better approximation. The memory requirement of each method is estimated by the memory consumed by data members of the respective class, method parameters and local variables.

4.2.2. Finding the set of types.

To find the set of types ACE first builds the call graph of the application using the Soot framework. The set of types of an application is exactly the set of types that appears as a receiver of the methods in the call graph.

4.2.3. Construction of class relation graph (CRG).

The CRG is a directed graph \( G_c = (V_c, E_c) \) where \( V_c \) is the set of types that we computed in the previous step and \( E_c \) is the set of usage, export or import edges among types. The CRG construction algorithm explores every node (method) in the call graph and works as follows:

For each method invocation:

```java
public void test(int x)
{
    double sum;
    if(x<10)
    {
        String s = "Less than ten";
        s.toString();
    }
    else sum = Math.sqrt(x);
}
public void test(int) {
    Test r0;
    int i0;
    java.lang.String r1;
    double $d1;  // identityStmt
    r0 := @this: Test; // identityStmt
    i0 := @parameter0: int; // identityStmt
    if i0 >= 10 goto label0; // ifStmt
    r1 = "Less than ten"; // assignStmt
    virtualinvoke r1.<java.lang.String: java.lang.String toString>(); // virtual method invocation
    goto label1;  // gotoStmt
    label0: $d1 = (double) i0; // assignStmt
    staticinvoke <java.lang.Math: double sqrt(double)>($d1); // static method invocation
    label1: return; // returnStmt
}
```

Figure 4. Java method and Jimple code.
1. Let $A$ be the class containing the method call statement, $B$ be the receiver class of the method, $P$ be the list of types of method parameters and $R$ be the return type.

2. If $A \neq B$, then add the following edges between type $A$ and $B$,
   a. An usage edge $(A, B)$,
   b. An export edge $(A, B, P[])$,
   c. An import edge $(A, B, R)$.

   Figure 5 shows the CRG deduced from the example program. For better visualization, we summarize all usage, export and import edges between type $A$ and $B$ into a single edge as $(A, B, E[], I[])$, where list $E[]$ contains a set of types exported from type $A$ to $B$ and list $I[]$ contains a set of types imported from type $B$ to $A$. It should be noted that, the Java runtime system classes that are also invoked by the references to classes System.out and String are omitted in the final CRG.

   Each edge $(A, B)$ in CRG also contain a list of methods through which type $A$ and $B$ interacts. With this list and information contained about each method in MD, the edge $(A, B)$ in the CRG is then further categorized as read-only/write. For a method $M$ appears in directed edge $(A, B)$ in CRG, the estimated computation weight and memory usage of $M$ from MD are added as $w(B)$ (computation weight of node $B$). The communication weight $w(A, B)$ depends both on how many times $A$’s enclosing statements invokes a method of $B$ and number of bytes required to represent the method parameter and return type. Final values of $w(B)$ and $w(A, B)$ are calculated by adding the weights caused by all methods appearing in edge $(A, B)$. For example edge $(Example, Course)$ in Figure 5 involves four methods such as
   
   $\text{Course: void } <\text{init}> (\text{java.lang.String})$
   $\text{Course: void addStudent(Student)}$
   $\text{Course: java.lang.String getCourseName()}$
   $\text{Course: Student findhGpa()}$

   Some of these methods modify data members of type $Course$, so the edge $(Example, Course)$ is categorized as write. The weight $w(v)$, where $v$ is an object of type $Course$, is then the summation of computation performed by the above methods.

4.2.4. Construction of initial object graph (OG).

OG is a weighted directed graph $G = (V, E)$ where $V$ is the set of run-time instances and can be of type static, concrete or indefinite (see Section 4.1) and $E$ is the set of edges among the nodes in $V$ and can be of type read-only, write, create or reference. To obtain the set of objects, we first identify allocations in the program. In Jimple, this includes statements that allocate objects and arrays and that load string constants. Some example of the allocation statements in Jimple are,

$$p = \text{new Student}, q = \text{newarray(int)[12]}, r = \text{"Hello"}$$
Identifying the allocation statements is performed at the same time when method database is constructed as both requires examining each Jimple method. Each method in MD keeps track of all allocations inside it including the class of the allocated object and the type of allocation (concrete/indefinite). Static edges are also identified in CRG, where the receiver object of the method call is of type static. For instance, edges (Example, St_Student) and (Student, St_Student) in Figure 5 are static edges. In Jimple, static invocation is always preceded by staticinvoke as shown in Figure 4.

The OG construction algorithm utilizes these entries and works as follows:

1. For each static type, add one static object in OG.
2. Repeat the following for all static objects in OG
   2.1. For a static object of type A, extract all the allocations from all the methods in the method database where the className part of the method signatures matches A.
      2.1.1. Add one concrete object of respective type for each concrete allocation in A.
      2.1.2. Add one indefinite object of respective type for all allocations of that particular type inside a control dependent statement in A.
   2.2. For each concrete object of type A added in the graph, add concrete/indefinite objects in the same way as in step 2.1.1
   2.3. For each indefinite object of type A added in the graph, all allocation statements in A are treated as uncertain and therefore an indefinite object is added to the graph for all such allocations in A.
3. For each concrete or indefinite object added to the graph, add a create and a reference edge from the parent to the newly created object.
4. For each static edge (A, B), add a reference edge from all objects of type A to static object B.

Figure 6 shows the object graph generated from our example program.
4.2.5. Propagating usage and reference edges.

The algorithm now iterates over all triples of objects \((a, b, c)\) in OG for which reference edges \((a, b)\) and \((a, c)\) or \((b, c)\) exist in OG and matches the types of the objects against the data flow edges \((A, B, C)\) in CRG, then a new reference edge \((b, c)\) or \((a, c)\) is added to the object graph as explained in Figure 7. Finally a use edge \((a, b)\) is added to the OG if \((A, B)\) exists in CRG and \((a, b)\), exists in the OG. For each added usage edge, the corresponding types of usage and computational and communicational weights are also propagated from CRG to OG. For clarity, the OG in Figure 6 only shows read/write relations between objects while omitting the creation and reference relations.

4.3. Applicability of object graph

Based on the object graph, it is possible to categorize objects as 1) immutable (has no incoming write edge i.e. never changes after been created), 2) Single Read/Write (Object A is only accessed by object B and no one else), 3) Single Write Multiple Read (Object A is written by object B but read by many others) and 4) Multiple Read Multiple Write (Object A is accessed by many others). Effective distribution decisions can then be generated for each object category such as replicate objects of category (1) across the system, objects of category (2) do not need to be remotely invokable, they only need to be co-located with their accessor object, objects of category (3) need to be remotely invokable and initially co-located with their writer objects etc. The objective is to place the most communicating objects in the same machine, thereby minimizing network traffic. Such classification and distribution strategies for each object along with their computation and communication significances are extremely helpful during initial object placement and preserving maximum resource utilization.

5. Self-Healing

In addition to providing self-organizing properties, ACE also instruments the targeted application with self healing primitives. It also inserts sensors and hooks at strategic points, so that at runtime, it is possible to interact with the managed code. During runtime, the injected sensors are responsible for saving the current status information (SI) of the application at certain check pointed position. When any runtime failure oc-
curs, the sensors also collect the context information (such as, object and method name, byte code program counter, formal and actual parameters, runtime stack trace) related to that failure and passes that as failure information (FI) to the fault analyzer. The fault repository holds models and information for some of the most frequently occurring failures. The fault analyzer uses those fault models (FM) to analyze the runtime failures. Figure 8 shows the overall fault architecture within ACE.

Once analyzed, the fault analyzer information (FAI) is passed to the fault healer for further processing. The fault healer takes the FAI and uses the last check pointed consistent status information (SI) and tries to reconstruct the faulted method, so that it could be restarted after the point where the failure occurs. If the fault healer is successful in creating such a recovery information (RI), it is passed to the managed application for recovery. Otherwise, the system administrator is notified of the new failure information (NFI) and the feedback is then saved in the fault repository as a new failure model (NFM). Once the new fault information is added to the repository, the fault healer will again try to heal the fault and will try to restart the application.

5.1. Code transformations

To add the extra functionality to an existing class required for the self-healing purposes, code transformation at byte code level is necessary. Byte code rewriting tool Javassist [6] is used to introspect and retrofit existing byte codes with self healing primitives. To handle any runtime exceptions, an extra try-catch block is inserted in every method. At runtime, if any runtime exception happens, it is caught and analyzed to find the root cause of that exception. A clone of the running method is created where, at the beginning of the code, new code is inserted to reinitialize the object with the last known state and to restart the method after the point where the fault occurs. Since the rest of the code already has the injected exception handling mechanism, any further faults will be handled same way as described above. Figure 9 shows the transformations that occurred during the lifetime of a method.

To support existing programs with desired autonomic behavior several adaptations are needed. Following few sections describes those adaptations.

5.2. Local variables

To save the state information of any running method inside the object, local variable values have to be saved during check pointing along with other necessary information, such as line number of the next instruction to be executed, value of actual parameters
and value of any object field. To gather the current state of the object, all the local variables in the current scope along with any fields and class variable’s values need to be saved. As the Java compiler does not save any local variable information (such as type information, variable scope etc.), the static analyzer gathers this information by analyzing the byte code and adds that information as a byte code attribute [13] to each method. During self-healing transformations, this attribute is used to create proper wrappers to save the corresponding object status.

The challenge is to recreate a local variable table from the byte code when there is no information saved for the local variables. Byte code instructions access local variables from a zero based array in the runtime stack as shown in Figure 10. Depending on the type of method (class or instance), variables and arguments are indexed differently as shown in Figure 10. Along with all local variables, method parameters and object instance (only with instance method) are also placed on that array. Depending on the data type, variables will occupy 1 or 2 cells in the array. Also, depending on the scope of the variable, cells could be reused by different variables within different scopes of the method body. To determine variable scope, variable index within the array and data type the algorithm (algorithm 1) shown in Figure 11 is presented.

The time taken by algorithm 1 is proportional to the number of local variables in the method. Only 8 bytes is
used to save all necessary information about a local variable. This keeps code inflation to a minimum. See Section 6 for some comparisons. The algorithm requires only one pass over the code to determine the local variables and it is performed simultaneously during the static analysis, therefore there is no extra overhead for executing this algorithm.

5.3. Adding checkpoints

To restart seamlessly from a crash or after a runtime exception, the internal state (fields, method parameters and local variables, next byte code to be executed) of the object has to be made persistent. According to the given user policy, instrumentation points in the code are determined and appropriate method calls are inserted to checkpoint the current state of the object. The following checkpoint-able positions are identified at byte code level:

1. After several write operations.
2. Before and after an I/O interaction.
3. Before and after any data interaction between two objects.

The checkpointing algorithm describes below adds a status object as a field into the target class, so that the checkpointed data can be saved through that field. The algorithm for checkpointing in byte code is per method basis and uses different byte code attributes, such as

1. For all byte code instructions in a method, do
2. Get next opcode (and operand, if any)
   2.1. If the opcode is of a data store type, then determine the type from the opcode (for primitive type). For reference (object) type, check the constant pool and determine the type information.
   2.2. Determine the variable index from the opcode suffix or from the operand. Also, determine the corresponding byte code line, where it is declared for the first time.
   2.3. If the opcode represents an already identified variable with the same data type, then update that variable’s scope information.
   2.4. If the opcode represents an already identified variable with different data type (same variable index is used by two variables in two different scope), then create a new variable entry.
code, exception, line number and local variable (generated by algorithm 1) attribute.
1. Insert an array of objects at the top of the method body. The size of this array is equal to the number of local variables in the method.
2. Determine the next checkpointable position in the byte code.
3. Assign object array with local variables having corresponding index value.
4. Load this object array and all other status information in to the run time stack.
5. Call the checkpoint method, which takes all the status information from the run-time stack and saves that in a persistent store.

5.4. Status data

During runtime, whenever any checkpoint is encountered, the current status of the object is incrementally saved to a platform independent disk file. We implemented the checked point data manipulation algorithm by synthesizing and re-engineering ideas found in existing fault tolerant techniques for distributed applications. Using pre-defined policies, users could modify the behavior of such algorithms to suit their needs. For example, to save disk space, once an object quits its execution and is garbage collected, all the corresponding status information related to that object are deleted from the status data file.

5.5. Runtime exceptions

To handle any runtime exceptions, the method body is encapsulated by another try-catch block (which catches the Throwable super class) to give the method another opportunity to continue after the statement where the exception is thrown. However, adding a new try-catch block introduces three different scenarios as follows, which need to be addressed at byte code level to fully support runtime exceptions:
1. *Subclass of Throwable or Exception is already being caught by the method body.*
   ACE solution: Re-throw that exception so that the enclosing try-catch block could catch that and continue with its self healing mechanism.
2. *Already an encapsulating try-catch block that catches either java.lang.Exception or Throwable.*
   ACE solution: Instead of adding a new try-catch to enclose the entire method, the catch block is modified and self healing mechanism is added.
3. *Nested try-catch block is already within the method body.*
   ACE solution: Re-throw every nested exception that being caught, so that the outer most enclosing try-catch block could catch that and continue with its self healing mechanism.

6. Evaluation

Experiments were conducted on the techniques presented above on byte code generated from Java programs having varying code structure and complexities. The techniques described in this paper can support Java byte code from JDK versions 1.2 to 1.5. A standard PC equipped with Pentium 4, 3 GHz CPU and 512 MB RAM running Windows XP is used to conduct all the experiments. To generate the local variable information, our algorithm (algorithm 1) produce smaller files than using the Java compiler’s debugging switch (-g during compilation). Since programmers of legacy code may not
use that switch, the self-healing code injector must rely on algorithm 1, which, as shown in Table 1 is efficient. Experiments with two different forms of checkpointing policies was conducted. In the normal policy, checkpoints are inserted after every statement in the method, whereas in the optimized policy, we used checkpointing decisions as described in Section 5.3. The resultant code has a $O(n)$ code inflation, where $n$ is the size of the original code.

Table 2 shows the timing data taken during execution. It is evident from Table 2 that, the algorithms takes linear time to execute. For the added functionalities in the code, we have to be comfortable with such a small increase of pre-processing time along with the running time.

7. Conclusion

This paper presents an approach to make legacy object oriented programs, or any code targeting the Java virtual machine, self-organizing self healing. This technique injects legacy user code with self-healing capabilities by statically analyzing the byte-code and instrumenting it in such a way that the application becomes a self-healing component. The goal is to make this transition as simple and easy to the application programmer in fulfillment to the goals of an autonomic system. Incremental construction of an evolving model [404] is taking place and the future research agenda includes code injection for other autonomic properties, optimization of such code injections, automatic program distribution and autonomic interaction between the autonomic elements.
8. References


