AN ABSTRACT OF THE PROJECT OF

Vasanth Williams for the degree of

Master of Science in Computer Science presented on January 2003.

Title: Metacontent Based Techniques for the Regression Testing of Component Based Software: A Case Study

Abstract approved:

[illegible]

Gregg Rothermel

Component based software technologies are viewed as essential for creating the software systems of the future. However, the use of externally provided components has serious drawbacks for a wide range of software engineering activities, often because of a lack of information about the components. One such drawback involves validation of components. To address this problem previous researchers have proposed the notion of *metacontent*. Metacontent describes static and dynamic aspects of a component, and consists of information (*metadata*) about components, and utilities (*metamethods*) for computing and retrieving such information. In this project we implement three new metacontent based techniques that address the problem of validating component based applications after they have been modified (also known as "regression testing"): a code based approach, a specification based approach, and a hybrid approach that uses information both at the code and at the specification level. We present a case study that applies all three techniques to a real component based system.

Metacontent Based Techniques for the Regression Testing of Component Based Software: A Case Study

by

Vasanth Williams

A project report

submitted to

Computer Science Department

Oregon State University

in partial fulfillment of

the requirements for the

degree of

Master of Science

Presented March 31st 2003
Commencement March, 2003

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my major professor, Dr. Rothermel, for his valuable ideas and constructive criticism. The example he set is of a person of great integrity, as a nice person, as an outstanding scientist, and as a patient educator.

I would like to express my sincere gratitude to Dr. P. Tadepalli, and Dr. B. Bose, for serving on my committee.

I am especially thankful to Hyunsook Do for her valuable suggestions, guidance, and helpful contributions throughout the project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Preface

In early 2001, the faculty of Computer Science voted to allow, on a trial basis, group Master's projects, in which two or more Master's students work together under a supervisor or supervisors to complete projects of larger scope than those which could be completed by single individuals. It was decided that such a project could involve a single project report, but that each student contributing to the project would need to have primary responsibility for a sufficient portion of the project, and this responsibility should carry over to preparation of the report, in which each student would have primary responsibility for a substantial and clearly demarcated portion. It was further decided that each student would have an independent defense, in which they presented the work, focusing on their contribution, but also making the relationship of that contribution to the larger project clear.

To my knowledge, this is the second group project completed under these guidelines.

The purpose of this preface is to clarify the individual contributions of the three students involved in the project. The project is primarily one involving construction of infrastructure for experimentation with testing techniques, and performance of experimentation. Each of the three students involved: Soumya Chattopadhyay, Vasanth Williams, and Weiyun Wu, had primary responsibility for one portion of the infrastructure construction and experimentation with one specific testing technique, with shared responsibility for overall production of the infrastructure and consideration of results. Each student also contributed to some degree, however, to the work done by the others, through group meetings and individual consultation.

The following summarizes the allocation of primary responsibilities for completion of the project.

Soumya Chattopadhyay

Infrastructure: UML diagrams for the application, tools for linear independent path generation, TSL specification for the application, development of test suite for the application, collection of traces from the execution of the test suites (component and application), jDejavu.

Methodologies: Code Based testing procedure.

Report sections: Introduction, Background, Code Based subsection of Metadata Based Regression Test Selection Techniques, NanoXML subsection of Subject Infrastructure, Code Based Technique subsection of Case Study, Comparison across Techniques, Conclusions and Future Work, NanoXML package information in Appendix, and Method-wise Selection Information in Appendix.

Vasanth Williams:

Infrastructure: UML diagram for the component, TSL specification for the component, development of test suite for the component, collection of traces from the execution of the test suite (component), Tools for mapping of spectra (`MapTest`) and for test selection (`TestSelection`).

Methodologies: Hybrid Technique procedure.

Report sections: Hybrid subsection of Metadata Based Regression Test Selection

Techniques, Test Suites and Automation subsections of the Subject Infrastructure, Hybrid Technique subsection of Case Study and Comparison across Techniques.

Weiyun Wu:

Infrastructure: UML diagrams for the component classes, TSL specification for the component, development of test suite for the component, the IgnoreNew and IncludeNew comparing tools to select dangerous edges, the mapping tool to map edges and paths, the selection mapping tool to map dangerous edges and paths.

Methodologies: Specification based testing procedure.

Report sections: Specification based subsection of Metadata Based Regression Test Selection Techniques, UML Diagrams subsection of the Subject Infrastructure, Specification Based Technique subsection of Case Study and Comparison across Techniques.

Overall management of the project was conducted by myself (Dr. Rothermel), with Ph.D. student Hyunsook Do providing substantial assistance.

Gregg Rothermel

Chapter 1
INTRODUCTION

Of late there has been tremendous interest in component-based software as an aid in engineering large software systems [9, 15]. Increasingly, software engineers are building systems by integrating externally developed software components with application-specific code. Although this use of components provides many advantages, serious drawbacks in that use are becoming apparent. These drawbacks affect a wide range of software engineering activities. For example, component usage threatens our ability to validate software and assess reliability, complicates maintenance, causes problems in program understanding, and introduces threats to security.

Validation of component based software (and of applications that use components) is a cause for concern. Several researchers have considered validation problems and provided techniques for testing component based software. The problem with these techniques is that they do not address issues of system evolution, yet with most successful systems we spend more time testing them as they evolve, than testing them initially.

Regression testing is the process of retesting components and applications as they evolve. Harrold, et al. [4] suggest several techniques for validating evolving components based on additional information (*metacontent*) provided along with the components. Metacontents describe static and dynamic aspects of a component, can be accessed by the component user, and can be used for various tasks. Metacontents consists of information (*metadata*) about components and utilities (*metamethods*) for computing and retrieving such information. (For simplicity in this report, we use only the term metadata to refer collectively to both metadata and metamethods.)

Although theoretically promising, this metadata based approach has not yet been implemented or empirically studied. This project attempts to remedy this, exploring the application of this metadata based approach to the problem of regression test selection for component based software. We present techniques that use metadata for regression testing of component based software using three different types of strategies: code based regression testing using edge-level regression test selection algorithms; specification based regression testing based on the category-partition method; and a hybrid of the two. We evaluate these techniques through a case study applying them to several versions of an XML parser (NanoXML) and an application program that uses that parser. Our empirical results demonstrate that there can be significant savings in the number of test cases that must be rerun for regression testing when component metadata are available, and thus indicate the potential usefulness of metadata for regression testing.

The main contributions of the project are the following:

1. construction of infrastructure that allows empirical study of the use of metadata for regression test selection using the three techniques
2. a case study that demonstrates the usefulness of metadata in code based regression test selection for a real program.

In the next chapter we give some background on regression testing and test selection in general, and component metadata based regression test selection techniques in particular. Then, in Chapter 3, we describe how the three different techniques used in this study work, and the algorithms they use. Chapter 4 describes the infrastructure that we built to carry out the study, including the test subjects, test suites for the subjects, UML diagrams, and automation. Implementation details of the three techniques are reported in Chapter 5; this chapter presents the actual procedure used to study the techniques, modifications to existing tools required to implement the techniques, and issues that had to be resolved to perform the study, as well as results of the studies. We summarize the results and future work in the last chapter.

Chapter 2

BACKGROUND

2.1 Regression Testing

Regression testing is the verification (of a software system) that newly added features and bug fixes have not created problems in the way of new faults in previously working functions. Hence also called *verification testing*, regression testing is initiated after a programmer has attempted to fix a recognized problem or has added source code to a program that may have inadvertently introduced faults. As such regression testing is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance activity.

2.2 Regression Test Selection

Let P be a procedure or program, let P' be a modified version of P, and let T be a test suite for P. A typical regression test proceeds as follows:

1. Select $T' \subseteq T$, a set of test cases to execute on P'.
2. Test P' with T', establishing P''s correctness with respect to T'.
3. If necessary, create T'', a set of new functional or structural test cases for P'.
4. Test P' with T'', establishing P''s correctness with respect to T''.
5. Create T''', a new test suite and test execution profile for P', from T, T', and T''.

Each of these steps involve important problems. Step 1 involves the *regression test selection problem*: the problem of selecting a subset T' of T with which to test P'. Step 3 involves the *coverage identification problem*: the problem of identifying portions of P' or its specification that require additional testing. Steps 2 and 4 involve the *test suite execution problem*: the problem of efficiently executing test suites and checking test results for correctness. Step 5 involves the *test suite maintenance problem*: the problem of updating and storing test information. In this project we look at the regression test selection problem only.

The simplest regression test "selection" strategy, *retest all*, reruns every test case in the initial test suite. This approach, however, can be prohibitively expensive - rerunning all test cases in the test suite may require an unacceptable amount of time. An alternative approach, *regression test selection*, reruns only a subset of the initial test suite. Of course, this approach is imperfect as well - regression test selection techniques can have substantial costs, and can discard test cases that could reveal faults, possibly reducing fault detection effectiveness. This tradeoff between the time required to select and run test cases and the fault detection ability of the test cases that are run is central to regression test selection. Because there are many ways in which to approach this tradeoff, a variety of test selection techniques have been proposed [3], including the following.

2.2.1 Minimization Techniques

Minimization-based regression test selection techniques [2] attempt to select minimal sets of test cases from *T*, that yield coverage of modified or affected portions of *P*. For instance, they may identify a subset T' of *T* that ensures that every segment that is statically reachable from a modified segment is exercised by at least one test case in T' that also exercises the modified segment.

2.2.2 Dataflow Techniques

Dataflow-coverage-based regression test selection techniques select test cases that exercise data interactions that have been affected by modifications [6]. For example, a technique might require that every definition-use pair that is deleted from *P*, new in *P′*, or modified for *P′* be tested. The technique selects every test case in *T* that, when executed on *P*, exercised deleted or modified definition-use pairs, or executed a statement containing a modified predicate.

2.2.3 Safe Techniques

Most regression test selection techniques - minimization and dataflow techniques among them - are not designed to be *safe*. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program. In contrast, when an explicit set of safety conditions can be satisfied, safe regression test selection techniques guarantee that the selected subset, *T′*, contains all test cases in the original test suite *T* that can reveal faults in *P′*. For example, the technique of Rothermel and Harrold [13] uses control-flow-graph representations of *P* and *P′*, and test execution profiles gathered on *P*, to select every test case in *T* that, when executed on *P*, exercised at least one statement that has been deleted from *P*, or at least one statement that is new in or modified for *P′*. This forms the basis of the Dejavu algorithm used as a regression test selection tool [13].

2.2.4 Ad Hoc / Random Techniques

When time constraints prohibit the use of a retest-all approach, but no test selection tool is available, developers often select test cases based on "hunches", or loose associations of test cases with functionality. One simple approach is to randomly select a predetermined number of test cases from *T*.

2.2.5 Component Metadata based Regression Test Selection Techniques

The techniques presented in the previous section are all (except for the retest all technique) code based, and all defined initially for testing entire imperative programs or procedures. More recently some researchers have turned to regression test selection techniques that use metadata for test selection [4].

To perform regression test selection, we need information about the coverage achieved by the test suite on the original version of the software. We also need information about the changes made to the set of components. These can be combined into *metadata,* which is then provided by the component developer. The presence of metadata lets component developers provide information useful for regression test selection without disclosing the source code of the components they distribute. In particular, only version information, coverage measurement facilities, and information about changes between versions of components need to be provided for the techniques to be applicable.

In this report we look at three component metadata based regression test selection techniques – code based, specification based and hybrid.

Chapter 3
METADATA BASED REGRESSION TEST SELECTION TECHNIQUES

In this chapter we describe the three different metadata based regression test selection methodologies that we investigated – code based, specification based, and hybrid, in turn.

3.1 Code Based

The first metadata based technique we consider for regression test selection is a code based technique. Code based testing techniques select test cases based on a coverage goal expressed in terms of some aspect of the code. There are many entities that can be selected for coverage, such as statements, edges, paths, methods, or classes. Such coverage is usually used as an adequacy criterion for a test suite: the higher the coverage, the more adequate the test suite.

In particular, for edge-coverage techniques, the program is instrumented so that when it executes, it records the relevant edges (in a graph model associated with the program) traversed by each test case in the test suite *T*. With this information, we can associate the relevant edges in *P*, the program under test, with each test case in *T*.

Code based regression test selection techniques construct some representation, such as a control-flow graph, call graph, or class-hierarchy graph, for a program *P*, and record the coverage achieved by the original test suite *T* with respect to some entities in that representation. When a modified version P' of *P* becomes available, these techniques construct the same type of representation for P' that they constructed for *P*. The algorithms then use the representations for *P* and P' and compare them to select the test cases from T for use in testing P', based on (1) differences between the representation for P and P' with respect to the entities considered, and (2) information about which test cases cover the modified entities.

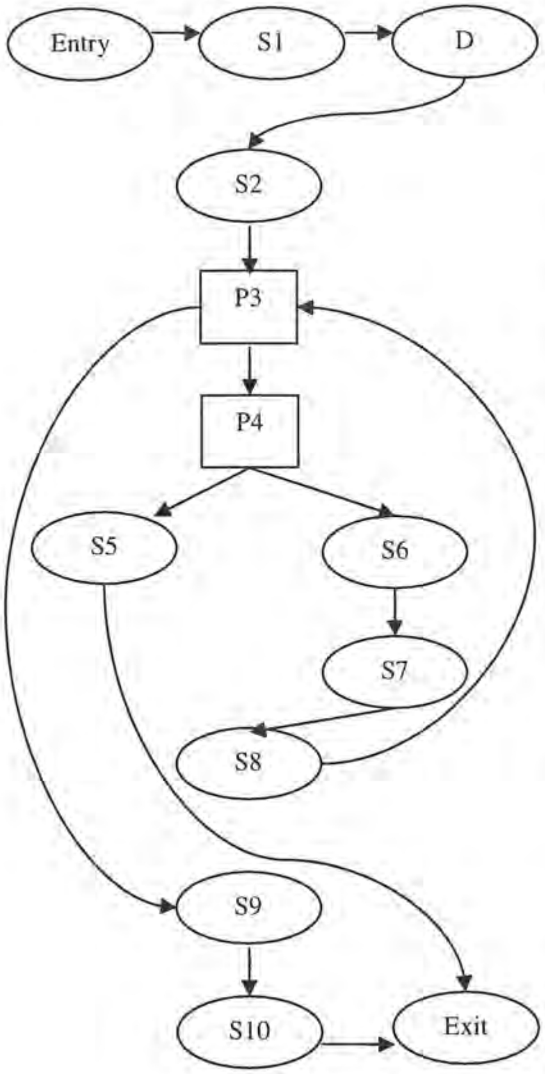
We use Rothermel and Harrold's approach, which is based on a graph-traversal of the representations of the original and modified versions of the software, as a representative of a code based regression test selection technique [13]. In particular, we consider a specific implementation of Rothermel and Harrold's approach: the Dejavu tool.

3.1.1 The Dejavu Technique

Before proceeding further we describe the Dejavu technique. To do this we first describe control flow graphs and instrumentation, on which the technique depends.

3.1.1.1 Control Flow Graphs

A *control flow graph* (CFG) for procedure P contains a node for each simple or conditional statement in P; edges between nodes represent the flow of control between statements. Figure 1 shows procedure `avg` and its CFG.



Procedure avg

```
S1. count = 0

S2. fread (fileptr, n)

P3. while (not EOF) do

P4.   if (n < 0)

S5.     return (error)

      else

S6.     numarray[count] = 0

S7.     count++

      endif

S8.   fread (fileptr, n)

    end while

S9. avg = calcavg (numarray, count)

S10. return (avg)
```

Figure 1: Procedure avg and its CFG

In Figure 1, *statement nodes* (shown as ellipses) represent simple statements. Predicate nodes, shown as rectangles, stand for conditional statements. Labeled edges (*branches*) leaving predicate nodes represent control paths taken when the predicate evaluates to the value of the edge label. Statement and predicate nodes are labeled to indicate the statements in *P* to which they correspond. The figure uses statement numbers as node labels; however, the actual code of the associated statements could also serve as labels. A unique *entry* node and a unique *exit* node represent entry to and exit from P, respectively.

3.1.1.2 Code Instrumentation

Let P be a procedure with CFG G. P can be instrumented such that when the instrumented version of P is executed with test t, it records a branch trace that consists of the branches taken during this execution. This branch trace information can be used to determine which edges in G were traversed when t was executed: an edge (n_1, n_2) in G is traversed by test t if and only if, when P is executed with t, the statements associated with n_1 and n_2 are executed sequentially at least once during the execution. The information thus gathered is called an *edge trace* for t on P.

Given test suite T for P, a *test history* for P with respect to T is constructed by gathering edge trace information for each test in T, and representing it such that for each edge (n_1, n_2) in G, the test history records the tests that traverse (n_1, n_2). Note that the foregoing can be done for each procedure in the program $P'$to obtain a test history for P'.

For convenience, assume the existence of function `TestsOnEdge` (n_1, n_2), that returns a bit vector v of size $|T|$ bits such that the kth bit in v is set if and only if test k in T traversed edge (n_1, n_2) in G.

3.1.1.3 Dejavu Regression Test Selection Algorithm

Dejavu's goal is to identify all non-obsolete tests in T (tests that remain valid despite changes) that execute changed code with respect to P and P'. In other words, the algorithm aims to identify tests in T that (1) execute code that is new or modified for P, or (2) executed code in P that is no longer present in P'. These tests are known as *modification traversing* tests. To identify the modification-traversing tests in T, Dejavu identifies the non-obsolete tests in T that have nonequivalent execution traces (sequences of statements executed) in P and P'.

```
algorithm   SelectTests(P, P′, T):T′
input       P, P′: base and modified versions of a procedure
            T: a test set used to test P
output      T′: the subset of T selected for use in regression testing P′

1. begin
2.     T′ = ∅
3.     construct G and G′, CFGs for P and P′, with entry nodes E and E′
4.     Compare(E, E′)
5.     return T′
6. end

procedure   Compare(N, N′)
input       N and N′: nodes in G and G′

7. begin
8.     mark N as "N′-visited"
9.     for each successor C of N in G do
10.        L = the label on edge (N, C) or ε if (N, C) is unlabled
11.        C′ = the node in G′ such that (N′, C′) has label L
12.        if C is not marked "C′-visited"
13.            if ¬LEquivalent(C, C′)
14.                T′ = T′ ∪ TestsOnEdge(N, C)
15.            else
16.                Compare (C, C′)
17.            endif
18.        endif
19.    endfor
20. end
```

Figure 2: Test Selection Algorithm

Figure 2 presents `SelectTests`, the algorithm that does this. `SelectTests` takes a procedure *P*, its modified version *P′*, and the test suite *T* for *P*, and returns *T′*, a set that contains tests that are modification-traversing for *P* and *P′*. `SelectTests` first initializes *T′* to ∅ and then constructs CFGs *G* (with entry node *E*) and *G′* (with entry node *E′*) for *P* and *P′*, respectively. Next, the algorithm calls `Compare` with *E* and *E′*. `Compare` ultimately places tests that are modification-traversing for *P* and *P′* into *T′*. `SelectTests` returns these tests.

Compare is called with pairs of nodes N and *N′* from *G* and *G′*, respectively, that are reached simultaneously during the algorithm's traversal. Given two such nodes *N* and *N′*, Compare determines whether *N* and *N′* have successors whose labels differ along pairs of identically labeled edges. If *N* and *N′* have successors whose labels differ along some pair of identically labeled edges, tests that traverse the edges are modification-traversing due to changes in the code associated with those successors. In this case Compare selects those tests. If *N* and *N′* have successors whose labels are the same along a pair of identically labeled edges, Compare continues along the edges in G and *G′* by invoking itself on those successors.

Lines 7-20 of Figure 2 describe Compare's actions more precisely. When Compare is called with CFG nodes *N* and *N′*, Compare first marks node *N* "*N′*-visited" (line 8). After Compare has been called once with *N* and *N′* it does not need to consider them again - this marking step lets Compare avoid revisiting pairs of nodes. Next, in the for loop of lines 9-19, Compare considers each control flow successor of *N′*. For each successor *C*, Compare locates the label *L* on the edge from *N* to *C*, then seeks the node *C′* in *G′* such that (*N′*, *C′*) has label *L*; if *(N, C)* is unlabeled, ε is used for the edge label. Next, Compare considers *C* and *C′*. If *C* is marked "*C′*-visited", Compare has already been called with *C* and *C′* so Compare does not take any action with *C* and *C′*. If *C* is not marked "*C′*-visited", Compare calls LEquivalent with *C* and *C′*. The LEquivalent function takes a pair of nodes *N* and *N′* and determines whether the statements *S* and *S′* associated with *N* and *N′* are lexicographically equivalent. If LEquivalent(*C*, *C′*) is false, then tests that traverse edge (*N*, *C*) are modification-traversing for P and P'; `Compare` uses `TestsOnEdge` to identify these tests, and adds them to T'. If `LEquivalent(`C, C'`)` is true, `Compare` invokes itself on C and C' to continue the graph traversals beyond these nodes.

Given a pair of procedures for which CFGs G and G' contain n and n' nodes, respectively, and given a test suite of $|T|$ tests, if `Compare` is called for each pair of nodes (N, N') ($N \varepsilon G$ and $N' \varepsilon G'$), the running time of `SelectTests` is $O(|T|nn')$.

3.1.2 jDejavu

We have adapted Dejavu to handle Java subjects (jDejavu). jDejavu uses a control-flow graph as the representation, and the entities are the edges in the graph. To select test cases to be rerun jDejavu performs a synchronous traversal of the control-flow graph (CFG) for P and the control-flow graph (CFG′) for P', identifies edges modified from CFG to CFG′ or leading to modified nodes, and selects the test cases that cover such edges as the test cases to be rerun.

To achieve better regression test selection when the source code of the component is unavailable, we can use component metadata. To support test selection for code based regression testing, we need three types of metadata for each component. First, we need to know the edge coverage achieved by the test suite with respect to the component so that we can associate test cases with edges. Second, we need to know the component version. Third, we need a way to query the component for the edges affected by changes in the component between two given versions. The component developer can provide this information in the form of metadata and metamethods, and package them with the component.

We then construct a metadata aware version jDejavu$_{MA}$ of jDejavu. This tool builds the matrix "test cases"–"edges covered" by gathering the component coverage data for each test case. Thus, when a new version of a component is acquired, jDejavu$_{MA}$ queries the new version about which edges are affected by the changes and selects test cases to rerun, based on the affected edges and the matrix.

3.2 Specification Based

Specification based testing techniques develop test cases based on functional descriptions of a system, and are considered complementary to code based techniques. Various types of specifications, such as natural language specifications, FSM diagrams, and UML (Unified Modeling Language) are available. Among these, UML is increasingly popular for modeling object-oriented and component-based systems.

Statechart diagrams are one type of diagrams UML provides for modeling the dynamic aspects of systems. These diagrams are useful for modeling the lifetime of an object. They describe the possible states that a particular object can get into and how the object's state changes as a result of events that affect the object. When designing object-oriented systems, a statechart diagram can be drawn for each class to show the lifetime of a single object of that class.

We use an example to illustrate statechart diagrams. We slightly modified the UML statechart specification of class `Dispenser` of the vending machine presented in Harrold et al. [4]. Figure 3 shows the statechart specification of class `Dispenser`. Figure 4 shows the implementation of class `Dispenser`. The dispenser machine has five states: `Empty`, `Insufficient`, `Enabled`, `Initial`, and `Empty`. Among them, three states (Empty, Insufficient, and Enabled) accept two events (setCredit and dispense), and produce two actions (nack and ack). If event setCredit is received in state Empty, then three transitions are possible: the machine stays in state Empty, reaches state Insufficient, or reaches state Enabled based on a credit value which is expressed in a guard condition. If the credit is set to zero, the machine remains in state Empty. If the credit is greater than zero and less than 50, the machine transitions to the Insufficient state because the cost per item is 50. If the credit is equal to or greater than 50, the machine transitions to the Enabled state in which a user may get the item he wants. In states Insufficient and Enabled, event setCredit is consumed without producing any external effect. The credits in these two states originally are not zero; when the setCredit event happens, the credit values in these states do not change, therefore the machine stays in the same state. Event dispense triggers a nack or ack action depending on the availability of the requested item. The machine can take a transition from the Empty state to the Final state automatically, in which case the lifetime of an object in class Dispenser is over.

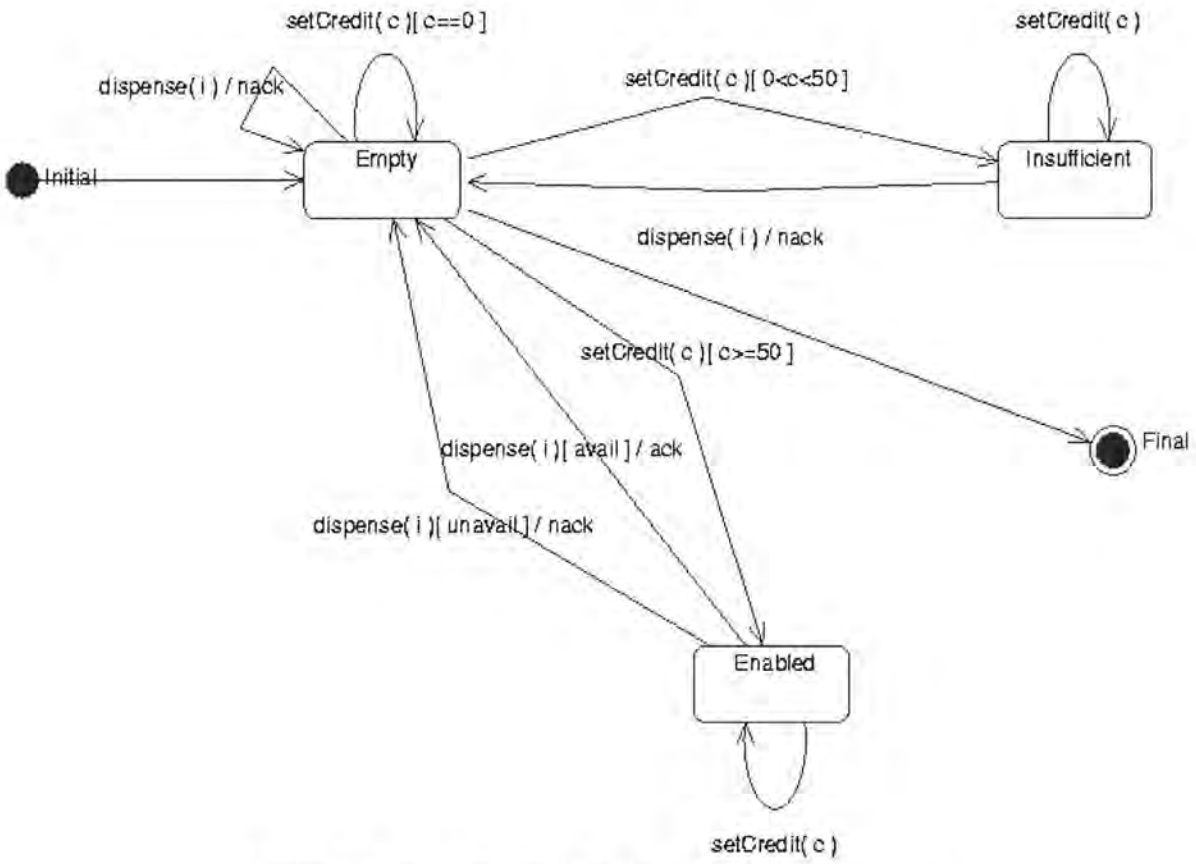


Figure 3: Statechart specification of class `Dispenser`

```
Class Dispenser {
        final private int COINVALUE = 25;
        final private int COST = 50;
        final private int MAXSEL = 4;
        private int credit;
        private int itemsInStock[] = {2, 0, 0, 5, 4};

        public Dispenser() {
                if(credit != 0) system.out.println("Credit already set");
                else Credit  =  nOfCoins * COINVALUE;
        }
        public int dispense( int selection) {
                int val = 0;
                if (selection > MAXSEL) val = -1;   // invalid selection
                else if (itemsInStock[selection] < 1) val = -2;   // selection unavailable
                else if (credit < COST) val = -3;   // Insufficient credit
                else {
                        val = COST;
                        itemsInStock[selection]--;
                }
                credit = 0;
                return val;
        }
}  // class Dispenser
```

Figure 4: Implementation of `Dispenser` in C

3.2.1 Metadata Based Test Selection Using UML Statecharts

Harrold et al. suggest an approach to regression test selection using metadata based on UML statechart diagrams. Their technique, based on one defined by Hartmann et al. [7], constructs a global behavioral model by composing the statechart diagrams for components incrementally using a heuristic reduction algorithm and specific composition rules. This reduction algorithm can reduce the size of a composed statechart by eliminating some unreachable states.

Harrold et al. use this technique to perform specification based test selection for component based software which integrates application-specific code with externally-developed software components. To illustrate the technique, they consider the situation of a component developer who distributes a set of components *C* and a component user who acquires such components to integrate them into application *A*.

Because Harrold et al. technique's requires a specification for the component to be available, it cannot be applied if the component developer provides the component user with the set of components in *C* and no additional information. Therefore, the component developer must provide metamethods that let the component user retrieve, for each component in *C*, a specification expressed as a statechart diagram.

With this information available, the component user can then produce statechart diagrams for application *A* and combine these statechart diagrams with the statechart diagrams for the components in *C* and build a global behavioral model. Given the global behavioral model, the component user can then generate test cases for the application using any testing technique based on state-machine coverage.

Suppose now that the specification for one or more components is changed, and a new version of *C*, *C'*, is released, and the component user must retest *A* with *C'*. In the absence of information about changes between C and C', the component user may need to execute all of the test cases for A that exercise components in C. If information about changes between the specification of C and C' is available, however, the component user can exploit such information to perform regression test selection. To allow this, the set of states and transitions in the statechart diagrams for C that are affected by the changes to C's specification must be attached to C. Again, the component developer can make this information available through metadata: each component must be provided with metamethods that let the component user retrieve information about changes in the specification between two different versions of the component.

We initially attempted to use Harrold et al.'s metadata technique for specification based test selection as defined in [4]. We first produced a statechart diagram for each class in each version of our tested system (Section 4.4 provides details on our process for doing this). We then combined the statechart diagrams of two component classes and a testdriver to obtain the global behavioral model of the classes and the testdriver. This global diagram had approximately one thousand states. We defined a path in a statechart diagram to be a sequence of states and transitions beginning at the start state and ending at the end state. We also placed restrictions on cycles (such as self-transitions) in a path: no cycle was allowed to occur more than once. Because each path represents a "testing requirement", that is, a sequence of events that needs to be exercised by a test case, testers take each path and develop tests for them. Therefore we generated all the paths of this global statechart diagram. The number of paths generated, however, numbered in the thousands.

3.2.2 Adapting Harrold et al.'s Technique

There are at least nine classes in each component version in our tested system. If we combine all the statechart diagrams in one component version with the statechart diagram of the application in order to obtain a global behavior model of the system, there will be a huge number of states and paths generated for the system. We would be required to design a huge number of test cases for the system in one version, which is impractical.

An alternative to considering all paths with less than one cycle is to use the concept of linearly independent paths [12] as testing requirements. A linearly independent path is a path that must move along at least one edge that has not been traversed before this path is defined. The computation of cyclomatic complexity provides the upper bound for the number of linearly independent path. The cyclomatic complexity $V(G) = E - N + 2$, where E is the number of edges, and N is the number of nodes in a diagram. Using linearly independent paths, we can produce fewer paths but could not solve the problem of having too many states in the global diagram. Therefore, we developed an alternative technique for specification based selection.

Our alternative technique, like Harrold et al.'s, requires a specification for each component to be available, represented as a statechart diagram for each component. We do not, however, combine all the statechart diagrams together to form a global diagram; instead we identify testing requirements for each component individually. So we focus on each individual statechart diagram.

To illustrate the approach, let C be a set of components, suppose we have the statechart diagram for one component in C and a set of test cases for that component, and suppose the specification for the component is changed in a new version of C, C'. Without information about changes between the component in C and C', we may have to rerun all the test cases that exercise the component. If we have information about the specification changes made between the two versions of the component, we can use this to perform regression test selection, and choose just the important tests.

We again use the `Dispenser` component example to illustrate the approach. Suppose that after we change the specification of component `Dispenser`, a new version of `Dispenser` is released. Suppose the first change to the specification is: when we set the credit, the new credit value is always the old credit value plus the added credit value, instead of being the old credit value when the old credit is not zero. Suppose that the second change to the specification is: after we dispense an item, instead of always setting credit value to zero, we let the credit value equal the old credit value minus the cost per item if the vending machine vends an item, or let the credit value remain the same as the old credit value if the vending machine does not vend an item. We modified the statechart specification of class `Dispenser` shown in Figure 3 to reflect these changes, resulting in the statechart specification shown in Figure 5.

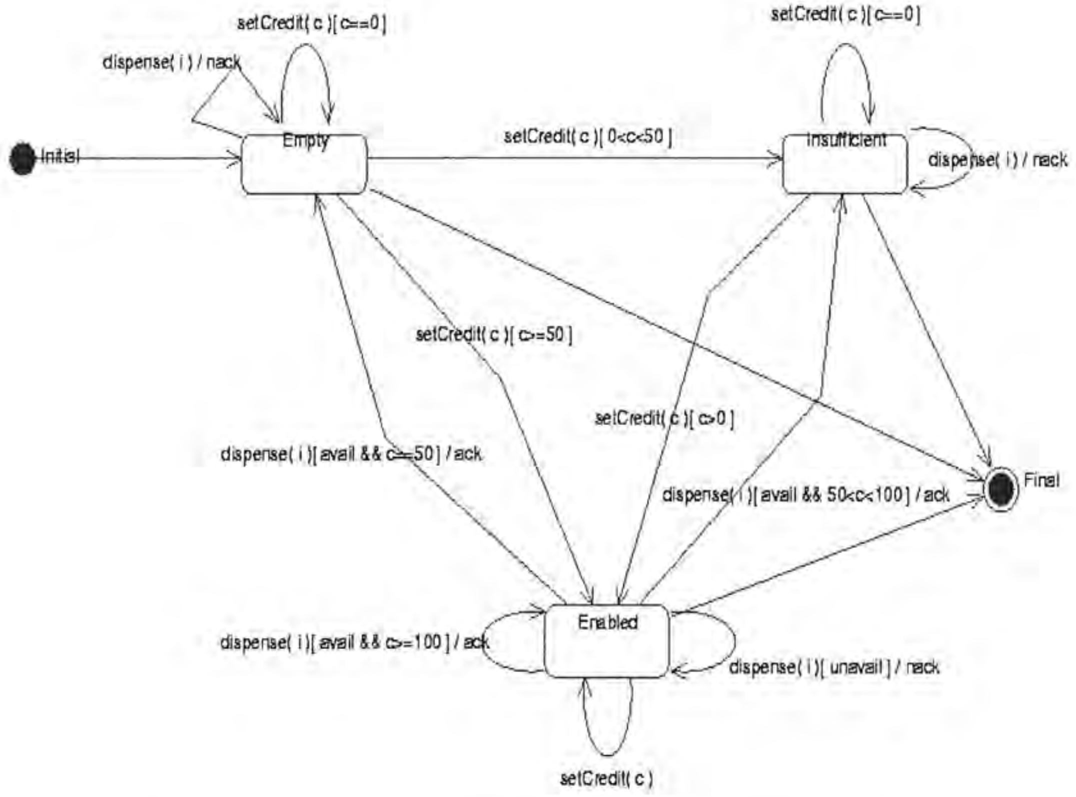


Figure 5: Modified Statechart specification of class Dispenser

The following explains the changes in the statechart specification in Figure 5. When the event `setCredit` occurs in state `Insufficient`, the machine remains in the same state if the added credit value is zero, or the machine transitions to the `Enabled` state if the added credit is greater than zero because the new credit value will be equal to or greater than 50. In the `Insufficient` state, when event dispense occurs, the required item cannot be given and the credit value remains the same, so the state remains the same and the nack action happens. When event dispense occurs in state `Enabled`, there is no state change and the nack action occurs if the required item is not available. If the required item is available, the ack action will occur and the state will remain the same, or transition into the `Empty` state or the `Insufficient` state depending on the value of credit. Finally all three states (`Empty`, `Insufficient`, `Enabled`) can automatically transition into the final state.

Our alternative selection technique assumes that the component developer has created and provided the revised statechart diagram for the component in version *C′*, just as in the example of the component `Dispenser` discussed above. We then compare the two statechart diagrams and identify the set of states and transitions in the statechart diagram of the component in *C* that are affected by the changes to the component's specification in *C′*. Each testing requirement that includes at least one affected state or transition must be retested. These requirements would have to be associated by the application developer with their test cases. Those associated test cases are selected to be rerun.

An important discovery in this work is that we can use the Dejavu selection algorithm presented in Section 3.1.1 to compare two statechart diagrams, except in our situation we need to find ways to handle new edges (transitions) added to the revised graph, and edges deleted in the revised graph. New edges mean that these edges appear in the revised diagram, but do not appear in the previous diagram. We identified two approaches for handling new edges: the first approach is the Include-New-Edge (IncludeNew) approach; the second approach is the Ignore-New-Edge (IgnoreNew) approach. To illustrate, suppose we are comparing node *A* in a statechart diagram with node *A′* in the revised statechart diagram. Suppose there are one or more new outgoing edges from node *A′* (which means these new edges exist in the outgoing edges of node *A′*, but do not exist in the outgoing edges of node *A*). Using the IgnoreNew approach, we ignore the newly added edges of node *A'*; using the IncludeNew approach we select all the incoming edges at node *A* if node *A* is not the successor of the start node. In the case in which node *A* is the successor of the start node, we just ignore the new outgoing edges from node *A*′.

The assumption behind the IncludeNew approach is that the test requirements that reached the source nodes of edges added in the previous diagram may have some effect on the new edges in the revised diagram. Since we don't know which incoming edges in the previous diagram relate to the new outgoing edges in the revised diagram, we select all the incoming edges by the IncludeNew approach. The assumption behind the IgnoreNew approach is that all the test requirements that reach the source nodes of added edges in the previous diagram may have no effect on the new edges in the revised diagram, so we ignore edges reaching source nodes of added edges. In this case, if the new outgoing edges are relevant to the incoming edges, then the IncludeNew approach will be safe but cost more; while the IgnoreNew approach will be unsafe but cost less. If the new outgoing edges are not relevant to the incoming edges, the IgnoreNew approach may be more effective and inexpensive; while the IncludeNew approach may be more ineffective and costly.

Handling deleted edges is simple. Deleted edges are edges which appear in the previous diagram, but do not appear in the revised diagram. If there is a deleted outgoing edge for node *A*′(which means this deleted edge appears in the outgoing edges of node *A*, but does not appear in the outgoing edges of node *A*′), then we select this deleted edge in the original diagram in both approaches.

We provide an example of our specification based selection approach by using the `Dispenser` component. The specification for this component in *C* is the statechart diagram in Figure 3, the specification for this component in $C'$is the statechart diagram in Figure 5. Based on the diagram in Figure 3, we can generate four linearly independent paths, as shown in Table I. Following the modified Dejavu selection algorithm, we compare Figure 3 and Figure 5 and identify the set of states and transitions in Figure 3 which are affected by the changes in Figure 4. In this case, the difference between component `Dispenser` in C and in C' affects the following transitions in C: (1) `setCredit(c)[0<c<50]` from the `Empty` state to the `Insufficient` state, (2) `setCredit(c)` from the `Insufficient` state to the `Insufficient` state, (3) `setCredit(c)[c>=50]` from the `Empty` state to the `Enabled` state, and (4) `setCredit(c)` from the `Enabled` state to the `Enabled` state. We then consider these four affected transitions relative to the paths in Table 1. Paths 2, 3 and 4 go through at least one of the four affected transitions, so these three paths are selected, thus the testing requirements which correspond to those selected paths must be retested, and the test cases connected to those testing requirements should be rerun.

Table 1: Paths for `Dispenser` in C

Path#	Path
1	Initial:Empty, Empty:F
2	Initial:Empty, Empty:Insufficient:[0<c<50]setCredit(c), Insufficient:Insufficient:setCredit(c), Insufficient:Empty:dispense(i)/nack Empty:Empty:dispense(i)/nack, Empty:Empty:[c==0]setCredit(c), Empty:F
3	Initial:Empty, Empty:Enabled:[c>=50]setCredit(c), Enabled:Enabled:setCredit(c), Enabled:Empty:[unavail]dispense(i)/nack Empty:Empty:dispense(i)/nack, Empty:Empty:[c==0]setCredit(c) Empty:F
4	Initial:Empty, Empty:Enabled:[c>=50]setCredit(c), Enabled:Enabled:setCredit(c), Enabled:Empty:[avail]dispense(i)/ack, Empty:Empty:dispense(i)/nack, Empty:Empty:[c==0]setCredit(c), Empty:F

3.3 Hybrid

The hybrid method of test selection is so called because it uses information both at the code level as well as at the specification level. One hybrid technique is described in [10]. In this technique, the component developer applies a specification–based testing technique to the component, but test coverage is calculated on the code.

Specification based testing techniques develop test cases based on functional descriptions of a system. One technique that can be used is the category-partition method, described in [11].

3.3.1 The Category Partition Method

The category partition method requires us first to decompose the specification for a program under test into functional units to be tested independently. Then, for each functional unit, we identify the parameters and the environmental conditions that can affect its behavior. Next, these parameters and the environmental conditions have to be divided further into choices that characterize them. The parameters, conditions and choices are then expressed in a language. The language for expressing the specification in this way is known as TSL (test specification language). Given a test specification in TSL, a tester can produce a set of test frames for each unit by computing the cross-product of the different choices. Each test frame corresponds to a test case that must be developed. Constraints can be added to choices to reduce and control the combinations. Constraints help reduce the number of test frames developed to some affordable number.

The category partition method can be used on applications just as described above. But to adapt the method for use with a component library we had to decide on additional conditions. Consider a system like the one shown in Figure 6.

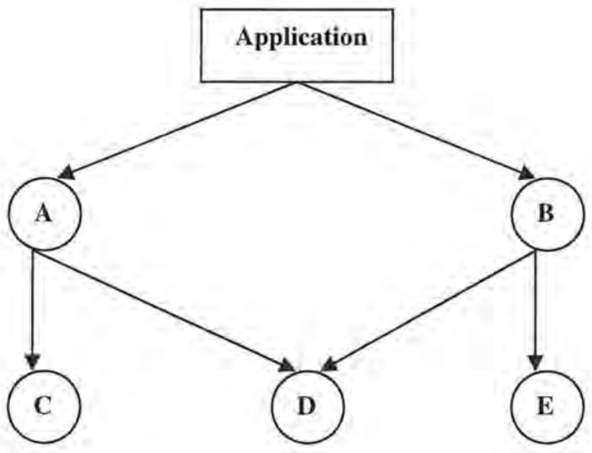


Figure 6: Application and Component Interaction

In this system, we have five methods in a component, namely *A*, *B*, *C*, *D* and *E*, but only methods *A* and *B* are available as interfaces to an application. Methods *C*, *D* and *E* are called by other component methods. When we test a component such as this, we divide the component into only two functional units based on the interfaces made available to the application by the component. The interfaces made available to the application are taken as functional units in the category partition method. The parameters and the environmental conditions are identified specific to these functional units. Then, as described above, these are expressed using TSL's test specification language.

The example of a simple `find` method which can be found in a component library, based on an example given in [11], will illustrate the category partition method. The `find` routine takes in a pattern and a file name as parameters, and returns true or false if the pattern is present in the file or not.

Syntax:

```
bool find (String, File) throws FileNotFoundException
```

Function:

The `find` routine will take in a string and a file name as parameter and output true or false if the string is present in the file or not. The method will throw a `FileNotFoundException` if the file in the parameter is not found.

A TSL test specification for the `find` method in the component could look like the following with constraints.

```
Parameters:
        Pattern size:
                Empty                               [property Empty]
                Single character                    [property NonEmpty]
                Many character                      [property NonEmpty]
                Longer than any line in the file    [error]
        Embedded blanks:
                No embedded blank                   [if NonEmpty]
                One embedded blank                  [if NonEmpty and Quoted]
                Several embedded quotes             [if NonEmpty and Quoted]
        File Name:
                Good file name
                No file with this name              [error]
                Omitted                             [error]

Environments:
        Number of occurrences of pattern in file:
                None                                [if NonEmpty]
                Exactly one                         [if NonEmpty] [property Match]
                More than one                       [if NonEmpty] [property Match]
        Pattern occurrences:
                One                                 [if Match]
                More than one                       [if Match]
```

3.3.2 Metadata based Selection using TSL

Suppose that we test our component using TSL. When we modify our component we change our TSL. We want to tell the application users which of their tests involve changes in our TSL specifications. This is different than state based testing which works from a functional specification of the system, because here we work from a test specification. But this is a strength of this approach, because it might be complementary to other approaches and might work well in practical cases when other specifications are not available. The key to making this approach work is to be able to let the application user obtain a mapping between their test cases and the component developer's TSL specification.

Orso et al. [10] present a technique in which the tester of a component uses the category-partition method for testing the component just as described above. According to Orso et al., if the TSL specifications and test frames are at an appropriate level of abstraction, it should be possible to map code coverage to test frame coverage. Under this approach, a test case in test suite *T* is said to cover a test frame *tf* if (1) the parameters of calls to a single functionality match the corresponding choices in *tf*, and (2) the state of the component matches the environmental characteristics in *tf*.

To compute the coverage of the component achieved by a given test case in terms of test frames, the code has to instrument according to test frames. Now, when performing regression testing of the component, if the user of the component knows which frames are affected, they only need to rerun test cases associated with those frames. This way, when the component version changes, the component developer can inform the user (through metadata) of the test frames that are affected and the user needs to run only the test cases in the test suite that correspond to those test frames.

3.3.3 Adapting Orso's Technique

In this research, we began by attempting to apply Orso et al.'s technique as initially presented. The approach suggested by Orso et al. proved impractical for our subject, however, because we could not instrument the code to directly report execution of test frames. The main reason for this is that, for our program, the parameters were instances of classes and this made instrumentation of the code very difficult and complex. For example, if a method *A* of the component takes an object which is an instance of a class *B* as a parameter, it becomes difficult to track the environmental state of this object. We have to know this environmental state in order to instrument the code with relevant test frame information. Therefore we had to develop a different way to do the mapping.

We considered a different approach to this problem. Our idea was to map the spectra of the tests, run on the application, to the spectra of the tests associated with the test frames of the components. A program spectrum as described in [5] characterizes, or provides a signature of, a program's behavior. The path spectrum which we use to map the tests records the complete path that is traversed by the program. The ways to get these spectra are either by collecting the trace of methods hit or the trace of the basic blocks hit for each test that is executed.

In this modified strategy, a component provider obtains the traces for the test cases developed for the test frames by executing them on the component individually. These are provided as metadata to the application developer. The application developer collects the traces for their testing of the application that uses the component. These traces report the functions and basic blocks hit, in the component, during the execution of the test cases. The traces of the test cases of the component and the application can be compared in terms of trace equivalence. This provides a mapping between the test frames of the component and the test cases of the application. Through this approach, when a new version of the component is released, the application developer can hope to approximate the frames that are affected. Since these frames correspond to the test cases that were mapped by the application developer to the application test cases, the application developer can obtain the test cases of the application that need to be rerun to test the component. We used this modified hybrid technique on our subject to compare the results with other techniques.

Chapter 4

SUBJECT INFRASTRUCTURE

To examine the techniques just described, we need to study them empirically. This required that we obtain infrastructure to support such study, including a component library, application that uses it, state diagrams and tests.

4.1 NanoXML

As a component library we selected NanoXML. NanoXML is a small XML parser for Java, written by Marc De Scheemaecker.[1] The extensible markup language, XML, is a way to mark up text in a structured document. It is designed to improve the functionality of the web by providing more flexible and adaptable information identification. XML is considered extensible because it is not a fixed format like HTML (a single, predefined markup language). Instead, XML is actually a 'metalanguage', a language for describing other languages, which lets people design their own customized markup languages for limitless different types of documents.

We looked for a relatively small program that had a sizeable number of classes (15+) and was written in Java. NanoXML turned out be the ideal candidate. It is a very easy to use, non-GUI based, freely available, and also buildable from its source without any other external libraries.

Figure 7 gives a high-level representation of the major NanoXML components.

[1] Available at http://nanoxml.sourceforge.net/orig/

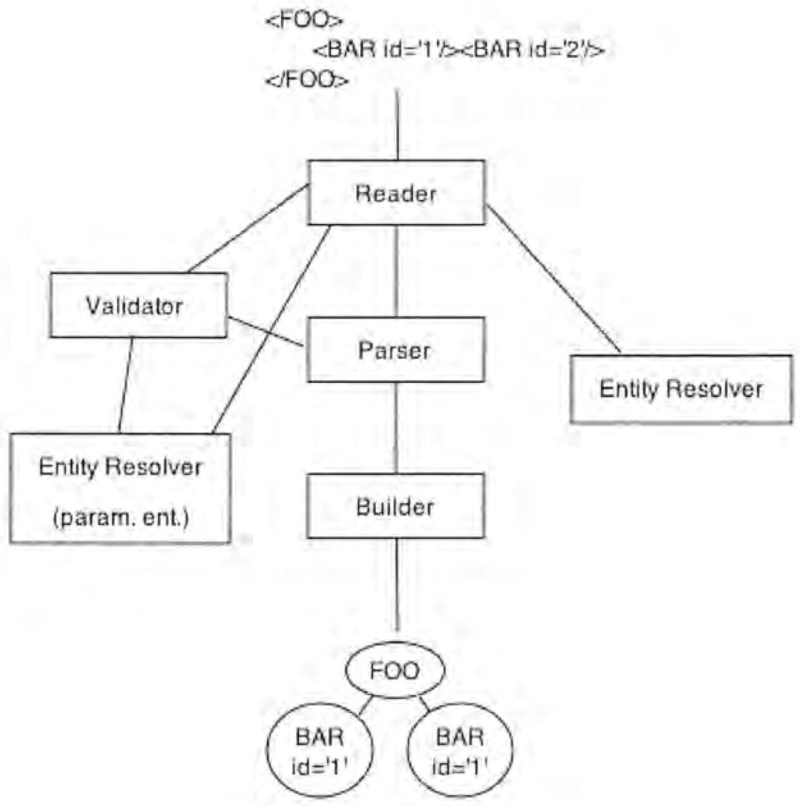


Figure 7: NanoXML Components

The components shown in the figure can be described as follows:

- The *reader* retrieves data from a Java input stream and provides character data to the other components.
- The *parser* converts the character data it retrieves from the reader to XML events which it sends to the builder.
- The *validator* parses a DTD (document type definition) and validates the XML data. The current validator does only the minimum necessary for a non-validating parser.
- The *entity resolver* converts entity references (&...;) and parameter entity references (%...;) to character data. The resolver uses the reader to access external entities.

- The *builder* interprets XML events coming from the parser and builds a tree of XML elements. The standard builder creates a tree of IXMLElement. The user can provide their own builder to create a custom tree or if they are interested in the XML events themselves, e.g. to use XML streaming. [16]

Table 2 provides data on components of NanoXML.

Table 2: Number of Component Classes and Methods

	Versions					
	0	1	2	3	4	5
Total number of classes	9	12	14	15	15	17
Total number of methods	103	120	177	216	217	198

We also needed an application program that would use the components of NanoXML, be compatible with all the versions of NanoXML, and be easy to use and understandable. JXML2SQL (available with NanoXML) is such an application, it takes as input an XML file and either transforms it into an HTML file (showing the contents in tabular form) or an SQL file. There are two versions of the application because we modified the application JXML2SQLApp to work for version v0 of the component and this version of the application was called version v0, and the version that runs with the rest of the versions of the component was called version v1. Table 3 provides data about the applications:

Table 3: Number of Application Classes and Methods

	App1	App2
Total number of classes	9	9
Total number of methods	36	36

4.2 Test Suites

Once we had the six versions of NanoXML, our component subject under consideration, and the two versions of JXML2SQLApp, the application that uses the NanoXML component, we next needed to build test suites for each component version and for the two application versions. There are two versions of the application because we modified the application JXML2SQLApp to work for version v0 of the component and this version of the application was called version v0, and the version that runs with the rest of the versions of the component was called version v1.We decided to build the test suites based on the category-partition method [11] described in section 3.3. By developing this uniform and concise test specification, we could easily add, delete and modify specifications across the versions.

4.2.1 Component Test Suite

We first developed the TSL specification for version v0 of NanoXML. We divided the component into functional units. As functional units, we followed the process described in Section 3.3 and considered each entry point in the program which was available to the user of the component.

We wrote a TSL specification for each functional unit. These test specifications generated a total of 214 test frames after adding constraints. For each of the test frames, we created input files specific to each test frame. The input files were either an .xml file or a .dtd file or both. After developing input files for the 214 test frames, we wrote test drivers to run these tests. The test drivers were written in Java for each of the test frames. With this the infrastructure for the test suite for version v0 was complete.

We wished to simulate a process in which a test suite is progressively refined across system versions. Thus, the test specification for each of the subsequent version was written based on the specification of its preceding version. We modified some parts of the specification to fit the newer versions as the old specifications had become obsolete. For some functional units which had become obsolete, we modified the specifications or added new ones. However, no new functional units were added beyond those existing in version v0, that is, we did not consider new functionality of the upper versions. The test drivers also needed some changes as they were no longer compatible with the newer versions. After the necessary changes, the infrastructure for the test suites was ready for versions from v0 to v5. This gave us 214 test cases for versions v0 and v1. Two more tests were added to v2 because of the increase in parameters in one of the functional units, which increased the number of choices and test frames. Thus, version v2 – v5 had 216 test frames.

4.2.2 Application Test Suite

We developed an application test suite based on the category-partition method by following the same process which we just described in Section 4.2.1. The test specification we developed was the same for both versions of the application as there was no functional change in the application. Unlike the test suite for the component, the application test suite did not require any test drivers since the application itself acted as the driver for its tests. However, we developed input files for each of the test frames. As in the case of the components, the input files were either `.xml` files or `.dtd` files or both for each test frame. This process resulted in creation of a suite of 28 tests for each version of the application.

4.3 Automation

The foregoing process requires us to generate test cases from the TSL specs. This generation was automated with the use of a tool called `tsl`. The `tsl` tool takes in as input a TSL specification and generates the corresponding test frames for the specifications. For example, given a specification file called `spec.tsl`, when we type the command

```
        tsl -c spec.tsl -o spec.frame
```

a `spec.frame` file gets created containing the test frames generated for the specification file `spec.tsl`. Using `tsl` we were able to generate all the test frames for the component and application specification files.

Having created the test suites for both the component and the application, we next needed to automate the execution of the tests. To do this, we first converted the test frames into test cases and described them using a test description language. This test description language was developed previously at OSU for use with C programs, and we modified it slightly to work with Java programs. A tool from the Aristotle toolset (`Javamts`) was used to generate executable test scripts from each test description file. These scripts could be used to automatically run all tests.

In addition to being able to run all tests, we also needed to be able to collect traces for these tests. To do this, we used the Galileo instrumentor, (which instruments Java class files in order to get traces when the Java program is run), developed at OSU, to instrument the NanoXML class files. Once the class files had been instrumented, the scripts were run over the NanoXML components. We then collected traces for the run of each test. We separately collected traces indicating the functions hit, the blocks hits and the sequence of functions hits. The function and sequential function traces each list the functions that were called in the execution of the test, with the difference that the sequential trace lists the order in which the functions were called. There could be duplicates in a sequential trace as well, since a single function could be called many times. The blocks hit trace gives a list of basic blocks that were called during the execution. All the traces and the outputs were stored for further use, such as test selection.

4.4 UML Diagrams

The specification based testing technique presented in Section 3.2 requires us to have a statechart diagram for each component class in each version. We do not have text specifications for any component class in NanoXML, so we performed reverse engineering (the creation of a model from code) to create statechart diagrams [1]. To do this, we first examined the code of a component class thoroughly, chose the start and final states for an object of the class, and decided on the states of the object by considering the conditions in which the object may exist for some identifiable period of time. For example, in the `Dispenser` example there are three stable states for the `Dispenser` component. (The choice of what constitutes a meaningful state is left to the diagrams' designer).

Next, we decided on a meaningful partial ordering of stable states over the lifetime of the object. For example, in the `Dispenser` example, starting from state `Empty`, a transition can go to state `Insufficient` or state `Enabled`, and state `Empty` can transition to state `Final`.

Next, we decided on the events that may trigger a transition from state to state. An event is usually a method call, or sometimes an event can combine with boolean expressions expressed as guard conditions, as with the `setCredit` and `dispense` events in the `Dispenser` example. We may also attach actions to some transitions. Actions occur quickly and are not interruptible, like the `ack` and `nack` actions in the `Dispenser` example. An action can also be a method call.

Next, we checked that all states were reachable under some combination of events, and checked that no state except the final state was a dead end from which no combination of events would transition the object out of that state.

Finally, we traced through the state machine manually, checking it against expected sequences of events and their responses to make sure that the state machine was correct.

By following this procedure, we created nine statechart diagrams for NanoXML components in version 0, and twelve, fourteen, fifteen, fifteen and seventeen diagrams for versions 1 through 5, respectively.

Though constructing statechart diagrams through reverse engineering was not simple, but Java being object-oriented and NanoXML's classes and methods being well defined based on the object-oriented design concepts, it was relatively easy to identify states and transitions in the state diagrams.

Chapter 5
CASE STUDY

We applied the three testing techniques described in Chapter 3 to our test subject (NanoXML) using the infrastructure we built. This section describes the procedure and results for each technique and then compares them over all the component version pairs.

5.1 Code Based Technique

5.1.1 jDejavu

We implemented the code based component metadata based regression test selection technique using a modified version of Dejavu, and some instrumentation tools from the Galileo system. `gCFG`, `gInstrumentor` and `gFilter` are the tools to make control-flow files, instrumented class files, and trace files respectively. A host of other helper programs were used in the implementation. For example, many viewers (some from the BCEL library, some from Galileo) were used to view the byte source (`ByteSourceViewer.class`), the map files (`MapViewer.class`), the cf files (`CFViewer.class`), and the trace files (`TraceViewer.class`). These viewers present the files in human readable format. In addition there were handlers (`TraceHandler.class` etc.), instrumentors (`Probe.class` etc.), and control flow graph generators (`Graph.class`, `Node.class` etc.). These were all used in the first phase of the procedure to obtain instrumented class files, and source files.

The second phase of the work involved obtaining the trace files and selecting the tests – this was done by jDejavu. jDejavu is a modified version of Dejavu; most of the structure is the same (the test selection algorithm in particular), but the rest was adapted to work with Java subjects. jDejavu uses the intra-procedural test selection algorithm, described in Section 3, on Java files. In Dejavu, the source file is the actual `.c` file, but since in Java we worked with multiple source files, the 'source code' for Java files was not the `.java` files themselves but a concatenation of the `.class` files as obtained using the Byte-Source Viewer. This was then used to compare code segments from different versions. jDejavu consists of the core files: `dvu.h`, `dvu1.c`, `dvu_compare.c`, `dvu_corresp.c`, `dvu_output.c`, and `dvu_util.c`. `dvu_corresp.c` compares the Java byte code files lexicographically. Test *t* is modification traversing for *P* and *P′* if and only if the compared segments are nonequivalent.

In jDejavu, nodes have attached tests as opposed to edges having tests linked to them (in Dejavu). `dvu_util.c` selects tests based on the nodes.

5.1.2 Procedure

The code based technique is executed as follows (for version *P*):

1. Make .class executables from the `.java` source code.
2. Make viewable byte code files (also `.class`), from the .class executables (using the Byte-Source Viewer from the BCEL library).
3. Concatenate the byte code files into a single source file for one version (*P*).
4. Make the `*.cf` and `*.map` files using the `gCFG` builders. These files contain the control flow and block information for *P*.
5. Instrument the source files with `gInstrumentor`.
6. Build traces using the `gFilter` and drivers.
7. Build the test history file using `th_builder`.

8. Use jDejavu to select tests from the test history of version *P*, which should be used to test version *P'*.

The above is the procedure for both application test selection and component test selection. The only difference being, in case of the application test selection, in Step 7 the application itself was the driver, and for component test selection separate drivers were written.

5.1.3 Results

5.1.3.1 Application Test Selection

Table 4 shows the number of tests selected from the application test suite using the code based technique. Except in one case (v1-v2), very few of the application tests needed to be rerun when the version of the component changed.

Table 4: Application Test Selection

Versions	Code based
v0-v1	6/28 (21.43%)
v1-v2	28/28 (100.00%)
v2-v3	6/28 (21.43%)
v3-v4	0/28 (0.00%)
v4-v5	6/28 (21.43%)

One observation from Table 4 is that there are no tests selected from v3 to v4. This occurred because there is very little code changed between component versions v3 and v4. In fact the only difference is that v4 has *one* extra method (the `insertChild` method in class `XMLElement`).

Another interesting result from Table 4 is that from v1 to v2 all the tests were selected. Table 5 shows us that `XMLParserFactory` is the class causing selection of all the tests. To investigate this further we look at each method of this class. This helps us identify methods being hit the most; code changes in these methods would cause high selection rate of tests. In particular, we want to look at each method (*M*) of each class to see how many tests (*T*) are selected when we go from an old version of the program (*P*) to a new version (*P′*). The following could be possible:

1. *M* is not in *P′*
    - *T* hits *M* – select *T*
    - no *T* hits *M* – nothing to select
2. *M* is in *P′*
    - *M* has code changes
        - *T* hits M
            - *T* passes through the code changes – select them
            - *T* does not pass through code changes – nothing to select
        - no *T* hits *M* – nothing to select
    - *M* does not have code changes
        - *T* hits *M* – nothing to select
        - no *T* hits *M* – nothing to select

Table 5: Application Test Selection (Class-wise)

Component Classes	v0 – v1	v1 –v2	v2 – v3	v3 – v4	v4 – v5
`NonValidator`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`StdXMLBuilder`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`StdXMLParser`	6 (21.43%)	6 (21.43%)	6 (21.43%)	0 (0%)	6 (21.43%)
`StdXMLReader`	6 (21.43%)	4 (14.29%)	4 (14.29%)	0 (0%)	6 (21.43%)
`XMLElement`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`XMLParseException`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`XMLParserFactory`	0 (0%)	28 (100.00%)	0 (0%)	0 (0%)	0 (0%)
`XMLValidationException`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`XMLWriter`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`ContentReader`	-	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`XMLEntityResolver`	-	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`XMLUtil`	-	5 (17.86%)	0 (0%)	0 (0%)	5 (17.86%)
`ValidatorPlugin`	-	-	0 (0%)	0 (0%)	0 (0%)
`XMLException`	-	-	5 (17.86%)	0 (0%)	0 (0%)
`XMLAttribute`	-	-	-	0 (0%)	0 (0%)

The following tables show the selection data for each method for two typical classes. An N entry means that the method is new in *P′*. The fraction shows the number of tests selected and the number of tests that hit the method. In these cases all the tests that reached a changed method were selected.

Table 6: class `XMLParserFactory`'s methods

Methods		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	`createDefaultXMLParser`	(0/28)	(0/28)	(0/28)	(0/28)	(0/28)
2	`createDefaultXMLParser`	(0/0)	N (0/0)	(0/0)	(0/0)	(0/0)
3	`createXMLParser`	(0/28)	N (28/28)	(0/28)	(0/28)	(0/28)
4	`<init>`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)

Table 7: Class `StdXMLReader's` Methods

Methods		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	`atEOF`	(6/6)	(0/6)	(0/6)	(0/6)	(6/6)
2	`atEOFOfCurrentStream`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
3	`read`	(5/5)	(0/5)	(0/5)	(0/5)	(5/5)
4	`getLineNr`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
5	`openStream`	(0/0)	(4/4)	(4/4)	(4/4)	(4/4)
6	`stringReader`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
7	`<init>`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
8	`startNewStream`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
9	`unread`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
10	`getEncoding`	-	(0/0)	(0/0)	(0/0)	(0/0)
11	`stream2reader`	-	(0/0)	(0/0)	(0/0)	(0/0)
12	`getPublicID`	-	(0/0)	(0/0)	(0/0)	(0/0)
13	`getSystemID`	-	(0/0)	(0/0)	(0/0)	(0/0)
14	`fileReader`	-	(0/0)	(0/0)	(0/0)	(0/0)
15	`<init>`	-	(0/0)	(0/6)	(0/6)	(0/6)
16	`<init>`	-	(0/0)	(0/0)	(0/0)	(0/0)
17	`setPublicID`	-	(0/0)	(0/0)	(0/0)	(0/0)
18	`setSystemID`	-	N (0/0)	(0/0)	(0/0)	(0/0)
19	`finalize`	-	-	(0/0)	(0/0)	(0/0)
20	`class$`	-	-	-	(0/0)	(0/0)

We see that the `createXMLParser` method from class `XMLParserFactory` selects all 28 tests. The reason is that this is a 'core' method that is called to create the XML parser (NanoXML *is* an XML parser) hence all tests hit this method, and from version v1 to v2 this method is changed, so all tests are selected.

5.1.3.2 Component Test Selection

We also look at the component tests selected by the code based technique because it helps us consider test selection effectiveness later Section 5.4.

Table 8 shows the number of tests selected across the different versions (total number of tests being 214 (v0 to v3) and 216 (v4 and v5)). The total number of test selected is very high for most of the pairs (more tests pass through changed code), except from v3-v4. This is because between v3 and v4 only one new method has been introduced.

Table 8: Component Test Selection

Versions	Number of selected tests
v0 – v1	203 (94.86%)
v1- v2	214 (100.00%)
v2 – v3	214 (100.00%)
v3- v4	0 (0.00%)
v4 – v5	210 (97.22%)

Version pairs v1-v2 and v2-v3 show a 100% selection rate. This is not due to any single class or method change (as opposed to the application test selection case (v1-v2)), as we can see from Table 9 which gives us a more detailed class-wise test selection rate. As expected the core classes `NonValidator`, `StdXMLBuilder`, `StdXMLParser`, `StdXMLReader`, and `XMLParserFactory` have a lot of code changes and so correspondingly a lot of test selection.

Table 9: Component Test Selection (Class-wise)

Component Classes	v0 – v1	v1 –v2	v2 – v3	v3 – v4	v4 – v5
`NonValidator`	195 (91.12%)	201 (93.93%)	150 (70.09%)	0 (0%)	67 (31.02%)
`StdXMLBuilder`	129 (60.28%)	150 (70.09%)	150 (70.09%)	0 (0%)	0 (0%)
`StdXMLParser`	203 (94.86%)	206 (96.26%)	205 (95.79%)	0 (0%)	207 (95.83%)
`StdXMLReader`	197 (92.06%)	206 (96.26%)	51 (23.83%)	0 (0%)	207 (95.83%)
`XMLElement`	0 (0%)	12 (5.61%)	140 (65.42%)	0 (0%)	0 (0%)
`XMLParseException`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`XMLParserFactory`	0 (0%)	206 (96.26%)	0 (0%)	0 (0%)	0 (0%)
`XMLValidationException`	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
`XMLWriter`	0 (0%)	68 (31.78%)	68 (31.77%)	0 (0%)	75 (34.72%)
`ContentReader`	-	117 (54.67%)	0 (0%)	0 (0%)	0 (0%)
`XMLEntityResolver`	-	95 (44.39%)	0 (0%)	0 (0%)	15 (6.94%)
`XMLUtil`	-	205 (95.79%)	0 (0%)	0 (0%)	207 (95.83%)
`ValidatorPlugin`	-	-	0 (0%)	0 (0%)	0 (0%)
`XMLException`	-	-	73 (34.11%)	0 (0%)	0 (0%)
`XMLAttribute`	-	-	-	0 (0%)	0 (0%)

Studies [8] have shown that when there are lots of code changes between versions the opportunities for test selection decrease. That is, the amount of change made between regression testing sessions strongly affects the costs and benefits of regression test selection techniques. These results further validate those findings.

We look at the same classes from Tables 6 and 7 in Tables 10 and 11. The test selection is similar to those seen for the application tests, with all tests being selected for the method `createXMLParser` in class `XMLParserFactory`. Appendix B presents results for other classes.

Table 10: Class `XMLParserFactory's` Methods

Methods		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	`createDefaultXMLParser`	(0/197)	(0/201)	(0/204)	(0/206)	(0/206)
2	`createDefaultXMLParser`	(0/6)	N (5/5)	(0/2)	(0/2)	(0/2)
3	`createXMLParser`	(0/203)	N (206/206)	(0/206)	(0/208)	(0/208)
4	`<init>`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)

Table 11: Class `StdXMLReader's` Methods

Methods		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	`atEOF`	(197/197)	(48/205)	(0/205)	(0/207)	(207/207)
2	`atEOFOfCurrentStream`	(0/66)	(0/0)	(0/0)	(0/0)	(0/0)
3	`Read`	(197/197)	(156/205)	(0/205)	(0/207)	(207/207)
4	`getLineNr`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
5	`openStream`	(0/0)	(51/51)	(51/51)	(0/51)	(51/51)
6	`stringReader`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
7	`<init>`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
8	`startNewStream`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
9	`Unread`	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
10	`getEncoding`	-	(0/204)	(0/204)	(0/206)	(0/206)
11	`stream2reader`	-	(0/206)	(0/206)	(0/208)	(0/208)
12	`getPublicID`	-	(0/0)	(0/0)	(0/0)	(0/0)
13	`getSystemID`	-	(0/0)	(0/0)	(0/0)	(0/0)
14	`fileReader`	-	(0/206)	(0/206)	(0/208)	(0/208)
15	`<init>`	-	(206/206)	(0/206)	(0/208)	(0/208)
16	`<init>`	-	(0/0)	(0/0)	(0/0)	(0/0)
17	`setPublicID`	-	(0/0)	(0/0)	(0/0)	(0/0)
18	`setSystemID`	-	N (0/0)	(0/0)	(0/0)	(0/0)
19	`Finalize`	-	-	(0/0)	(0/0)	(0/0)
20	`class$`	-	-	-	(0/0)	(0/0)

5.2 Specification Based Technique

5.2.1 Procedure

Our specification based selection technique requires that we have a statechart diagram for each component in each version of our tested system - NanoXML. We used the Rational

Rose tool to draw these statechart diagrams. (For details on how we created the diagrams see Section 4.4.)

As described in Section 3.2.2, our technique also requires us to have linearly independent paths for each component in each version, because generating all possible paths is not practical, so we built a path generation tool, which generated linearly independent paths for each statechart diagram in each version. Using the `Dispenser` component as an example, the path generation result for `Dispenser` is shown in Table 1.

Following the modified Dejavu selection algorithm presented in Section 3.1.2, we built a tool which compared two statechart diagrams (original and revised) and generated the affected edges (the set of states and transitions affected by the changes to the specification). We used this tool to compare the statechart for each component in each version with the corresponding statechart in a successive version, and determined the affected edges for each component in this version. Section 3.2.2 shows an example result of this step applied to the `Dispenser` component.

We also built a mapping tool which maps all the edges to the paths in a diagram. The output is an edge-path matrix with 0 or 1, where 0 indicates that the edge is not on the path, and 1 indicates that the edge is on the path. We used this mapping tool to map all the edges to the paths for each component in each version (with the exception of components in the last version, v5) and obtained an edge-path matrix for each component in each version except v5.

Finally we extracted the affected edges from the edge-path matrix, and obtained the component is often either 100% or 0%, and (2) in most components the selection rates do not change between the two approaches, or slightly decrease for the IgnoreNew approach.

Table 12: Selection Rate for Each Version

Versions	IncludeNew approach			IgnoreNew approach		
	Total Path Num	Selected Path Num	Path Selection Rate	Total Path Num	Selected Path Num	Path Selection Rate
v0-v1	89	51	57%	89	48	54%
v1-v2	129	116	90%	129	103	80%
v2-v3	156	100	64%	156	73	46%
v3-v4	187	51	25%	187	10	5%
v4-v5	188	77	41%	188	71	38%

5.2.3 Discussion

In this study, we explored the specification-based test selection technique by using UML statechart diagrams. Looking at overall selection data shown in Table 12, the path selection rate is reduced by both the IncludeNew and the IgnoreNew approaches. It seems that cost can be reduced by using the two approaches. But the data in Table 13 shows that our overall savings are due to components that are not changed at all. For example, all the components in Version 1 are changed, so the selection rate for this version is very high (80% to 90%); while in Version 3, most components are not changed, so the selection rate is low (5% to 25%). Also shown in Table 13, the selection rates in the changed components are often 100%, which suggests that selection at the fine grain of individual paths in those components does not help. It might often be equally cost effective just to consider which components have changed and run all their tests.

The data in Table 13, however, also shows that there are a few components with low but non-zero selection rates, such as the `XMLUtil` and `XMLException` components. When test costs are very high, it might be worth seeking extra savings on such components by selecting at the path level.

In this study we used two different approaches to select paths. The IgnoreNew approach might let us save tests but sacrifices safety. But our results show that there was often not a lot of difference in selection rates between the two approaches per component. But occasionally there was a difference, so again if costs are very high, it could be worth while to use the IgnoreNew approach, at a sacrifice of safety. If costs are not very high, it may not be worth while to use the IgnoreNew approach.

Table 13: Selection Rate for Each Version (Class-wise)

Components	Version0		Version1		Version2		Version3		Version4	
	Include New	Ignore New	Include New	Ignore New	Include New	Ignore New	Include New	Ignore New	Include New	Ignore New
XMLParserFactory	0%	0%	83%	66%	0%	0%	0%	0%	0%	0%
StdXMLReader	100%	100%	100%	25%	0%	0%	0%	0%	100%	100%
StdXMLParser	95%	91%	100%	95%	90%	85%	45%	45%	100%	100%
NonValidator	100%	100%	100%	100%	100%	100%	0%	0%	100%	100%
XMLElement	0%	0%	91%	90%	100%	86%	100%	3%	0%	0%
XMLWriter	0%	0%	100%	25%	100%	100%	0%	0%	0%	0%
StdXMLBuilder	100%	100%	100%	100%	90%	90%	0%	0%	0%	0%
XMLParseException	100%	0%	100%	100%	100%	100%	0%	0%	0%	0%
XMLValidationException	0%	0%	100%	100%	100%	100%	0%	0%	0%	0%
ContentReader	N	N	100%	100%	0%	0%	0%	0%	100%	100%
XMLEntityResolver	N	N	100%	50%	0%	0%	0%	0%	100%	0%
XMLUtil	N	N	52%	47%	0%	0%	0%	0%	55%	52%
ValidatorPlugin	N	N	N	N	100%	30%	0%	0%	0%	0%
XMLException	N	N	N	N	20%	20%	0%	0%	0%	0%
XMLAttribute	N	N	N	N	N	N	0%	0%	0%	0%

N represents no such component in this version.

files. It then compares the component trace file information with the information from the application traces files. If there is a match between any of the application trace files and the component trace files, the application trace file's name is stored under the trace file name of the component trace file. A match in our case is measured in terms of a specific percentage of the functions and the basic blocks hit being identical. For example, a 90% function match means 90% of the functions in a component test trace are present in a particular application test trace. If all the blocks that were hit in a component test were present in the trace of an application test, then there is said to be 100% match. We can use any percentage we want as an acceptable match. This implementation allows us to compare either the function hits or the basic block hits separately.

Unfortunately, the mapping using the basic blocks hit could not be used for our subject as there was no significant correspondence between the trace files even if we reduced the percentage of match required to infer a correspondence. When using basic blocks as the mapping criteria, a mapping was not obtained as there were no tests in the application that hit 95% or more of the blocks hit by any component test. This remained true for even a low match percentage of 65%. Therefore the basic blocks mapping was not useful for the study.

The mapping using the functions, however, did produce a correspondence for our subject. For an acceptable match percentage set at 95%, we had a well distributed mapping between the application tests and the test frames.

Once we had the mapping information, we collected the test frames that we changed from one version to another in the component. Since no new functional unit was added, the test frames changed were limited. A test frame is said to be changed if the test driver for that particular test frame was changed or if there had been new frames added or old ones modified. Frames were added or modified to make the test case for that particular test frame work for the newer version of the component.

We then wrote a program in Java called `TestSelection` which takes in the mapping information output by `MapTest` and a file containing the test frames that are changed, and outputs a file listing the application test cases which need to be run for each version.

5.3.2 Results

Table 14 shows the test frames selected as changed. The reason the number of test frames that are changed from v0 to v1 is 214 is because all the test drivers from v0 to v1 needed to be changed. There were no changes in the test frames from version v3 onwards.

Table 14: Component Frames Changed

Versions	Frames Changed
v0 – v1	214/214 (100.00%)
v1 – v2	47/214 (21.96%)
v2 – v3	17/214 (7.94%)
v3 – v4	0/216 (0.00%)
v4 – v5	0/216 (0.00%)

Table 15 gives the number of application tests that need to be rerun when the version of the component changes. The reason 23 of the 28 tests were selected for v1 is because of the fact that almost all the test frames changed from v0 to v1. These frames changed primarily because the test drivers changed when we moved from version v0 to v1.

Table 15: Application Tests Selected

Versions	Application Tests Selected
v0-v1	23/28 (82.14%)
v1-v2	5/28 (17.86%)
v2-v3	5/28 (17.86%)
v3-v4	0/28 (0.00%)
v4-v5	0/28 (0.00%)

Our results on versions 1 through 5 were heavily influenced by the way we developed our TSL specifications. We did not add to our specifications to account for new functionality that was added to the component in newer versions. But the test frames were only modified to make the existing ones work for the newer version. For example, a function which takes in a three parameters instead of an earlier two, the test frames needs to be changed slightly to account for this. Had we built the TSL specification to take the added functionality into account, the number of tests selected could have been greater. But then again, we developed our test plans and tests before the experiment was designed and it seemed intuitive to do it this way without any prior knowledge of the experiment. The results of the experiment thus avoid a potential source of bias.

The type of the system we studied may also have played a role in these results. The system was a parser, and thus the mapping was very difficult to obtain between application tests and the component test as the input file played a large role in determining the path taken by the program. Also, the sequential function trace and the basic block trace could have been used to obtain a mapping between the two.

5.4 Comparisons across Techniques

The three techniques yielded very interesting results. Unfortunately, it was difficult to compare the state based techniques with the other two techniques, namely, the code based and the hybrid technique. The reason for this was that the state based techniques selected test paths rather than test cases for regression test selection across the versions. In the future, when test cases are developed for the different test paths, the comparison between the three techniques would become more accurate.

We thus compare (Table 16) the work required to be redone (percentage of retesting required) for the three techniques. Although we are comparing slightly different things, the percentages represent the retesting effort required as given by the techniques. For the code based and the hybrid techniques it is the percentage of application tests selected to be rerun. And though the specification based technique does not select tests from the application test suite we can still compare with it because the number of test paths gives an idea of the number of test cases.

Table 16: Comparison across Techniques

Versions	Code Based	State Based (a)	State Based (b)	Hybrid
v0 to v1	21.43%	57 %	54 %	82.14%
v1 to v2	100.00%	90 %	80 %	17.86%
v2 to v3	21.43%	62 %	44 %	17.86%
v3 to v4	0.00%	25 %	5 %	0.00%
v4 to v5	21.43%	41 %	38 %	0.00%

From v3 to v4 there were no code changes, except the addition of a new method, and so there are no tests selected by the code-based technique, but since the state based technique depends on UML diagrams and paths therein, some tests are selected.

Table 17 is a further comparison between the code based and the hybrid method. Apart from versions v0-v1 and v3-v4, there is a marked difference between the tests selected by the methods. The code based selects more component tests overall than the hybrid technique.

When a new system version comes out, the system tends to become stable in its functionality and at some point the system's functionality does not change any more and possibly some variable/syntax changes or minor error corrections may occur. This explains why the hybrid technique selects fewer tests than the code based technique as the system reaches higher versions. The hybrid technique generates test cases based on the TSL specification, which considers the system at a functional level, and it selects no tests on v3-v4 and v4-v5, showing there were no functional changes in those versions. The presented code based technique, however, reflects every single code (basic block) change in the system, and thus it still selects a large portion of tests from v4 to v5.

Table 17: Comparing the Code Based and Hybrid Techniques

Versions	Application Tests Selected		Component Test Selected	
	Code Based	Hybrid	Code Based	Hybrid
v0-v1	6/28	23/28	203/214	214/214
v1-v2	28/28	5/28	214/214	47/214
v2-v3	6/28	5/28	214/214	17/214
v3-v4	0/28	0/28	0/216	0/216
v4-v5	6/28	0/28	210/216	0/216

Table 18 shows the application tests selected by the code based and the hybrid techniques. The numbers shown in the table represent test case numbers 0 through 27. The methods select different tests due to different approaches and different algorithms used. The individual reasons for this kind of selection for the code based and the hybrid techniques are discussed in Sections 5.1.3.1 and 5.3.2 respectively. Together the two techniques seem to compliment each other, with little overlap in the test cases selected.

Table 18: Application Test Selection for Code Based and Hybrid Techniques

Versions	Code Based Technique	Hybrid Technique
v0-v1	0 1 2 3 4 6	0 1 2 3 7 8 9 10 11 12 13 14 16 17 19 20 21 22 23 24 25 26 27
v1-v2	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27	22 24 25 26 27
v2-v3	0 1 2 3 4 6	22 24 25 26 27
v3-v4	-	-
v4-v5	0 1 2 3 4 6	-

The code based technique is based on comparisons of CFGs between pairs of methods to see if there is any code change inside the method. If so, any test cases that go through the changed code (actually changed basic blocks) will be selected. The code based technique can be implemented at different levels, depending on what level we instrument the code at. In our case the code based technique is at very low level, since we instrument at the basic block level.

The specification based technique is based on comparison of statechart diagrams between two classes. If there is any change between two statechart diagrams (such as method parameter changes), the state is considered 'dangerous' and paths (or test requirements) that go through this state are selected. The specification based technique is thus at higher level than the code based technique. The overall selection rate of the code based technique is thus expected to be higher than that of the specification based technique.

The hybrid technique is at a functional level. When we made the TSL specifications, we divided the whole system into several functional units (a functional unit is not necessarily a class - it may involve several classes), then we considered the parameters and environmental conditions for each functional unit. The tester can control the size of the tests and in our implementation we tested at a 'high level'. The overall selection rate of the TSL based technique turned out lower than those of the other techniques. Also when we compare two component versions by using TSL, we just adapted test frames of previous versions to the later versions. This caused there to be few frame changes between versions, lowering the selection rate further.

Chapter 6
CONCLUSIONS AND FUTURE WORK

We have looked at three techniques for regression testing of component based applications. The first technique is code based, the second technique is specification based, and the third is a hybrid of the two. We constructed infrastructure that allows use of metadata for regression test selection using the three techniques; and studied a real world component-application that demonstrates the potential usefulness of metadata in regression test selection.

Overall the code based technique performed the worst with an average selection rate of almost 100% for component test selection. But it selected an average of 22% of the application tests. This shows that the test selection rate differs for different drivers. When we performed test selection with test drivers which cover most methods in the components, we ended up selecting more tests because more changed code is likely to be hit.

The specification based and hybrid techniques fared better, but the results basically depend on the component developer. Changing code with no specification changes or changing specifications and not the code will result in different percentages of test selection. Also developers sometimes release versions after a long time (with lots of code and specification changes), this may lead to a higher test selection rate. And sometimes releases with little or no change (in our case between v3 and v4) will select no tests. Releasing component versions regularly and with the corresponding specification changes would make the techniques most effective.

The specification based testing technique required us to have a statechart diagram for component classes. Since we did not have text specifications for any component class in NanoXML, we reverse engineered the diagrams by looking at the code. This resulted in very low level diagrams (for example, every method call is a transition). This can result in very complex diagrams when two or more diagrams are combined. We thus need higher level diagrams to work with. As a next step, we would look for component-applications that have diagrams included with them, that is, which include diagrams that were made as design requirements. We can then expect to do integration tests of components and applications by combining statechart diagrams.

The code based results show that the component had relatively large changes in code. We would like to look at different programs that have different amounts of code changes between releases. This would help in determining different levels of instrumentation (statement-level, method-level, and component-level).

Also the study needs to be conducted on a wider range of programs with more diverse code bases, specifications and UML diagrams. This study has been performed on only one subject (NanoXML), but the study achieved significant by-products:

1. identifying metadata in each of the three techniques.
2. construction of infrastructure for testing general component-based software.
3. better understanding of the effects of software domain on test technique adaptation.

APPENDIX A

Package net.n3.nanoxml [16]

Interface Summary	
IXMLBuilder	NanoXML uses IXMLBuilder to construct the XML data structure it retrieved from its data source.
IXMLElement	IXMLElement is an XML element.
IXMLEntityResolver	An IXMLEntityResolver resolves entities.
IXMLParser	IXMLParser is the core parser of NanoXML.
IXMLReader	IXMLReader reads the data to be parsed.
IXMLValidator	IXMLValidator processes the DTD and handles entity references.

Class Summary	
NonValidator	NonValidator is a concrete implementation of IXMLValidator which processes the DTD and handles entity definitions.
StdXMLBuilder	StdXMLBuilder is a concrete implementation of IXMLBuilder which creates a tree of IXMLElement from an XML data source.
StdXMLParser	StdXMLParser is the core parser of NanoXML.
StdXMLReader	StdXMLReader reads the data to be parsed.
ValidatorPlugin	ValidatorPlugin allows the application to insert additional validators into NanoXML.
XMLElement	XMLElement is an XML element.
XMLEntityResolver	An XMLEntityResolver resolves entities.
XMLParserFactory	Creates an XML parser.
XMLWriter	An XMLWriter writes XML data to a stream.

Exception Summary	
XMLException	An XMLException is thrown when an exception occurred while processing the XML data.
XMLParseException	An XMLParseException is thrown when the XML passed to the XML parser is not well-formed.
XMLValidationException	An XMLValidationException is thrown when the XML passed to the XML parser is well-formed but not valid.

APPENDIX B
METHOD-WISE TEST SELECTION INFORMATION

Class ContentReader's Methods

		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	read	-	(117/117)	(0/117)	(0/119)	(0/119)
2	close	-	(117/117)	(0/117)	(0/119)	(0/119)
3	<init>	-	N (0/0)	(0/0)	(0/0)	N (0/0)
4	finalize	-	-	(0/0)	(0/0)	(0/0)

Class XMLParseException's Methods

		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	getLineNr	(0/0)	N (0/0)	-	-	-
2	<init>	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
3	<init>	(0/0)	N (0/0)	(0/33)	(0/23)	(0/23)
4	getException	-	N (0/0)	-	-	-
5	<init>	-	N (0/0)	-	-	-

Class StdXMLBuilder's Methods

Methods		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	getResult	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
2	<init>	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
3	addAttribute	N (121/121)	N (143/143)	(143/143)	(0/144)	(0/144)
4	addPCData	(0/0)	N (117/117)	N (117/117)	(0/119)	(0/119)
5	endElement	N (128/128)	(0/148)	(148/148)	(0/150)	(0/150)
6	newProcessingInstruction	(0/0)	N (0/0)	(0/0)	(0/0)	(0/0)
7	startBuilding	(0/0)	N (0/0)	(0/0)	(0/0)	(0/0)
8	startElement	N (129/129)	N (150/150)	(150/150)	(0/151)	(0/151)
9	elementAttributesProcessed	-	(0/0)	(0/0)	(0/0)	(0/0)
10	finalize	-	-	(0/0)	(0/0)	(0/0)
11	<init>	-	-	-	(0/207)	(0/207)

Class XMLValidationException's Methods

		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	getAttributeName	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
2	getAttributeValue	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
3	getElementName	(0/0)	(0/0)	(0/0)	(0/0)	(0/0)
4	getLineNr	(0/0)	N (0/0)	(0/0)	(0/0)	(0/0)
5	<init>	(0/0)	N (0/0)	(0/0)	(0/0)	(0/0)

Class `NonValidator`'s Methods

		v0-v1	v1-v2	v2-v3	v3-v4	v4-v5
1	scanIdentifier	N (66/66)	-	-	-	-
2	scanString	N (66/66)	-	-	-	-
3	readChar	N (66/66)	-	-		-
4	processAttList	N (4/4)	N(16/16)	(0/16)	(0/16)	(16/16)
5	processElement	N (66/66)	N (67/67)	(0/67)	(0/67)	(67/67)
6	processEntity	N (66/66)	N (67/67)	(0/67)	(0/67)	(67/67)
7	skipComment	N (0/0)	-	-	-	-
8	skipTag	N (4/4)	-	-	-	-
9	skipWhitespace	N (66/66)	-	-	-	-
10	ReadergetEntity	N (4/4)	-	-	-	-
11	<init>	(0/0)	(0/214)	(0/206)	(0/208)	(0/208)
12	PCDataAdded	(0/0)	N (0/0)	(0/0)	(0/0)	(0/0)
13	attributeAdded	N (121/121)	N (143/143)	N (143/143)	(0/145)	(0/145)
14	elementEnded	N (128/128)	N (0/0)	N (0/0)	(0/0)	(0/0)
15	elementStarted	N (129/129)	N (150/150)	N (150/150)	(0/152)	(0/152)
16	parseDTD	N (66/66)	N (67/67)	(0/67)	(0/67)	(67/67)
17	processConditionalSection	-	N (0/0)	(0/0)	(0/0)	(0/0)
18	processIgnoreSection	-	N (0/0)	(0/0)	(0/0)	(0/0)
19	elementAttributesProcessed	-	N (149/149)	N (149/149)	(0/151)	(0/151)
20	finalize	-	-	(0/0)	(0/0)	(0/0)
21	getParameterEntityResolver	-	-	(0/0)	(0/0)	(0/0)
22	setParameterEntityResolver	-	-	(0/0)	(0/0)	(0/0)
23	<clinit>	-	-	(0/0)	(0/0)	(0/0)

BIBLIOGRAPHY

[1] G. Booch, J. Kumbaugh, I. Jacobson. The Unified Modeling Language User Guide. Addison Wesley, pages 336-339, 2001.

[2] K. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software, *Proc. of the Nat'l. Tele. Conf.* B-6-3, pages 16, Nov. 1981.

[3] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel., "An Empirical Study of Regression Test Selection Techniques", *Proceedings of the 20th International Conference on Software Engineering*, April 1998.

[4] M. J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using Component Metadata to Support the Regression Testing of Component-Based Software, *IEEE International Conference on Software Maintenance,* November, 2001, Florence, Italy.

[5] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An Empirical Investigation and Comparison of Program Spectra, *Technical Report OSU-CISRC-11/97-TR55 The Ohio State University,* November, 1997.

[6] M. Harrold and M. Soffa. An incremental approach to unit testing during maintenance, *Proc. of the Conf. on Softw. Maint.*, pages 362-367, Oct. 1988.

[7] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based Integration Testing. *International Symposium on Software Testing and Analysis*, pages 60-70, August 2000.

[8] J.-M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Application Frequency, *Proceedings of the 22nd International Conference on Software Engineering,* June, 2000, pages 126-135.

[9] P. M. Maurer. *Components: What if they gave a revolution and nobody came.* IEEE Computer, 33(6); pages 60-66, June 1998.

[10] A. Orso, H. Do, G. Rothermel, M. J. Harrold, M. L. Soffa, and D. S. Rosenblum. Using Component Metacontent to Support the Regression Testing of Component-Based Software, 2000. [Unpublished]

[11] T. Ostrand and M. Balcer ,"The Category-Partition Method for Specifying and Generating Functional Tests", *Communications of the ACM*, Volume 31, Number 6, June 1988, pp 676-686.

[12] R. S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, Fifth edition, 2001, pages 445-451.

[13] G. Rothermel and M. J. Harrold. A Safe, Efficient Regression Test Set Selection Technique, *ACM Transactions on Software Engineering and Methodology*, V.6, no. 2, April 1997, pages 173-210.

[14] Marc De Scheemaecker, "NanoXML/Java2.1"

[15] C.Szyperski. *Component Oriented Programming*. Addision Wesley, first edition, 1997.

[16] http://web.wanadoo.be/cyberelf/nanoxml/documentation/NanoXML-2-JavaDoc/index.html