Using Code Instrumentation for Debugging and Constraint Checking

By

FILARET ILAS

This thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Engineering and Computer Science

August 2009

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of FILARET ILAS find it satisfactory and recommend that it be accepted.

____

Orest J. Pilskalns, Ph.D., Chair

____

Scott Wallace, Ph.D.

____

Wayne Cochran, Ph.D.

ACKNOWLEDGMENTS

This work would not have been possible without the support and encouragement of my advisor Dr. Orest Pilskalns under whose supervision I chose this topic and began this thesis. I would like to thank Dr. Scott Wallace for his guidance during this research. I could not have completed this accomplishment without the support of my family and particularly my wife, Ruth. I owe much to the entire staff at WSU for their openness and availability.

Using Code Instrumentation for Debugging and Constraint Checking

Abstract

by Filaret Ilas, M.S.
Washington State University
August 2009

Chair: Orest Pilskalns

In software engineering the need for secure and high quality software has spurred intense research activity in the areas on software debugging, testing and constraint analysis. Code instrumentation is a common technique used to track application behaviour. The most popular usages for code instrumentation are software debugging, performance analysis, monitoring, distributed computing and aspect oriented programming. Typical instrumentation techniques provide information about code coverage during software testing activities. Current approaches make use of instrumentation by inserting additional code that monitors the behavior of a specific component. This thesis presents and applies two novel approaches that use an instrumentation technique: (1) A *Runtime Debugging* approach is aimed at detecting and resolving runtime faults in object-oriented code. The approach relies on bytecode instrumentation in order to provide code coverage for predefined unit tests. The results are analysed using Reverse Engineered techniques. The approach consists in merging both succesfull and faulty code execution traces and detecting the faults by analysing the differences in the output traces. (2) A *Security Constraint Checking* approach uses

the notion of security consistency in designs. Byte code instrumentation techniques are used to provide code coverage for selected unit tests. Direct acyclic graphs are constructed from the output traces using reverse engineered techniques. The graphs contain object method calls in a similar manner to UML Sequence Diagrams. This approach uses the results of the instrumentation to check for consistency with design generated security constraints. Furthermore this approach analyzes these views for security inconsistencies, and generates a set of recommendations.

Contents

ACKNOWLEDGEMENTS **iii**

ABSTRACT **iv**

1 Introduction **1**

1.1 Motivation 1

2 Background and Related Work **3**

2.1 Code Instrumentation 3

2.2 Software Testing 5

2.3 Software Debugging 7

2.4 Unified Modeling Language 10

2.5 Reverse-Engineering 11

2.6 Constraint Checking 12

3 Automated Debugging Approach **13**

3.1 Runtime Faults 13

3.1.1 Notions of Distance 15

3.1.2 Cache (In)Consistency: An Example 16

3.2 Our Approach 23

3.2.1 Unit Tests 23

3.2.2 ByteCode Instrumentation 24

3.2.3 Execute the Tests 25

3.2.4 Partition paths 27

3.2.5 Aggregate paths 27

3.2.6 Generate Sequence Diagrams 31

3.2.7 Reason about fault 32

4 Constraint Checking Approach **34**

4.1 Secure Patterns at the Design Level 35

4.2 The Approach 36

4.2.1 Create Unit Tests 37

4.2.2 Instrument the Code 37

4.2.3 Execute the Tests 38
4.2.4 Construct Graph 38
4.2.5 Verify Consistency 38
4.3 Security Constraints 38
4.4 Experimental Results 39
4.4.1 Operation Access 41
4.4.2 Composition 42
4.4.3 Multiplicity 42
4.4.4 Mutable objects 43
4.5 Experiment results and Conclusions 43

5 Conclusions and Future Work **51**

List of Figures

3.1 Aggregated paths 28
3.2 Successful and Fault Partitions. 30
3.3 UML Sequence Diagram. 32

4.1 Banking System Class Diagram. 46
4.2 Banking System Sequence Diagram. 47
4.3 Operation Access. 48
4.4 Operation Access. 49
4.5 Operation Composition and Multiplicity. 50

List of Tables

4.1 OCL constraints results 44

Chapter 1

Introduction

1.1 Motivation

When software fails, it is often difficult to find the faulty code that is responsible for the fault. Much effort has been expended to create debuggers that operate at the source code level inside of integrated development environments (IDEs) for the purpose of tracking down hard to find faults. Testing is often used to reveal failures. Automation techniques are often employed to increase code coverage when unit testing. Large amounts of unit tests are often generated, however, once the failure is revealed, the units tests are not employed for finding the related fault or faults. Code instrumentation is often used for testing performance, however, instrumented code can provide a wide range of information including code traces. This leads to our first research question:

1. Can we define an automated or partially automated method for finding faults by using the failure revealing unit tests and code traces?

When designing and implementing software, constraints are often manually added or generated for the purpose of increasing the quality of the code. These constraints may be related to performance, security, reliability among other quality related requirements. Imposing constraints in the implemented code can be difficult and often situations arise that cannot be foreseen until the code is integrated and deployed. Instrumented code can provide a great deal of information about executing code. This leads to our second research question:

2. Can we define a constraint checking method that uses code instrumentation to examine the results of unit testing?

Both of these questions have the common theme of using code instrumentation for revealing faults and failures. This thesis defines and implements two approaches that answers these questions.

Chapter 2

Background and Related Work

2.1 Code Instrumentation

Code instrumentation is the technique used to inject measurement code into existing computer programs in order to generate additional data during the program execution. Instrumentation techniques in most cases should not modify the behavior of the program that is being instrumented. In practice, this is not possible because when ever you add code the performance will be altered to some degree. Code instrumentation techniques try to minimize the impact to existing code unless the objective of the instrumentation is to specifically and dynamically change the behavior. Instrumentation is often used to monitor or measure software performance or used to diagnose errors and report trace information. Instrumentation techniques have been published as early as 1975 [1] and initially consisted of manually inserting additional tracking code inside the programs for debugging purposes. One of the earliest references of using instrumentation tools was as component of the software Parasight [2] (paralel programming environment for symmetrical shared-memory multiprocessors). Parasight had a feature to dynamically create "parasite" programs inside the parallel

application. The parasites could be considered as instrumentation programs with the purpose of observing the target application.

More specifically, instrumentation refers to the capability of a program to integrate one or more of the following [23] :

1. *Code Tracing* represents the technique for retrieving informative messages regarding the program execution at runtime. The information is used by developers for debugging purposes. Usually it contains low level information corresponding to entry/exit method calls, thrown exceptions, etc. The volume of trace messages is much higher than the logging messages. Code tracing occurs during the developement phase and the resulting messages are analysed by developers.

2. *Debugging* represents the activity of locating and fixing programming errors during the program developement phase.

3. *Performance Counters* represent components used to monitor program performance. The purpose of performance anaysis (*profiling*) is to determine what parts of program needs to be optimized. A profiler represents a performance analysis tool that measures the behaviour of program at runtime in terms of frequency and duration of function calls. The profiler records a trace of events or a statistic of the observed events.

4. *Event logging* represents the ensemble of components that monitor application events. Event logging occurs after the application is launched and the target personnel are system administrators. Unlike code tracing, logging provides high level information and details that are easy to understand by administrators. For example, they may provide system administrators with information useful for

diagnostics and auditing.

Current approaches apply instrumentation to source code, bytecode or compiled code for understanding or modifying program behaviour. The technique can be applied *statically* before the code is executed or *dynamically* while the program is running. Basic techniques of program instrumentation insert instrumentation code at certain points of interest in the program. An instrumentation point can correspond to method entry, method exit, method call or exceptions. During the execution, the instrumentation code is then executed together with the original program code. There are different techniques used to place instrumentation points inside the program. The factors that influence the performance of the instrumentation process are: (a) The number of instrumentation points: too few points can lead to not achieving the instrumentation purpose, too many points can reduce the performance of the instrumentation process. (b) The location of instrumentation points: randomly or systematically according to the structure of the program.

For software written with interpreted languages, bytecode instrumentation is often more desirable since the source code is not modified.

2.2 Software Testing

Software engineering is the discipline designed to assure the reliability of computer programs [4]. It consists of the application of systematic approaches to software production from specification and design to implementation, testing and maintenence. Software testing is performed in order to detect faults in code execution. Testing helps estimate the reliability of the code. In general, software testing implies different activities like testing small pieces of code (*unit testing*) or inspecting the overall runtime behavior for a program (*integration testing*). The testing activity requires

the evaluation of the program output for a given set of input values. The input values can be randomly generated or can be carefully chosen in order to simulate malicious inputs. Testing implies understanding of program requirements and implementation. There are several documented testing models[4]: object-oriented testing, component-based testing, protocol testing, reliability testing. Many originally hoped that *Object Oriented Programming* (OOP) would reduce the number of failures, therefore reducing testing [20]. R. Binder developes a study based on testing procedural language programs as well as testing programs that are using the OOP paradigm. It is emphasized that finding failures in OOP is even harder than in *procedural programming* (PP) because the concepts used (polymorphism, inheritance, dynamic binding, etc.) are more complex. Automation is a very inportant step in software testing. In large scale applications, testing implies the need for a large number of test cases to test various functionalities. Methods to automatically generate test cases are important for software testing. Much effort has been expended on test automation and there still is a lot of research in the field of automated testing [59]. Dustin et al. define automated testing as "the management and performance of test activities, to include the development and execution of test scripts so as to verify test requirements, using an automated test tool" [58]. The main reasons for using automated testing are that manual testing is time consuming and testing automation incereases efficiency. Most industry tools are based on automatic test case generation. Samuel P. et al. show the difference between generation of test cases based on source code and those based on design specifications [51]. Source code based test case generation is more difficult requiring use of reverse engineering, especially for complex systems or in case of component-based development where the source code might not be available. Design based test case generation generates test cases from UML state diagrams. The UML state charts diagrams are the basis for automatic test case generation [51]. The

model-based test case generation approach extend the previous approach by integrating collaboration diagrams along with state charts diagrams.

2.3 Software Debugging

The *debugging* process represents the activity of locating the faults (bugs) after a program fails during the testing process. The meaning of debugging can be defined as the activity of finding and eliminating "bugs". Code debugging is performed in order to improve the reliability of the code after a failure is observed. Various debugging techniques and tools are used in order to locate the code that caused the detected failure. The success of finding the failure-causing piece of code depends on several factors:

1. Programming language used to implement the code: debugging is easier when used with high-level programming languages (i.e., source-level debugging)

2. Debugging tools used to inspect the code.

3. The debugging skills of the person who performs the debugging activity.

Many papers emphasize that fault localization is the most expensive and time consuming component of the debugging process [7], [8]. Therefore, the effort to symplify and automate this process has continually been the subject of research efforts. The current approaches to find faults are divided in two main categories: *code coverage* and *state coverage* techniques. The *code coverage* techniques are based on identifying statements that contain the fault. Test cases are used to run the statements and identify failures. Code coverage will identify a suspicious statement if the number of occurences of this statement in failed test cases is greater than in passed test cases. The process to identify a suspicious statement can be easily automated but the fault

localization among the suspicious statements is a much more difficult task and it mainly requires user intervention and analysis. The *program state* is defined as beeing a variable and its value in a particular program location [16]. The suspicious state localization requires the examination of program states in the suspicious statements. State coverage represents the technique to identify suspicious states.

Agrawal et al. present two debugging techniques used to localize faults using execution slices and dataflow tests [5], [6]. The techniques used are dynamic program slicing and execution backtracking. The supporting tool SPYDER was designed for C language programs. The notion of static/dynamic slice is used to represent a set of statements of a program that might affect the value of output on the execution of a specific input. An execution dice represents a set of decisions in one execution slice which do not apear in other slices. The fault localization process is searching for slice execution failure and in every step. The searching is narrowed by eliminating the slices that do not reveal a failure.

Eagan et al. present a tool named TARANTULA that is designed to partially automate the fault finding process by visually encoding test information to help find program errors [7]. The techniques consists of using color to visually map the participation of program statements in the outcome of program execution. The faulty statements can be identified by visually inspecting the color map. xSlice represents an improved color mapping technique based on the use of set operations in order to determine the representation of statements in the program [50]. An improved version of the tool TARANTULA is presented in [8].

R. DeMillo et al. propose a systematic process model to localize faults by exploring relationships between failure modes and failure types [9]. They systematically employ heuristics in order to reduce the search domain for faults.

Delta Debugging is an automated debugging technique based primarily on isolating causes of failure automatically. This technique was used to find failure-inducing circumstances automatically. Andrea Zeller presents the concept of delta debuging as a disciplined, systematic and quantifiable process [10]. The circumstances taken into account are program input, user interaction and changes to the program code. The debugging process relies on specifying the program to be run along with an automated test function that identifies failures during the program run. Delta Debugging is applied to the program under different circumstances in order to separate the failure-inducing circumstances from the irrelevant ones.

Zeller defines the execution of a failing program as a sequence of program states that induce the following states up to a failure [11]. The Delta Debugging technique is used to isolate the variables and values of the program state and therefore narrow the differences between program states down to a minimum set of variables that caused the failure. *Hierarchical Delta Debugging* represents an optimized variant of Delta Debugging. Misherghi et al. will apply the original Delta Debugging procedure to every level of a program input from a lower to a higher position in the hierarchy [12]. The approach is based on the fact that the input data is structured hierarchically, therefore the process of generating test cases can be simplified.

Podgurski et al. define the concept of *trace proximity* as an mechanism to cluster failing traces based on trace similarity calculated as an Euclidian distance between traces [14]. The clusters are classified based on the failure severity and then subdivided in smaller clusters until the fault is localized. Liu et al. propose fault localization approach based on failure proximity named R-Proximity [13]. The approach groups the collected failing traces in clusters that point to the same fault location. The debugging tool SOBER is used to automatically locate the fault inside the cluster.

Huang et al. propose a new approach of state coverage in order to identify suspicious states [16]. The process is similar to code coverage and consists in running multiple test cases and identifying the failed cases. A suspicious state will be revealed if it appears in more failed cases than in passed cases. The debugging tool Debutant is used to evaluate the fault localization technique.

Most of fault localization techniques are specialized in finding just one fault. Jones et al. proposes an approach for finding multiple faults [17]. The process of finding one fault at a time is defined sequential debugging. The term of parallel debugging is used to define the new approach based on finding multiple faults at the same time. Parallel debugging technique partitions the detected failing test cases in fault-focusing clusters. The defined clusters will be combined with the passed test cases and specialized test suites are applied to detect a single fault per cluster.

2.4 Unified Modeling Language

In software engineering, the *Unified Modeling Language*(UML) represents a standardized specification language for object modeling. The UML was designed to be compatible with object oriented software development methods. The design views offered by the UML describe objects and interactions between objects. UML has 13 types of diagrams: Class diagram, component diagram, composite structure diagram, deployment diagram, object diagram, package diagram, activity diagram, state machine diagram, use case diagram, communication diagram, interaction overview diagram, sequence diagram, UML timing diagram. These diagrams are classified in hierachies: structure diagrams, behaviour diagrams and interaction diagrams. In practice only a small subset of these diagrams are used to design a system: use case diagram, class diagram and sequence diagram.

Sequence Diagrams provide dynamic information about calls between objects. The old Message Sequence Chart technique is incorporated by UML under the name of Sequence Diagram. The Sequence Diagram represents the dynamic relationships between objects in a system, by showing method calls and logical decisions. The diagram consists of objects represented by boxes at the top of the diagram. From each box extends a line representing the life-line of the object. The arrows between the lifelines present the method calls between the objects. The elements of a Sequence Diagram are the object names, class names, method calls, method parameters, return types, decision constructs, and looping constructs. Using the UML diagram support in order to develop a system is known as *Forward Engineering*. The structure of a system defined by a set of diagrams is translated into source code by developers. The characteristic of generation of source code from UML diagrams is called forward engineering.

2.5 Reverse-Engineering

Reverse Engineering (RE) represents the process of discovering the technological principles of a system by analyzing its structure, function and behaviour. Reverse Engineering could be considered the opposite of Forward Engineering. There is an obvious relationship between the classical black box testing and reverse engineering. In the reverse engineering model, the output of the implementation phase is reverse engineered back to the analysis phase. There are several known purposes for using the RE model:

- If the source code is not available for the software, RE is used to discover the possible source code.

- If the source code is avaiable, RE can be used to analyse the product and to discover higher level aspects of the program.
- Security auditing, removal of copy protection, circumvention of access restrictions

Current approaches develop techniques to recreate UML diagrams using the RE model.

2.6 Constraint Checking

The design phase is very important in software development. Not always the implementations are consistent with the models specified in the design phase. In complex systems these inconsistences are hard to detect. There are many approaches and tools for model and consistency validation. M. Gogolla et al. propose the tool USE (UML-based Specification Environment) for the validation of UML models and OCL(Object Constraint Language) constraints [55], [56]. The tool is based on an animator for simulationg UML models and an OCL interpreter for constraint checking (multiplicity and association constraints). K. Wang et al. propose an approach for runtime checking of UML association related constraints using java bytecode instrumentation [57]. Their approach relies on defining the implications of UML class parameters such as: navigability, multiplicity, exclusivity and lifetime. These invariants are verified during program execution.

Chapter 3

Automated Debugging Approach

3.1 Runtime Faults

It is well known that it is almost impossible to guarantee fault-free software. In software engineering we define the notion of *fault* as incorrect (incomplete) code or an implementation that does not comply to the design specifications which might lead to a failure during the program execution. Therefore, failures in software engineering represent abnormal program behavior caused by faults [9], [20]. A runtime failure represents a failure that occurs when the program is running. Failure monitoring can be done using multiple tests during the program normal execution flow. Most programmers experience the problem that tests merely detect runtime failure, but they often do not reveal the location or source of the fault responsible for the failure. Usually the analysis of the nature of failures can help to pinpoint the possible region of code responsible for faults. In [9] they propose to use a fault localization model that makes use of failure modes, failure types and slicing heuristics.

Current approaches to find faults target coding errors. The most commonly used tool to find coding errors is the debugger usually built into an integrated development

environment (IDE). The traditional debuggers work at the machine level. Many IDE's come with modern front-ends integrated debuggers that allow users to monitor their program execution via a graphical interface. Debuggers offer functions such as runing a program step by step (stepping), execution breaking in order to pause execution and examine the current state of certain variables at a particular location in the program. The user has the ability to set a watch on variables for observation. Interactive symbolic debuggers provide the capability of setting breakpoints therefore whenever the program flow reaches any of the locations where the breakpoints were set, the program execution is suspended and the user can inspect the program by displaying current values of variables. Most recently some debuggers have the ability to modify the state of the program while it is running. Once a failure is revealed, the following steps can be taken using a debugging tool:

1. Place a watch on suspicious variables.

2. Set break points.

3. Execute the suspicious code segment and observe variables.

4. Step through code as needed.

Another facility offered by debuggers is *tracing*. Here we use specified tracepoints and whenever the control flow reaches any of the tracepoints, trace information will be displayed and program execution is continued automatically. During Single-stepping, the program execution is suspended after each statement and the control is returned to the debugger.

This approach to find runtime faults works reasonably well if the failure occurs in close proximity to the underlying fault, for example, when both fault and failure occur within the same method. As the distance between failure and fault increases, however,

the traditional approach of setting break points and stepping through executing code becomes less effective.

3.1.1 Notions of Distance

The process of finding faults by using information about failures uses the notion of a "distance" between failure and fault. O. Pilskalns et al. present simple descriptions regarding the notion of distance [53]. They define and describe the differences between the syntactic and heuristic distances. The *syntactic* distance is measured as lines of executed code from the line of code where the failure occurs until the line of code where error causing code is found. The search for the fault can be exponential since it is almost impossible to accurately know which branch of code has been taken during a particular execution and the developer need to examine all these branches. The *heuristic* distance is based on the developer's intuition about where a particular fault may be located compared to the actual location of the fault. The developer creates a priority-queue of possible locations of the fault that would be examined until reaching the actual fault location. The distance between the first location in the queue and the location of the true fault in this priority-queue is defined as *heuristic* distance. This distance differs from developer to developer based on their knowledge of the source code and the established priorities.

The traditional debugging approaches make use of both syntactic and heuristic distances. For simple systems, syntactic distance would most likely be enough in order to find faults. In complex systems, as the syntactic distance increases, the developer is likely to spend significantly more time looking for the fault by setting break points based on a priority-queue of possible fault locations.

The correlation between syntatic and heuristic distance is not fundamental. Our

approach will present a tool that can reduce or even remove the correlation between these two measures. The tool can be used to quickly isolate sections of the source code that are likely contributors to the failure. By visually identifying these potentially problematic sections, our tool is able to decouple the relationship between syntactic and heuristic distance, thus making hard-to-find faults readily apparent.

3.1.2 Cache (In)Consistency: An Example

In this section, we provide a detailed illustration of one failure situation in which the heuristic distance is likely to be relatively high [53]. We will present a simplifed Data Storage system that implements data caching techniques to boost performance and ensure data synchronization. In this exemplar situation we store two-dimensional point information in a cache. This example observes the interaction between `DataConsumer` and `DataSource` objects. The `DataSource` object provides a volatile dataset based on its own internal state (see Listing 3.1). The `DataConsumer` object examines the datasets provided by multiple `DataSource` objects.

`DataSource`'s job of generating the dataset may be complex, requiring significant computational overhead. In this case, significant computation time can be avoided if the `DataConsumer` use an optimal caching policy for storing these datasets. The object in Listing 3.2 follows this model. The two `DataConsumer` methods (`invert()` and `setInitialValues()`) are used to manipulate the underlying `DataSource` objects. This, in turn, impacts the datasets that will be produced by the `getData()` method.

The caching approach is used by `DataConsumer` inside the `getPoint()` method. The `getPoint()` method is illustrated in Listing 3.3 and is intended to retrieve a particular data point for one of the datasets.

Listing 3.1: DataSource

```
public class DataSource {

  Version version = new Version();

  // two internal state values impact the results
  // returned by getData()
  private int type;            // internal state
  private double initialvalue; // internal state

  public int getVersion() {
    // the version is used to indicate changes to the
    // DataSource's internal state. So long as the
    // version remains unchanged, calls to getData()
    // should return the consistent results.
    return version;
  }
  public DataSet getData(int n) {
    // do something potentially complicated based on the
    // internal state and return a dataset with n elements...

    // source code continues...
```

Listing 3.2: DataConsumer

```
public class DataConsumer {
  DataSource[] srcs;
  // the cache maps DataSets to their DataSource and keeps a
  // Version number to quickly check if the DataSet is stale
  Cache<DataSource,DataSet,Version> cache;

  public void setInitialValues(double d) {
    srcs[0].setInitialValue(d);
    srcs[1].setInitialValue(d);
  }

  public void invert() {
    int t = srcs[0].getType();
    srcs[0].setType(srcs[1].getType());
    srcs[1].setType(t);
  }
  // source code continues...
```

Listing 3.3: DataConsumer.getPoint()

```
public Point2D getPoint(int s, int n) {
  CacheEntry e = cache.get(srcs[s]);
  DataSet ds;
  Version dsVersion = srcs[s].getVersion();

  if ( e == null ) {
    // if the dataset is not in the cache, fetch it
    ds = srcs[s].getData(size);
    cache.put( srcs[s], ds, dsVersion );
  }
  else if ( !e.getVersion().equals(dsVersion) ) {
    // inconsistency detected -- cache is stale, refresh it
    ds = srcs[s].getData(size);
    cache.put( srcs[s], ds, dsVersion );
  }
  else {
    // cache seems consistent
    ds = e.getData();
  }
  return (Point2D)ds.get(n);
}
```

The cache relies on a simple method for determining cache consistency. When a dataset is obtained from a `DataSource`, the version is also obtained and stored in the cache. So, to retrieve a data point, `getPoint()` first looks for a cache entry. If an entry is found whose version matches the `DataSource`'s current version, the entry is determined to be consistent and the point is fetched directly from the cache. Otherwise, `DataSource.getData()` is invoked to get a new copy of the dataset thereby refreshing the cache and providing the return results.

In this particular example the data retrieval relies on `DataSource.getVersion()` to indicate when the `DataSource` has changed in a manner that will affect the dataset it produces. Cache inconsistency occurs if this assumption is violated. Consider the two methods below, both of which change the internal state of a `DataSource` object.

Listing 3.4: DataSource methods

```
public void setInitialValue( double v ) {
  initialvalue = v;
  version = version.next();
}


public void setType( int t ) {
  // BUG: setting the type affects the data that an instance
  // would produce. We should increment the version number
  // to indicate such a change.
  type = t;
}
```

Listing 3.4 illustrates a type of fault that may be difficult to identify, especially given the context of how the `DataSource` object will be used. The cached dataset becomes susceptible to inconsistency because the `setType()` method does not appropriately increment the version. A fault occurs in a special case when `setType()` method is invoked between calls to the `DataConsumer`'s `getPoint()` method. More precisely the fault will occur if the `setType()` is called after the `setInitialValue()` method call which alters the version. The outcome is that syntactic distance is likely to be high since the fault will not occur in the same method that the stale data is used, nor will backtracing to the `invert()` method reveal the fault. The developer's search will need to continue back to `setType()`.

Besides the syntatic distance between fault and failure is high, there is also likely to be significant heuristic distance in this situation. Consider the two unit tests in Listing 3.5.

In both tests, identical methods are invoked in a different order and only one test (`testFailure`) produces a failure. This failure occurs because `invert()` calls `setType()` and this method is invoked between calls to `getPoint()`, therefore `DataConsumer`'s cache becomes inconsistent.

The developer might be inclined to assume that `invert()` and all of the methods it calls all work correctly since the success of the first test. Therefore in his priority queue the developer might look at many other code locations before inspecting the `invert()` and `setType()` methods. In this case the heuristic distance between failure and fault would be very large.

Listing 3.5: Unit Tests

```
public void testSuccess() {
  consumer.invert();
  for( int i = 1; i < nTests; i++ ) {
    consumer.setInitialValues( i );
    assertEquals( new Point2D.Double( 0.0, i ),
            consumer.getPoint( 1, 0 ) );
    assertEquals( new Point2D.Double( 3, i+6 ),
            consumer.getPoint( 1, 3 ) );
    assertEquals( new Point2D.Double( 3, i-9 ),
            consumer.getPoint( 0, 3 ) );
}}


public void testFail() {
  consumer.setInitialValues( 1 );
  assertEquals( new Point2D.Double( 3, 7),
          consumer.getPoint( 0, 3 ) );
  assertEquals( new Point2D.Double( 3, -8),
          consumer.getPoint( 1, 3 ) );
  consumer.invert();
  // fails! cache is out of sync!
  assertEquals( new Point2D.Double( 3, 7 ),
          consumer.getPoint( 1, 3 ) );
  assertEquals( new Point2D.Double( 3, -8),
          consumer.getPoint( 0, 3 ) );


}
```

3.2 Our Approach

Our approach relies on differentiating between successful code execution and fault revealing code execution. By tracing code execution of both successful and fault revealing unit tests, we can create directed acyclic graphs that show the differences. These graphs can be transformed into Unified Modeling Language (UML) Sequence Diagrams. UML Sequence Diagrams are often used by Software Engineers to represent the behavior of program in the design phase. Here we use Sequence Diagrams to reveal faults while eliminating the unnecessary clutter of code-level detail. The following steps outline our approach:

1. Create Unit Tests
2. Instrument the source code so message paths (and associated objects) can be traced.
3. Execute the tests and record objects and message paths.
4. Partition paths into fault and non-fault revealing partitions.
5. Aggregate all paths into a single graph and differentiate based on fault partitions.
6. Generate Sequence Diagram from differentiated graph.
7. Use Sequence Diagram to reason about fault.

3.2.1 Unit Tests

Our approach relies upon unit tests that provide coverage of the code that produces the fault. Additional unit tests are needed to provide coverage of the code using test

cases that do not fail. Therefore, our method is applicable when the application is mature enough that some unit tests succeed, but not so mature as to pass all of the unit tests.

Test coverage is important since our objective is to differentiate between successful code and faulty code. If the coverage is inadequate then a failed unit test may have little in common with successful tests negating the usefulness of differentiating the two. Ideally, adequate coverage would reveal localized differences in the object method traces of successful and failed unit tests. It is reasonable to assume that as test coverage increases so should our success rate. However, there is the possibility that the fault exists at a lower level (statement level) of the code.

3.2.2 ByteCode Instrumentation

Instrumenting the code is the process of inserting tracing code that records the method execution calls between objects. This can be accomplished by inserting code that logs each method call, the calling object's id, and the calling object's class type. Logging could be done at the source code level, but would require tools for both inserting and removing the instrumentation code. We take an alternate approach that simplifies the process for the developer by automatically inserting tracing methods into the Java Byte code. After the debugging process is complete, the Java Byte code can be discarded, and the (unmodified) source simply recompiled.

Since our goal is to create Sequence Diagrams, we chose to track method calls. However, we could choose a lower or high granularity level. For example we could choose to track the execution sequence line by line, or we could only track messages between components.

Our analysis tool uses the utilities in *org.apache.bcel* java library in order to ac-

complish the instrumentation of the byte code. Classes selected for instrumentation are loaded and injected with a reference to a static object named *LumberJack*. *LumberJack* uses a static counter to keep track of method calls and inserts trace code for each method *call* and each method *return*. The tracing code keeps track of the following information in the bytecode:

1. *method* contains information such as class name, method name and method signature (returned type and arguments);

2. *location* tracks the line number where the method occurs in the source code and indicates if the method is a return call or an initial call;

3. *runtime* tracks the order of calls during execution;

In addition, the *LumberJack* class provides tools for printing an XML representation of the trace logs.

3.2.3 Execute the Tests

During the execution process, the instrumented byte code is traversed using the unit tests. For this example, the unit tests provide branch coverage of the code. Each unit test generates an object-method trace through the code, which is recorded to an XML trace file. Listing 5 shows a sample of the data recorded in the trace file after the unit tests have been executed on the Cache Example. Each unit test trace is tagged as successful or unsuccessful based on the outcome of the test.

Listing 3.6: trace.xml

```
<vertex>
  <method>DataConsumer.invert()V</method>
  <location>called: DataConsumer.java:40
  </location>
  <runtime>myCallCount=1</runtime>
</vertex>

<vertex>
  <method>DataSource.getType()I</method>
  <location>called: DataSource.java:47
  </location>
  <runtime>myCallCount=1</runtime>
</vertex>

<vertex>
  <method>DataSource.getType()I</method>
  <location>return: DataSource.java:47
  </location>
  <runtime>myCallCount=1</runtime>
</vertex>

<vertex>
  <method>DataSource.getType()I</method>
  <location>return: DataSource.java:47
  </location>
  <runtime>myCallCount=1</runtime>
</vertex>

...

<method>DataConsumer.invert()V</method>
  <location>return: DataConsumer.java:45
  </location>
  <runtime>myCallCount=1</runtime>
</vertex>
```

3.2.4 Partition paths

The previous step provides enough information to allow us to differentiate between successful and failed code execution. The generated trace files contain information about each test and thus potentially the fault. Given such information it is trivial to partition the paths into what we have named fault and non-fault revealing partitions. Thus, the trace files associated with successful tests are classified as non-fault revealing and likewise unsuccessful tests are classified as fault revealing.

3.2.5 Aggregate paths

The merging algorithm aggregates all the trace paths generated during the unit test execution. Merging results in an acyclic graph where the vertices represent the actual method calls and the directed links between vertices specify the order of the method calls. Figure 3.1 displays the acyclic graph obtained for the Cache example. Every vertex in the graph contains the following information: id, method, location. Before merging every vertex id is named based on the unit test name and the index of the vertex in the trace path. After merging the vertex, the id may be renamed with a unique alpha-numeric symbol beginning with m to indicate that two vertices have been merged.

Trace paths are aggressively merged by looking for object-method calls that coexist between traces. Merging the results of two identical unit tests results in a linear graph with no branching. If two unit tests traverse different object-method calls, however, the process will introduce branches into the graph which may later merge back to the same path.

The merge algorithm iteratively processes object-method call traces. At each step, a new trace t_i is added to the graph G. Note that both t_i and G are directed acyclic

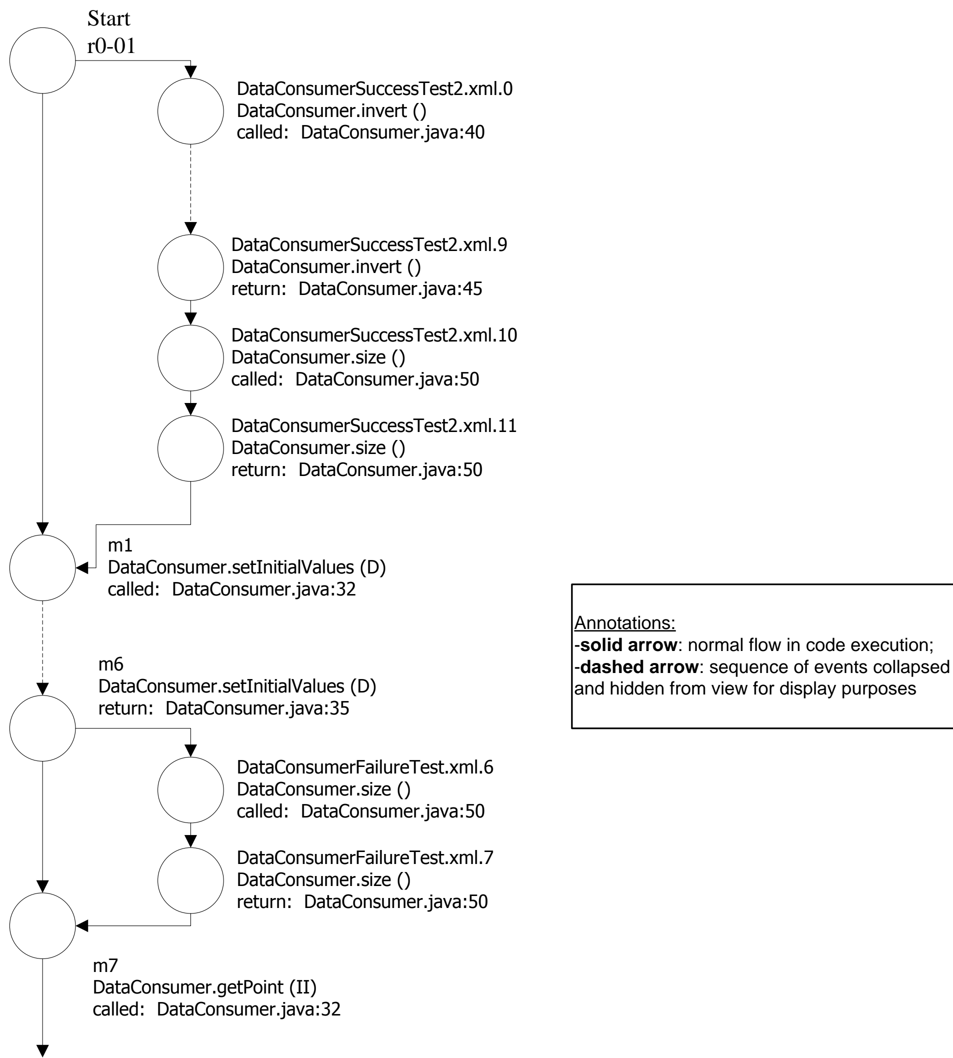


Figure 3.1: Aggregated paths

graphs but t_i has a branching factor of exactly one. When the algorithm begins, the graph G consists of only a single root node with the label *start*. When the algorithm is complete, G is the aggregation of all execution paths through the unit tests. The process follows five steps:

1. Initially, set m_g to the root of G and m_t to the root of t_i.

2. Place a pointer p_g at m_g and another pointer p_t at m_t.

3. For each child of the node pointed to by p_g, scan foward in t_i for a matching object-method call.

4. If a matching pair is not found, repeat the scan forward in t_i from m_t trying all descendents of p_g in a breadth-first fashion. If no match is found, add the directed graph rooted at m_t as a new child of the node pointed to by m_g. The algorithm is now complete; no new merging has occured.

5. Otherwise, the nodes at p_g and p_t are the same object-method call and represent a "rejoining" of the graph G and the trace t_i. Splice a new branch between m_g and t_g that includes the sequence between m_t and p_t exclusive of these endpoints. Repeat from step 2.

The algorithm above aggressively merges traces to reduce the number of branches in the aggregate representation. This results in a less complex and smaller graph than would be created if braches were not allowed to merge back to one another.

Figure 3.1 illustrates the result of merging two traces: one successful and one un-successful unit test. The resulting graph is rooted at the node labeled *start*. A branch occurs immediately, indicating that the initial execution paths of the successful and failed unit tests differ. A dotted line indicates that a sequence of events has been collapsed and hidden from view for display purposes. The user interface allows us to examine the method calls in details if required. After calls to and returns from `invert()` and `size()` the execution traces merge and execute the method `setInitialValues()` which is the first object-method call in the failed execution trace. Both traces return from that method before once again diverging briefly.

In Figure 3.2, the trace continues with a new branching after `getPoint()`. During the first four DataConsumer.getPoint() method calls the graphs correspond, and thus the nodes in the two unit tests merge together as expected. A new branching occurs caused by two different method calls in the unit tests as shown in the Figure 3.2. The right branch corresponds to the successful partition. The left side branch corresponds

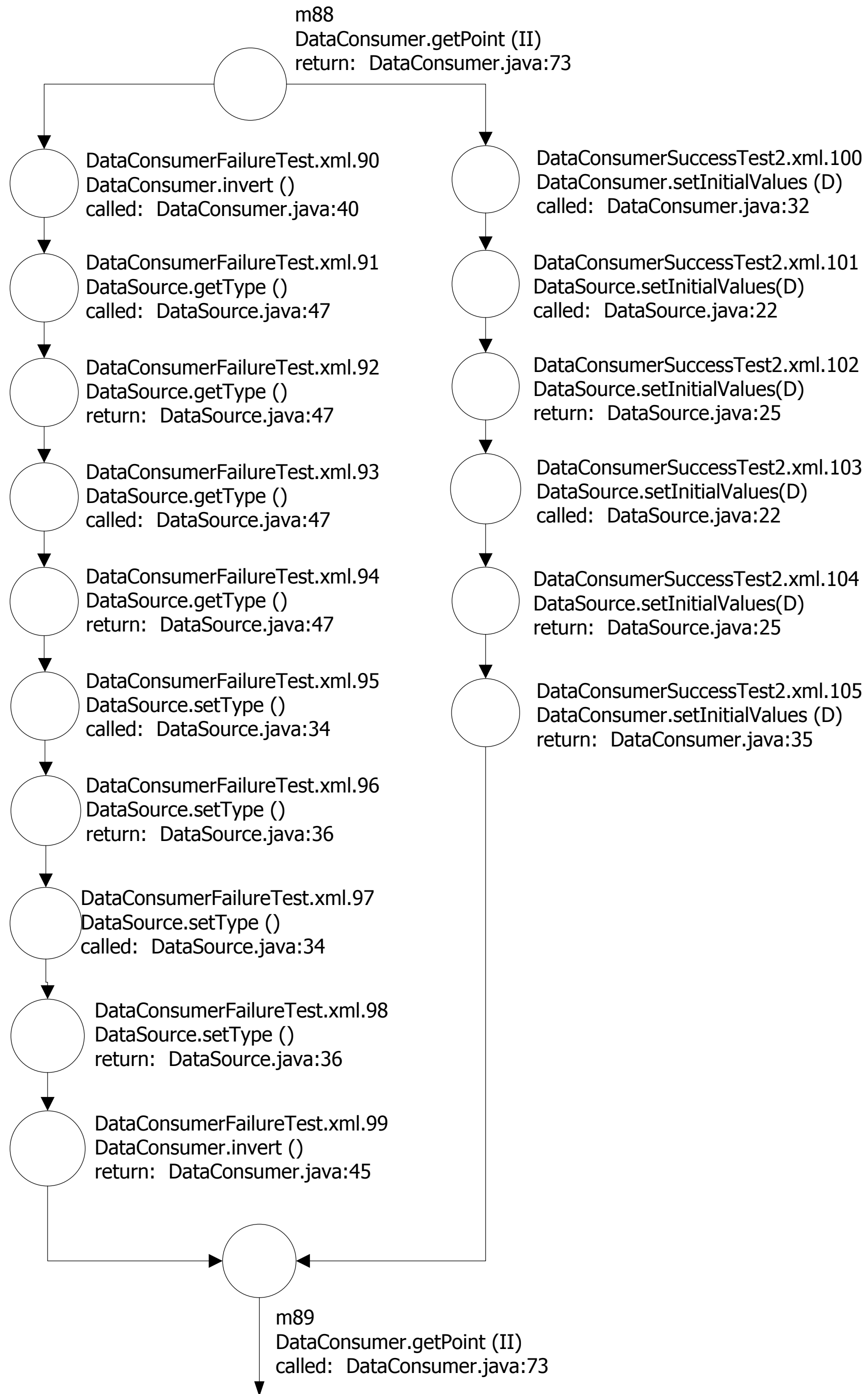


Figure 3.2: Successful and Fault Partitions.

to the fault revealing partition. The method call DataConsumer.invert() will cause cache inconsistency for the DataConsumerFailureTest unit test. Therefore during the next method call DataConsumer.getPoint() the normal code execution fails after the method DataSource.getVersion() returns a value different from the actual cache version. The successful unit test trace continues with the vertices corresponding to the method calls from the DataConsumerSuccessTest2.

3.2.6 Generate Sequence Diagrams

A UML Sequence Diagram is a behavioral representation of objects interacting with each other via method calls. In the previous steps we created an acyclic graph representing both successful and fault revealing unit tests. The graph is also a representation of objects interacting with each other. Therefore we can use the graph to generate UML Sequence Diagrams. We generate a Sequence Diagram for each branched segment of the direct acyclic graph that that contains a failed test. Each branch is visualized as a *combined fragment*. A combined fragment is used to visually display the conditional flow in a Sequence Diagram. Thus Sequence Diagrams are systematically generated by traversing each vertex, v, in the graph and using the following steps:

1. When a vertex contains more then one child and at least one child represents a failed test, create a new Sequence Diagram (if not already created) and create a combined fragment for each child vertex. Each child vertex should be represented as an object in the Sequence Diagram.

2. For each newly added child vertex, check its children, if is contains only one child, add the child vertex to the combined fragment and connect to the parent

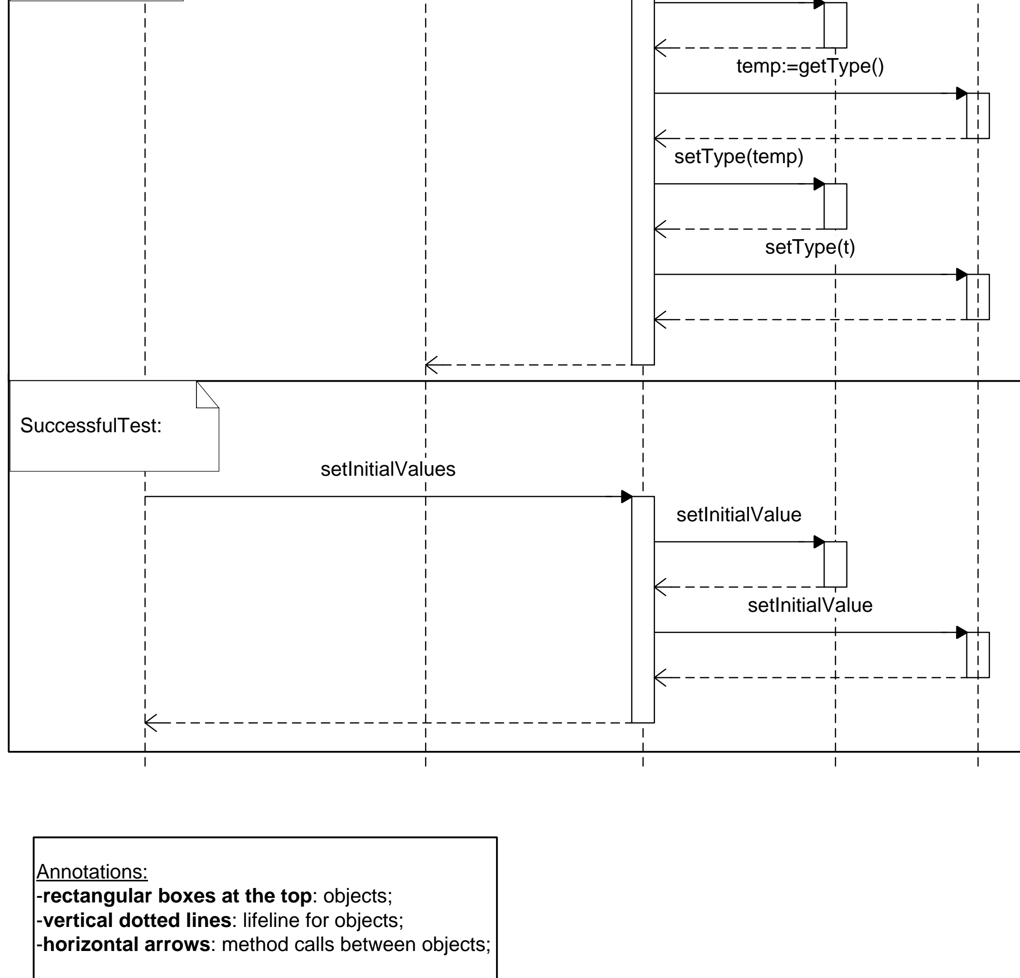


Figure 3.3: UML Sequence Diagram.

vertex using the method call in the previous vertex (label appropriately). If it contains more than one child return to step one.

Using this algorithm we created the Sequence Diagram in Figure 3.3 which represents the branched segment in the directed acyclic graph shown in Figure 3.2.

3.2.7 Reason about fault

The Sequence Diagram shows where we can find the section of code responsible for the failure of the unit test. It now seems obvious that the method *invert()* with it's underlying call *setType()* causes the undesired behavior. The DataConsumer's cache becomes inconsistent when this method is invoked between calls to *getPoint()*. Therefore the heuristic distance between failure and fault is mitigated. We can easily

find where the actual fault occurs. The diagrams do not reveal why this method causes the inconsistency. Once the fault location is found, the developer can examine the source code and develop a solution.

Chapter 4

Constraint Checking Approach

Secure coding practices are often applied during the implementation phase. An example of a secure coding practice may be to give an object the least amount of privileges necessary to complete a required task. Programmers follow many guidelines to ensure that their code is impervious to attacks. Traditionally, these rules have been important in the coding step of the development process. However, it has been demonstrated [20] that security must be a pervasive concern through out the development life-cycle. Thus we need methods to enforce security at all phases of development, namely:

1. Design.
2. Implementation (coding).
3. Testing.
4. Deployment.
5. Maintenance

O. Pilskalns et al. show that secure coding, testing, deployment and maintenance are significantly affected by the design [30]. Furthermore,they demonstrate that security can be increased with automated methods to detect anti patterns in code [31]. They proposed a method of generating security constraints based on security principles and known anti patterns. In software engineering, anti paterns represent design patterns that might be commonly used but it is ineffective in practice. However, Pilskalns et al. do not provide a method of enforcing the constraints. In this section, an approach is demonstrated that uses the generated constraints from Pilskalns et al. The approach uses instrumented code to trace the execution of a program. The trace code is then checked using the generated constraints. If the trace code is inconsistent with the generated constraints then a fault exists in the program under test.

4.1 Secure Patterns at the Design Level

Applications handling sensitive data are risk-prone. Because of this, clearly defined security patterns have been developed in the field of computer science. A pattern can be defined as a general reusable solution used to solve a common problems that occurs in designs. Secure design patterns address security vulnerabilities on all levels of system life-cycle: from design specifications to implementation providing details how to implement different functionalities in the system. Secure patterns involve restricting access to methods, object classes, variables, and other data that could potentially be exposed to unwanted clients. A good example is serialization. If enabled, an attacker may obtain a serialized object and potentially view private fields within that object. A method intended for the user, such as viewing or searching, should only be permited to the intended person. Thus, an attacker may view private data such as a SSN or bank account information while looking for an address. Several

individual practices like this may be combined into a high-level pattern which maps to a general design model (such as the Unified Modeling Language (UML)), which may take the form of either sequence or class diagrams. In addition to the UML specifications (either Class or Sequence Diagrams), the *Object Constraint Language* (OCL) allows the designer to specify constraints on the system which may not be clear from a diagrammatic perspective. Thus, the OCL provides a useful tool for rendering secure principles to the design.

4.2 The Approach

The proposed method involves combining the above approaches for specific use with respect to security constraints. Our intention is to use a simplified version of the debugging tool presented in the chapter 3, focusing on the role of the debugger to check for security inconsistencies as specified by OCL constraints. Since it has been shown that we can use the OCL for describing secure aspects of a system, we may test whether or not the implementation follows the original constraints specified in the design phase. In this case, we are focusing on the reverse-engineered Sequence Diagram analysis. The debugging tool focuses on localizing the code that caused the failure of the tested system. We want to use this feature in order to find the exact location of the code responsable for any inconsistency between the UML design specifications and the current implementation.

After the debugger generates a graph from unit tests, the graph may then be compared against the original pattern represented by the Sequence Diagram. The user can then visually inspect the two representations and reason about the inconsistencies with respect to security constraints. The following steps outline the approach:

1. Create Unit Tests

2. Instrument the source code so message paths (and associated objects) can be traced.

3. Execute the tests and record objects and message paths.

4. Construct directed acyclic graphs for every trace.

5. Verify if the security constraints specified in the design phase hold for each graph.

4.2.1 Create Unit Tests

The purpose of unit testing is to provide coverage for the code we want to check to ensure it is consistent with the specifications from a security perspective. Previously we used unit tests to differentiate between successful and faulty code. However in this approach unit tests provide coverage of the code we are checking for consistency. Thus the unit tests are used for code coverage regardless if they are fault revealing or not.

4.2.2 Instrument the Code

Code instrumentation involves inserting byte code into the objects. This code allows message paths (and as-sociated objects) to be traced. The instrumentation process inserts tracking code that records the method execution calls between objects. In our case, we will track the method calls. This allows us to eventually compare the trace with the generated security constraints that where obtained by analyzing Sequence Diagram in the design phase.

4.2.3 Execute the Tests

During the execution, the instrumented byte code is traversed using the unit tests. Each unit test generates an object-method trace through the code. Currently, the output from the test execution is the same as the debugger. However, as the constraint checking approach is automated, the code output will be modified to increase the ease of checking constraints.

4.2.4 Construct Graph

The trace paths defined during execution allow us to construct an acyclic graph. We merge the results in a graph where vertices represent the method calls and the directed links between vertices specify the order of the method calls. Every vertex in the graph contains the following information: id, method, and location. The graph allows us to see relationships between objects via method calls.

4.2.5 Verify Consistency

Since we have generated a graph from the unit tests, we may visually verify that the constraints specified in the original Sequence Diagram hold for the implementation.

4.3 Security Constraints

Pilakalns et al. define four rules which may be applied to the UML model [52]. These rules check for consistency between UML Class Diagrams and Sequence Diagrams. They define the following rules:

1. *Operation access: check if an object is allowed to use operations provided by another object;*

2. *Composition: check if the life span of an object doesn't exceed the life span of the container object;*

3. *Multiplicity: checks if the number of instances of an object doesn't exceed the maximum number of instances allowed to complete a task;*

4. *Mutable Objects: check if operations pass class variables by value or by reference.*

Furthermore, their approach translates these rules to OCL constraints. Our tool allows us to assess each of these types of security constraints.

4.4 Experimental Results

Our method was tested successfully on a simplified version of a banking system. We chose this type of example because in banking systems security and reliability of the components is very critical. The specific context of the system presents the standard communication between a bank teller and a private client. The communication between the system and the components *teller* and *customer* is performed via messages. The messages need to be encrypted and signed using RSA encryption. Any request for a certain service is managed by the transaction manager. The transaction manager can begin and end a transaction as well as transfer an encrypted message to the bank account. Any transaction will not be finalized until the transaction manager can indicate the end of transaction. The purpose of designing this system is to provide an increased level of security and reliability by specifying an additional set of constraints.

This system is modeled using UML and implemented using the Java programming language. The structure of the system is presented in Figure 4.1. The Class Diagram presents the simplified structure of the system in terms of classes, atributes

and relationships between classes. The *RSA* class provides encryption and digital signing. The *Account* class contains information about a specific bank client. The *TransactionManager* class starts and ends transactions between a client and bank and provides protection against failures.

The real world banking system are much more complex and use SSL protocols to enhance the security of the comunication between banking system and clients. The purpose of using this example is not to implement a fully functional banking system, but to check if the security constraints specified in the design phase hold. Therefore we will focus on the object interaction rather than simply coding the tasks specific for any component. The main simplifications from the real world banking systems are:

1. We do not implement any of the client authetication mechanisms. We consider that the clients are securely authenticated by default.
2. We do not use SSL protocols to secure the communication between clients and banking system.
3. We do not implement a full RSA encryption or signing since this is beyond the purpose of this experiment. It uses the default DES provided by the Java API.

Our approach will trace object-method calls; therefore, we will focus on the interaction between components. The basic behavior of the system is illustrated in the Sequence and Class Diagrams in figures 4.1 and 4.2. We will write unit tests to trace the code coverage for different situations in order to illustrate how to use this tool to check if the system is consistent with the constraints specified in the design phase.

4.4.1 Operation Access

First of all we will check for the operation access constraints. Pilskalns et al. [52] propose a set of operation access constraints which we will test in the following use cases. The constraint:

$context\ Account :: create()$

$inv : oclSet = \{Teller\}$

$inv : oclSet- > includes(source.type)$

will grant the access to $theAccount.create()$ method only for instances of the $Teller$ object. In the graphs in figures 4.3 and 4.4, we will differentiate between objects accessing this method in order to assess the system security. Every object-method call will be represented in the direct acyclic graph with two vertices corresponding to the entry point and return. The links between vertices correspond to the order of the method calls. Each vertex contains the following information: id, location and method. We can easily point out the vertices corresponding to the $create()$ method call. The order the methods are called is preserved in the graph, therefore we can search backward in the graph to see if there is any instance of the $Teller$ object responsible for invoking the $create()$ call. We find out that the $create()$ method is called by the $Teller.transfer()$ method. In this situation we conclude that the system might be secure regarding this operation access constraint. The example in figure 4.4 illustrates that this constraint is not held. The $create()$ method is called by an instance of the $Consumer$ object.

4.4.2 Composition

The life span of an object is represented in the graph using two vertices corresponding to the "called" and "return" locations. In the figure 4.5 we will verify whether or not the following composition constraint holds:

$context\ Message :: create()$

$\quad inv : account.AllInstances- > notEmpty()$

This constraint specifies that the life span of a message instance must not exceed the life span of an account instance. When we count the number of times the account object accesses methods, we observe that the life span of the account instance persists for the entire $Teller.transfer()$ method call (from creation to deactivation). The message instance also appears to be active from creation until it is destroyed at the end of transaction. Therefore, in this example, we see that the life span of the message instance does not exceed the life span of the account instance.

4.4.3 Multiplicity

The multiplicity rule implies that the number of instances of an object does not exceed the maximum number of instances allowed to complete a task. We will define the following constraint specifying multiplicity:

$context\ Message$

$\quad inv : self.AllInstances() <= 1$

This constraint limits the number of active message instances to one per transaction. According to the constraint, a message instance should be destroyed at the

end of transaction. The graph in figure 4.5 shows that this constraint is not correctly applied in our system. We see that there are two consecutive transactions for the same account that are using the same message instance. Due to this fact, we may conclude that the system is not consistent with the design specifications.

4.4.4 Mutable objects

This rule identifies mutable objects in method calls and requires a copy of passed objects to be made using constructor calls. We will define the following constraint specifying mutable objects:

$context\ Teller$

$\quad post : Message() : hasReturned()$

The constraint requires that a copy of the Message object to be made before a transaction occurs. We will demonstrate the results of applying the algorithm on the Diagram shown in 4.3. In this diagram an instance of the Teller class creates a new Message class and passes it to the the Account instance. The Account class does not create a copy.

4.5 Experiment results and Conclusions

The effectiveness of this approach was prooved by using the results of an academic experiment [31] where computer science students at Washington State University had to implement these four sets of constraints on a more elaborate version of secure client server system. The students were divided in two groups: a control group that did not receive any OCL constraint statements to include in their work, and a group

that received four sets of OCL constraints to implement in their work. All students received the same UML diagrams and an incomplete coded project in order to start working on their assignment. When the experiment was initially executed, unit tests where used to test the OCL constraints. To test our tool, we used the code generated by the students, and executed the code using the unit tests. We found that our constraint testing tool found the same errors except by using trace code. By using this approach we could see where the executed code violated the constraint. Unit tests only show that a constraint is violated, but not where. These results are promising, however, we need to test our approach on a larger system. The results from the initial experiment obtained are stated in Table 4.1.

Table 4.1: OCL constraints results

	Access	Result	Multipl.	Result	Compos.	Result	Mutable	Result	TOTAL
OCL Group									
student1	Y	P	Y	P	Y	P	Y	P	4
student2	Y	P	Y	P	Y	P	Y	P	4
student3	Y	P	Y	P	Y	P	Y	P	4
student4	N	F	N	F	N	F	N	F	0
student5	Y	P	Y	P	Y	F	N	P	3
Control Group									
student1	N	F	N	F	N	F	N	F	0
student2	Y	P	N	F	N	F	N	F	1
student3	N	F	N	F	N	F	N	F	0
student4	N	F	N	F	N	F	N	F	0

Most of the students in the OCL group implemented the given constraints (Y/N) in many different ways and the effect of the OCL implementation was analysed visually using our tool. Most of the constraints implementations were successfuly detected (P) using our tool with one exception (student5) where the implementation was incorrect (F).

The students in the Control Group didn't have to implement any of the OCL

constraints and the effects were noticed in the final OMDAG representation.

The efficiency of using our tool to check the constraints in final implementations was successfully proved in this particular case.

Overall, given that secure constraints exist for applications, we may use our approach to test whether or not software is indeed following a secure design. We propose a solution for discovering security inconsistencies by comparing a trace of the software generated by unit tests with the original Sequence Diagram. This reveals whether or not the software correctly implements the specified OCL constraints from the original document.

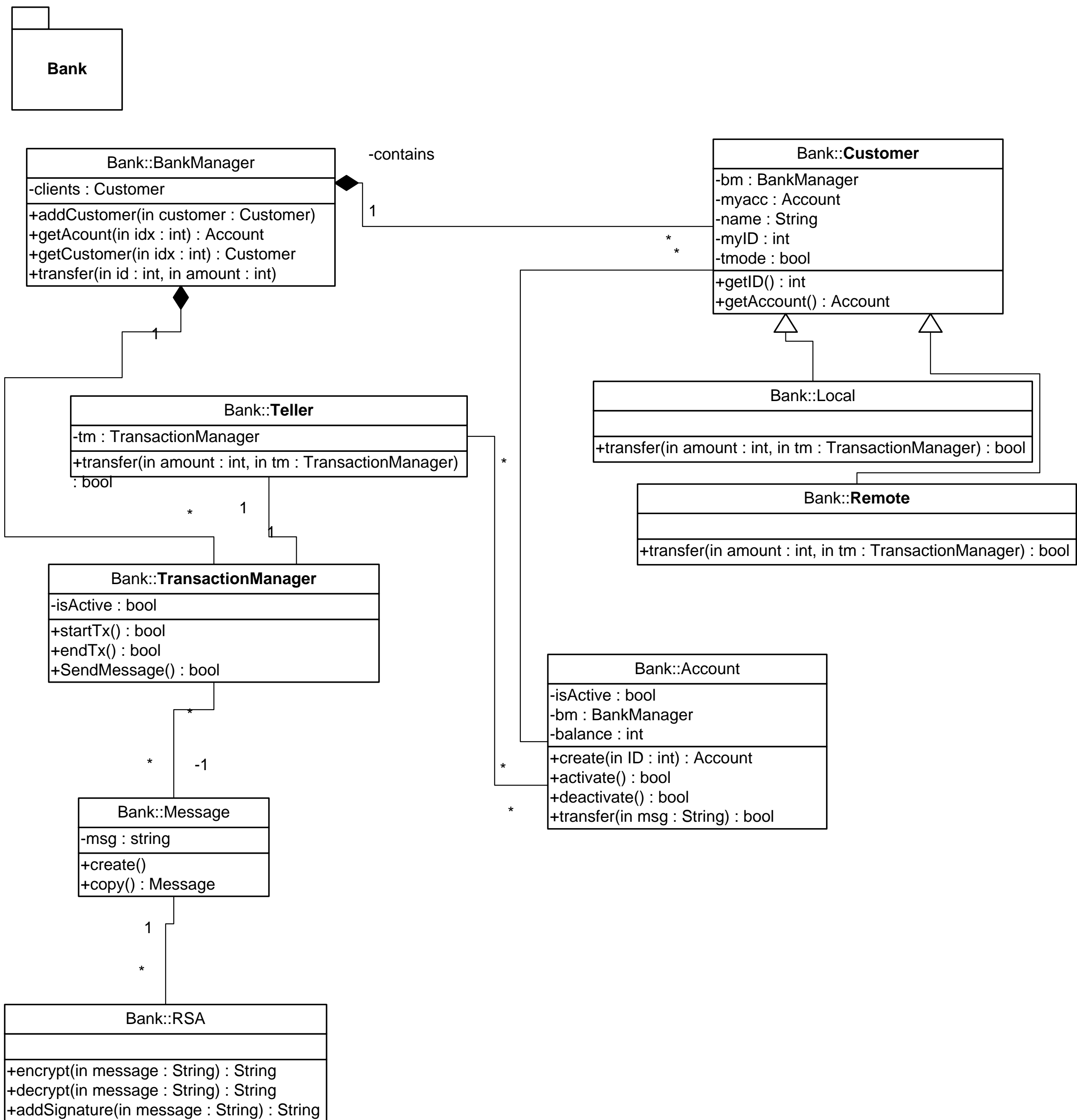
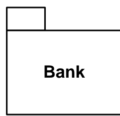


Figure 4.1: Banking System Class Diagram.



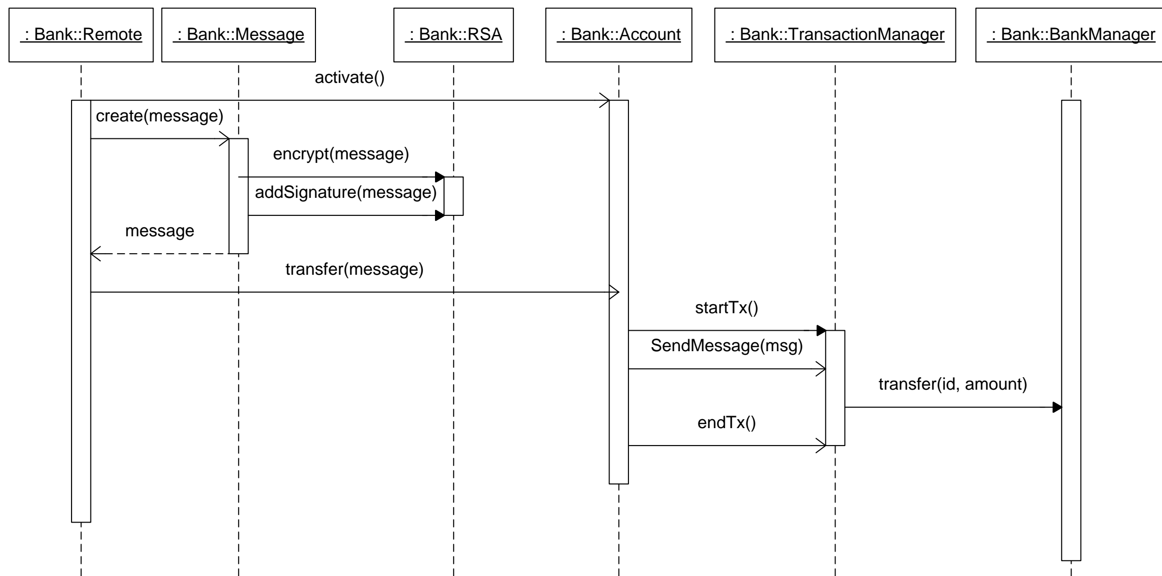


Figure 4.2: Banking System Sequence Diagram.

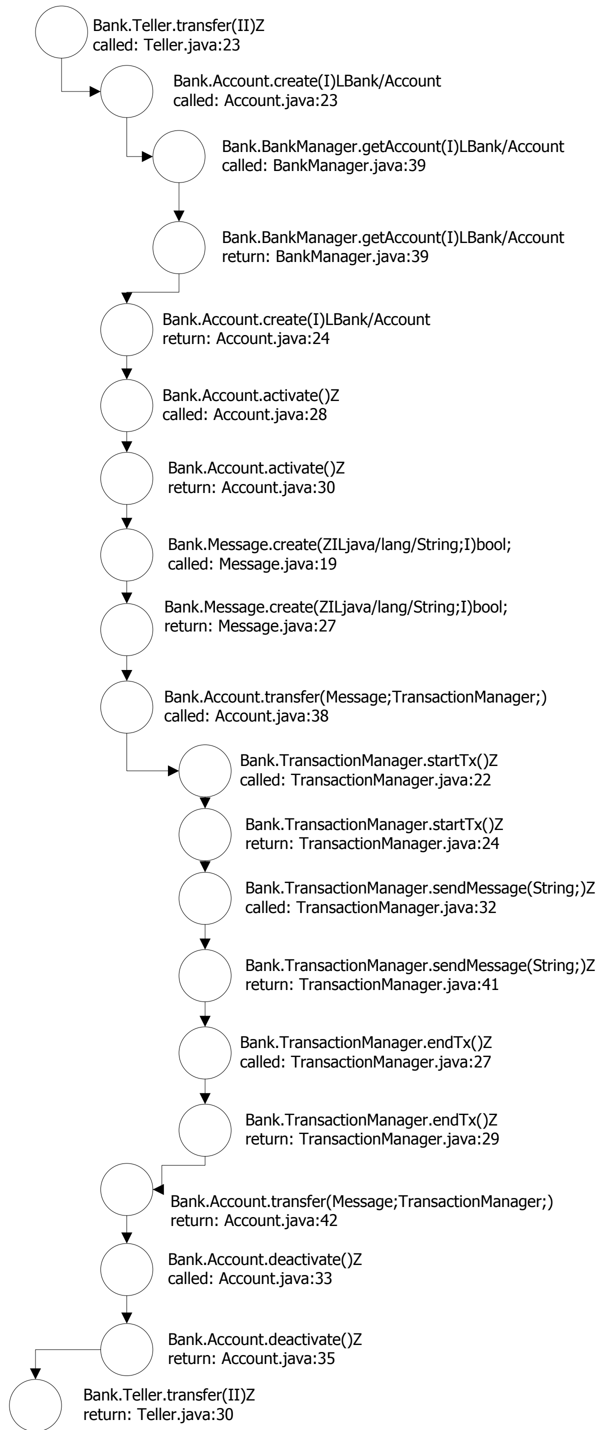


Figure 4.3: Operation Access.

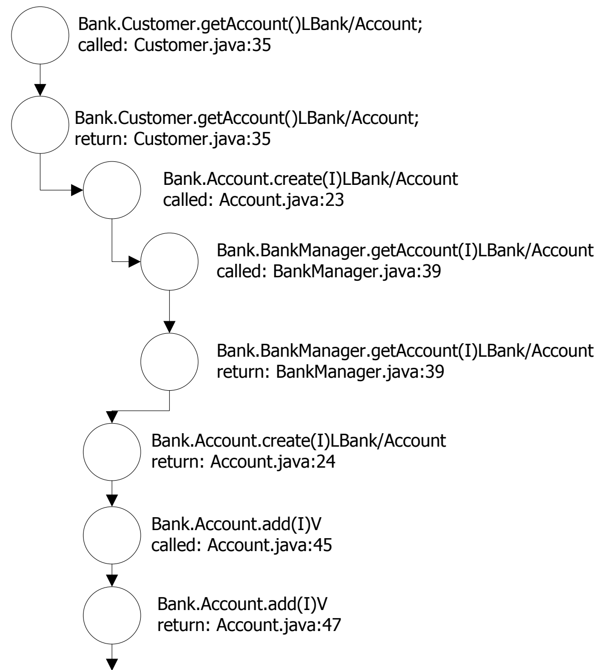


Figure 4.4: Operation Access.

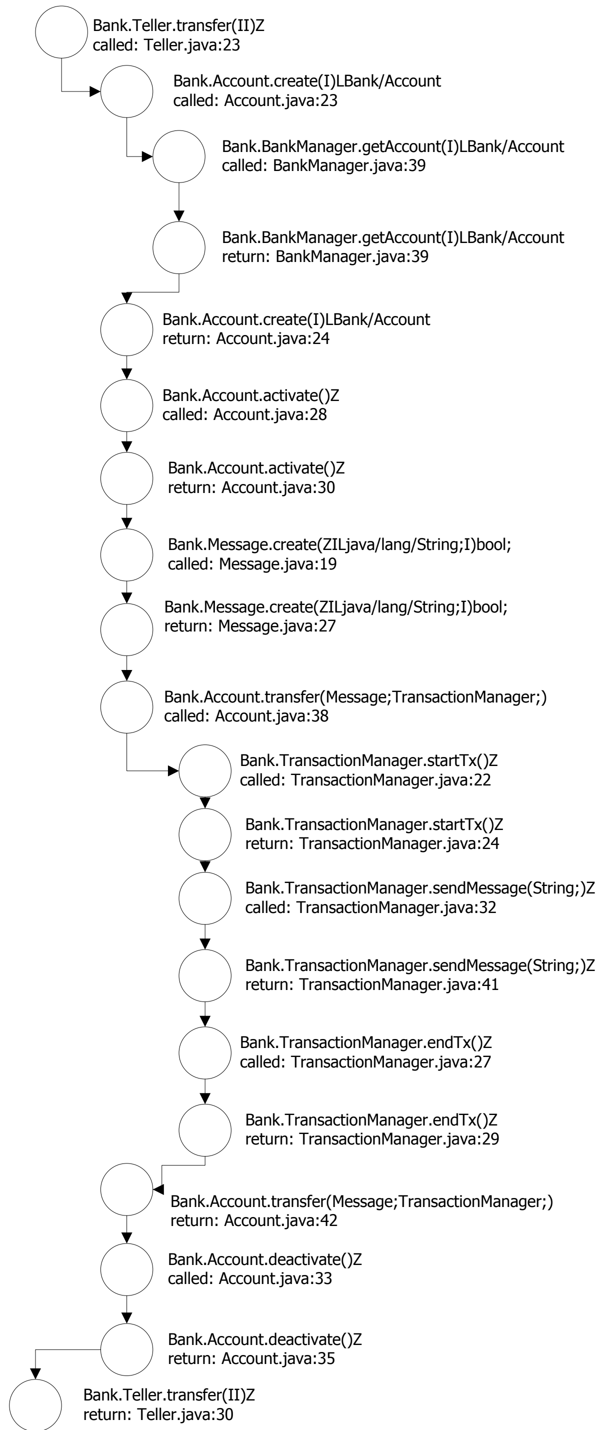


Figure 4.5: Operation Composition and Multiplicity.

Chapter 5

Conclusions and Future Work

This thesis proposes a systematic methodology that can assist software developers improve the quality of their work in the areas of debugging and constraint analysis. This methodoly uses the concept of code instrumentation as the main tool in order to achieve the proposed goals. We used the Java environment to build and test our tool therefore we used bytecode instrumentation. There are many alternatives to bytecode engineering techniques such as monitoring events generate by JVMPI (Java Virtual Machine Profiling Interface) or JVMTI (Java Virtual Machine Tool interface). Both techniques generate events for method entry/exit, object allocation or monitor entry but they present some disadvantages such as they disable VM optimizations and not much flexibility is provided. Other bytecode instrumentation techniques such as ASPECTJ or ASPECTWERKZ are using libraries BCEL or ASM for greater flexibility and simplicity. Our bytecode instrumentation module uses BCEL library and provides two instrumentation points corresponding to methodEntry() and methodExit() calls. The flexibility of the BCEL library allows us to track specific information such as: method additional information (name, parameters types, signature), location (code line number) or runtime information very useful for our reverse engineering module

that will create DAGs very similar in structure to Design Sequence Diagrams.

A very challenging aspect of this thesis was the generation of test cases. Since we use a Java based environment we use JUnit tests to provide coverage for the desired use cases. A developer cannot prove that his code is defect free, however, she can show the absence of faults via good test coverage. The approaches described in this thesis requires increased attention to providing test cases with good coverage in order to confirm the effectiveness of the approaches. However, if you do not have good coverage then you cannot have any conclusions concerning the techniques. For the constraint checking approach we need to generate unit tests that provide coverage for a specific use case so the we can check if a specific constraint is correctly implemented. If you only have one test, then you have a better chance of satisfying a constraint, because you may not have properly covered the code under test. For the code debugging approach it is necessary to generate two sets of unit tests that provide coverage for the code that produces the fault as well as test cases that do not fail. Therefore there is a slight difference in the way the unit tests are generated. Currently the unit tests are manually created for each specific case. For the code debugging approach, a higher number of unit tests will present a higher probability in fault detection. Therefore a large volume of unit tests is highly recommended in order to fully appreciate the proposed approaches. This aspect leads us to the first question we would like to address in the future work: should we attempt to automate the unit test generation module or at least to minimize the developer input? M.Prasanna et al. in [59] present a survey on automatic test case generation. They present several approaches for test case generation such as: random, path oriented, goal oriented and intelligent approaches. They are static and dynamic approaches. We will focus on developing a hybrid intelligent-goal oriented approach that will vary the input parameters in order to test the code in closer proximity to the fault revealing section in a certain number

of iterations.

Other concern regarding this tool is that the final results are displayed in form of Object Method Directed Acyclic Graphs (OMDAGs) and requires the developer interpretation and result reasoning. In most cases it requires the presence of the original Design Sequence Diagrams in order to compare the results. In [54], Pilskalns et al. present an approach for transforming the design Sequence Diagrams into OMDAGs. It would be useful to use this approach in order to simplify the developer results interpretation since it would have to compare similar OMDAGs. Therefore our future work will try to address this approach too.

The actual fault detection and constraint checking tool works as a standalone application inside the IDE NetBeans and requires the developer intervention in establishing the right environment to access the source code to be tested. Our future work will focus on integrating this tool in form of Plugin for one or more of the most popular Java IDE such as NetBeans or Eclipse.

Overall, the positive results of using this tool in several particular projects prooved the efficiency of our approach.

Bibliography

[1] D. Ferrari, M. Liu, "A general-purpose software measurement tool", In *Proceedings of the 1974 ACM SIGMETRICS conference on Measurement and evaluation* , December 1974.

[2] Z. Aral, I. Gerther, G. Schaffer, "Efficient debugging primitives for multiprocessors", In *Proceedings of the third international conference on Architectural support for programming languages and operating systems* , April, 1989.

[3] John C. Munson, "Software Engineering Measurement", In *Auerbach Publications; 1 edition*, pp. 257–262, March 12, 2003.

[4] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", In *Future of Software Engineering*, 2007.

[5] H. Agrawal, J. Horgan, S. London, W. Wong, "Fault Localization using Execution Slices and Dataflow Tests", In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pp. 143–151, 1995.

[6] H. Agrawal, "TOWARDS AUTOMATIC DEBUGGING OF COMPUTER PROGRAMS", *Doctor in Philosophy Thesis*, August, 1991.

[7] J. Eagan, M. J. Harrold, J. Jones, J. Stasko, "Technical note: Visually Encoding Program Test Information to Find Faults in Software", In *Proceedings of IEEE Information Visualization*, pp. 33–36, October 2001.

[8] J. Jones, M. Harrold, J. Stasko, "Visualization of Test Information to Assist Fault Localization", Proceedings of the 24th International Conference on Software Engineering, pp. 467 477, 2002.

[9] R. DeMillo, H. Pan, E. Spafford, "Failure and Fault Analysis For Software Debugging", Proceedings of the Computer Software and Applications Conference, pp. 515 521, 1997.

[10] A. Zeller, "From Automated Testing to Automated Debugging".

[11] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs", In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, November 2002.

[12] G. Misherghi, Z. Su, "HDD: Hierarchical Delta Debugging", In *Proceedings of the 28th international conference on Software engineering*, May 2006.

[13] C. Liu, J. Han, "Failure Proximity: A Fault Localization-Based Approach", In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, November 2006.

[14] A. Podgurski, D. Leon, P. Francis et. al "Automated support for classifying software failure reports", In *ICSE'03*, pp. 465 475, 2003.

[15] C. Yilmaz, C. Williams, "An Automated Model-Based Debugging Approach", In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, November 2007.

[16] T. Huang, P. Chou, C. Tsai, H. Chen, “Automated Fault Localization with Statistically Suspicious Program States”, In *ACM SIGPLAN Notices*, July 2007.

[17] J. Jones, J. Bowring, M. Harrold, “Debugging in Parallel”, In *Proceedings of the 2007 international symposium on Software testing and analysis*, July 2007.

[18] L. Briand, Y. Labiche, and Y. Miao,“ Towards the Reverse Engineering of UML Sequence Diagrams”, *IEEE 10th Working Conference on Reverse Engineering* November 13-17, 2003.

[19] Y. Gueheneuc, “Abstract and Precise Recovery of UML Class Diagram Constituents”, *ICSE'04* pp. 523, September 11-14, 2004.

[20] R. Binder, “ Testing Object-Oriented Systems Models, Patterns, and Tools”, *Object Technology Series*, Addison Wesley, Reading, Massachusetts, 1999.

[21] L. Briand and Y. Labiche, “A UML-based Approach to System Testing”, *4th International Conference on the UML*, pp. 194-208, Oct, 2001.

[22] M. Mustafa and K. Jeffrey, “Efficient Instrumentation for Code Coverage Testing”, in *ACM SIGSOFT Software Engineering Notes*, July, 2002.

[23] M. Factor, A Schuster and K. Shagin, “Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach”, in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* , Oct, 2004.

[24] W. Binder, J. Hulaas and P. Moret, “Advanced Java bytecode instrumentation ”, in *Proceedings of the 5th international symposium on Principles and practice of programming in Java* , September, 2007.

[25] W. Binder and J. Hulaas, “Flexible and efficient measurement of dynamic bytecode metrics”, in *Proceedings of the 5th international conference on Generative programming and component engineering* , Oct, 2006.

[26] P. Saxena, R. Sekar and V. Puranik, “Efficient fine-grained binary instrumentationwith applications to taint-tracking”, in *Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization* , April, 2008.

[27] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou and Y. Wu, “LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks ”, in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* , Dec, 2006.

[28] N. Kumar and R. Peri, “Transparent debugging of dynamically instrumented programs ”, in *ACM SIGARCH Computer Architecture News*, Dec, 2005.

[29] K. Chen and J. Chen, “On instrumenting obfuscated java bytecode with aspectss”, in *Proceedings of the 2006 international workshop on Software engineering for secure systems* , May, 2006.

[30] Pilskalns, Williams, Aracic, Andrews, “Security Consistency in UML Designs”, 30th International Computer Software and Applications Conference (COMPSAC) 2006

[31] Pilskalns, Williams, McDonald, “Generating Design Consistency Constraints”, Technical Report - Washington State University January, 2009

[32] E. Metz, R. Lencevicius and T. Gonzalez, “Performance data collection using a hybrid approach ”, in *Proceedings of the 10th European software engineering*

conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering , Sept, 2005.

[33] T. Moseley, A. Shye, V. Reddi and R. Peri, "Shadow Profiling: Hiding Instrumentation Costs with Parallelism ", in *Proceedings of the International Symposium on Code Generation and Optimization* , March, 2007.

[34] A. Nicoara, G. Alonso and T. Roscoe, "Controlled, systematic, and efficient code replacement for running java programs", in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, April, 2008.

[35] M. Biberstein, V. Sreedhar, B. Mendelson, D. Citron, A. Giammaria , "Instrumenting annotated programs ", in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, June, 2005.

[36] S. Reiss , "Visualizing program execution using user abstractions", in *Proceedings of the 2006 ACM symposium on Software visualization* , Sept, 2006.

[37] T. Wang, A. Roychoudhury "Dynamic slicing on Java bytecode traces ", in *ACM Transactions on Programming Languages and Systems (TOPLAS)* , March, 2008.

[38] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, "Visualizing the Execution of Java Programs", In S. Diehl, Ed., *Software Visualization*, vol. 2269, *Lecture Notes in Computer Science*, Springer verlag, 2002, pp. 151–162.

[39] T. Jacobs and B Musial, "Interactive Visual Debugging with UML", In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pp 115122, 2003.

[40] D. F. Jerding, J. T. Stasko, T. Ball, "Visualizing Interactions in Program Executions", *Proceedings International Conference on Software Engineering*, pp. 360–370, 1997.

[41] R. Kollman, M. Gogolla, "Capturing Dynamic Program Behavior with UML Collaboration Diagrams" *Proceedings CSMR*, pp. 58–67, 2001.

[42] C. Larman, "Applying UML and Patterns, Third Edition", *Prentice Hall*, 2005.

[43] Object Management Group, "UML 2.0 Draft Specification", http://www.omg.org/uml, 2005

[44] R. Oechsle, T. Schmitt; "JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)", In S. Diehl, Ed., *Software Visualization*, vol. 2269, *Lecture Notes in Computer Science*, Springer Verlag, 2002, pp. 176–190.

[45] T. Richner, S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles", In *Proceedings International Conference on Software Maintenance*, pp. 34–43, 2002.

[46] T. Systa, K. Koskimies, H. Muller, "Shimba – An Environment for Reverse Engineering Java Software Systems", *Software – Practice and Experience*, vol. 31(4), pp. 371–394, 2001.

[47] M. Telles and Y. Hsieh, "The Science of Debugging", *The Coriolis Group*, Scottsdale, AZ, 2001.

[48] M. Vans, A. von Mayrhauser, G. Somlo, "Program Understanding Behavior during Corrective Maintenance of Large-scale Software", *Int. Journal Human-Computer Studies*, vol. 51, pp. 31–70, 1999.

[49] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, J. Isaak, "Visualizing Dynamic Software System Information through High-Level Models", In *Proceedings OOPSLA*, pp. 271–283, 1998.

[50] xSlice: A Tool for Program Debugging; http://xsuds.argreenhouse.com/html-man/coverpage.html.

[51] P. Samuel, R. Mall, A.K. Bothra , "Automatic test case generation using unified modeling language (UML) state diagrams", in *IET Software* , pp: 79 - 93, April, 2008.

[52] O. J. Pilskalns, D. Williams, D. Aracic and A. Andrews, "Security Consistency in UML Designs", *30th International Computer Software and Applications Conference (COMPSAC)* 2006.

[53] O. J. Pilskalns, S. Wallace, F. Ilas, "Runtime Debugging Using Reverse Engineered UML", *10th International Conference on Model Driven Engineering Languages and Systems (MODELS)* 2007.

[54] O. J. Pilskalns, A. Andrews, R. France and S. Ghosh, "Rigorous Testing by Merging Structural and Behavioral UML Representations" , in *Proceedings from the 6th International Conference on the UML* October 2003, pp. 234248.

[55] M. Gogolla and M. Richters, "Validating UML Models and OCL Constraints" , in *Proc. 3rd Int. Conf. Unified Modeling Language (UML2000)* March 2000, pp. 265277.

[56] M. Gogolla, J. Bohling and M. Richters, "Validation of UML and OCL Models by Automatic Snapshot Generation" , in *Proceedings from the 6th International Conference on the UML* October 2003, pp. 265279.

[57] K.Wang, W Shen, “Runtime Checking of UML Association-Related Constraints” , in *Proceedings of the 29th International Conference on Software Engineering Workshops* May 2007.

[58] E. Dustin, J. Rashka, and J. Paul, “Automated software testing: introduction, management, and performance” , Boston: Addison-Wesley, 1999.

[59] M.Prasanna, S.N. Sivanandam, R.Venkatesan, R.Sundarrajan, in “A SURVEY ON AUTOMATIC TEST CASE GENERATION” http://www.acadjournal.com/2005/v15/part6/p4/.