Using Component Metacontent to Support the Regression Testing of Component-Based Software

Alessandro Orso[*]
orso@cc.gatech.edu

Mary Jean Harrold[*]
harrold@cc.gatech.edu

David Rosenblum[†]
dsr@ics.uci.edu

Gregg Rothermel[‡]
grother@cs.orst.edu

Mary Lou Soffa[§]
soffa@cs.pitt.edu

Hyunsook Do[‡]
dohy@cs.orst.edu

Abstract

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. In previous work, we proposed the use of component metacontents—additional data and methods provided with a component, to support software engineering tasks. In this paper, we present two new metacontent-based techniques that address the problem of regression test selection for component-based applications: a code-based approach and a specification-based approach. First, we illustrate the two techniques. Then, we present a case study that applies the code-based technique to a real component-based system. On the system studied, on average, 26% of the overall testing effort was saved over seven releases, with a maximum savings of 99% for one version.

1 Introduction

Interest in component-based software continues to grow with the recognition of its potential in managing the growing complexity of software systems [13, 21, 23].[1] With increasing frequency, software engineers are building systems by integrating externally-developed software components with application-specific code [3, 13, 23]. Although the use of components provides many advantages, serious drawbacks in that use are becoming apparent. These drawbacks impinge on a wide range of software engineering activities. For example, component usage threatens our ability to validate software [24] and assess reliability [3], complicates maintenance [23], causes problems in program understanding [8], and introduces threats to security [12].

In many cases, the drawbacks of component-based software technologies arise because of the lack of information about externally provided components. Providing the component's source code is not a viable solution to this problem because of intellectual property issues: Component developers often are unwilling to provide the required information unless they can do so without revealing source code or other sensitive details. Furthermore, not all the required information can be (easily and efficiently) derived from the code alone: Some information, such as data dependences or complexity metrics, may be expensive to compute; other information, such as documentation or changes with respect to previous versions, must be provided in addition to the code.

A preferable approach is to provide information with components that can be efficiently used "out-of-the-box," accommodates intellectual property needs, and yet, supports the full range of software engineering tasks for which component users require assistance. We refer to such additional information as *component metacontent*.[2] Metacontents describe static and dynamic aspects of a component, can be accessed by the component user, and can be used for various tasks. Metacontents consist of information (*metadata*) about components and utilities (*metamethods*) for computing and retrieving such information. As a parallel, consider battery manufacturers, who provide static information (metadata) such as battery size and voltage, with their batteries (components), and dynamic retrieval facilities (metamethods relying on "human" processors) that users can exploit to determine the charge remaining in a battery.

Existing component standards and environments, including DCOM [4], Enterprise JavaBeans [9], and .NET Common Language Runtime (CLR) [14], already supply some

[*] College of Computing, Georgia Institute of Technology
[†] Information and Computer Science, University of California, Irvine
[‡] Department of Computer Science, Oregon State University
[§] Department of Computer Science, University of Pittsburgh

[1] There are many existing definitions of component; for simplicity, and without loss of generality where the proposed work is concerned, we adopt an informal definition of a *component* as a system or subsystem developed by one organization and deployed by one or more other organizations, possibly in different application domains [15].

[2] For the sake of brevity, in the rest of the paper we simply refer to component metacontent as *metacontent*.

additional information about a component through metadata that are packaged with the component. The metadata available so far, however, are typically limited to information useful for compile-time and run-time type-checking (e.g., the name of the component's class, the names of its functions, and the types of the functions' parameters), and for design-time customization (e.g., the shape or color of a graphical user interface component, and the maximum size of the internal buffer of a data storage component). Researchers have proposed extending the use of such information for specific tasks [5, 17, 22, 26], but the varieties of metadata currently supported address only a limited range of software engineering problems, such as providing deployment descriptions of components [5, 17] or enhancing self-documentation [26]. None of the metadata proposed has addressed the important software engineering task of regression testing component-based software. Moreover, to the best of our knowledge, none of the approaches presented so far considers the use of metamethods to dynamically build metadata at run-time.

In previous work, we introduced a general framework for producing and consuming metacontents whose goals are (1) to support the broad range of software engineering tasks that depend on and can benefit from information about external components, and (2) to accommodate component providers' intellectual-property concerns [15]. This paper explores the application of our metacontent framework to the problem of regression test selection for component-based software. More precisely, we investigate the following question:

> Given (1) an application A that uses a set of externally-developed (black-box) components C and has been tested with a test suite T, and (2) a new version of this set of components C', how can we exploit metacontents to build a test suite $T' \subseteq T$ that can reveal possible regression faults in A due to changes in C' and that does not include test cases not impacted by such changes?

To address this question, there are several problems to resolve. First, we must identify the metacontents needed to perform the chosen task. 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. Second, we must determine how to adapt existing regression test selection techniques to incorporate the use of metacontents. Different types of techniques must be considered to demonstrate the general applicability of the approach. Finally, we must demonstrate the value of using metacontents for regression test selection on component-based software; we must study the cost-effectiveness of using the regression test selection technique proposed.

In this paper, we present metacontents and techniques that use metacontents for regression test selection on component-based software for two different types of approaches: code-based regression test selection, based on edge-level and method-level regression test selection algorithms [2, 7, 19]; and specification-based regression test selection, based on the category-partition method [16]. For both types of approaches, we (1) identify the metacontents necessary to determine the test cases to rerun and (2) present techniques for using the metacontents for regression test selection. We also describe the results of a study, performed on a real component-based system, that compares the costs of regression test selection with and without metacontents. The empirical results demonstrate that there can be significant savings in the number of test cases that must be rerun for regression testing when component metacontents are available, and thus indicate the usefulness of metacontents for regression testing.

The main contributions of the paper are the following:

1. identification and use of metacontents for regression test selection using a code-based technique;
2. identification and use of metacontents for regression test selection using a specification-based technique;
3. demonstration of the usefulness of metacontents in code-based regression test selection for a real program.

2 Component Metacontents for Regression Test Selection

We present two approaches for providing and using component metacontents for regression test selection: code-based and specification-based. The approaches address the general case of an application A that uses a set C of components. To illustrate, we use an example consisting of a component and an application that uses it. (For space reasons, the example is limited to a single component.) The component, `Dispenser`, and the application, `VendingMachine`, both in Java, are presented in Figure 1; note that, whereas we show the source code of `Dispenser` in the figure, we assume that the source code of `Dispenser` is unavailable to the developer of `VendingMachine`.

The application models a vending machine that dispenses specific items to a user. A user can (1) `insert` coins into the machine, (2) ask the machine to `return` the coins inserted and not consumed, and (3) ask the machine to `vend` a specific item. If the requested item is not available, the credit is insufficient, or the selection is invalid, the machine prints an error message and does not dispense the item.

We developed a test suite for `VendingMachine` (shown in Table 1). Each test case is a sequence of method calls.[3] The test cases are grouped into three

[3]For brevity, in the call sequences, we do not show the initial call to the constructor of class `VendingMachine`, which is implicitly invoked when the class is instantiated.

```
 1. public class VendingMachine {
 2.
 3.  final private int COIN = 25;
 4.  final private int VALUE = 50;
 5.  private int totValue;
 6.  private int currValue;
 7.  private Dispenser d;
 8.
 9.  public VendingMachine() {
10.   totValue = 0;
11.   currValue = 0;
12.   d = new Dispenser();
13.  }
14.
15.  public void insert() {
16.   currValue += COIN;
17.   System.out.println("Current value = " + currValue );
18.  }
19.
20.  public void return() {
21.   if ( currValue == 0 )
22.    System.err.println( "no coins to return" );
23.   else {
24.    System.out.println("Take your coins");
25.    currValue = 0;}
26.  }
27.
28.  public void vend( int selection ) {
29.   int expense;
30.   expense = d.dispense( currValue, selection );
31.   totValue += expense;
32.   currValue -= expense;
33.   System.out.println( "Current value = " + currValue );
34.  }
35. } // class VendingMachine
36.
37. public class Dispenser {
38.
39.  final private int MAXSEL = 20;
40.  final private int VAL = 50;
41.  private int[] availSelectionVals = {2,3,13};
42.
43.  public int dispense( int credit, int sel ) {
44.   int val=0;
45.   if ( credit == 0 )
46.    System.err.println("No coins inserted");
47.   else if ( sel > MAXSEL )
48.    System.err.println("Wrong selection "+sel);
49.   else if ( !available( sel ) )
50.    System.err.println("Selection "+sel+" unavailable");
51.   else {
52.    val = VAL;
53.    if ( credit < val )
54.     System.err.println("Enter "+(val-credit)+" coins");
55.    else
56.     System.err.println("Take selection"); }
57.   return val;
58.  }
59.
60.  private boolean available( int sel ) {
61.   for (int i = 0; i<availSelectionVals.length; i++)
62.    if (availSelectionVals[i] == sel) return true;
63.   return false;
64.  }
65. } // class Dispenser
```

Figure 1. Application `VendingMachine` and component `Dispenser`.

Table 1. Test suite used to test `VendingMachine`.

Test Case #	Test Case	Result
Value passed to vend: 3 (i.e., valid selection, available item)		
1	return	Passed
2	vend	Passed
3	insert, return	Passed
4	insert, vend	Failed
5	insert, insert, return	Passed
6	insert, insert, vend	Passed
7	insert, insert, insert, return	Passed
8	insert, insert, insert, vend	Passed
9	insert, insert, insert, insert, return	Passed
10	insert, insert, insert, insert, vend	Passed
11	insert, insert, return, vend	Passed
12	insert, insert, vend, vend	Passed
13	insert, insert, insert, return, vend	Passed
14	insert, insert, insert, vend, vend	Failed
15	insert, insert, insert, insert, return, vend	Passed
16	insert, insert, insert, insert, vend, vend	Passed
Value passed to vend: 9 (i.e., valid selection, unavailable item)		
17	vend	Passed
18	insert, vend	Passed
19	insert, return, vend	Passed
20	insert, vend, vend	Passed
Value passed to vend: 35 (i.e., invalid selection)		
21	vend	Passed
22	insert, vend	Passed
23	insert, insert, vend	Passed
24	insert, insert, insert, vend	Passed
25	insert, insert, insert, insert, vend	Passed

sets (1–16, 17–20, 21–25) based on the value of parameter `selection` that is passed to method `VendingMachine.vend`. The table indicates whether each test case passed or failed. Test cases 4 and 14 failed because of an error in method `Dispense.dispense`: If an available item is selected and the credit is insufficient, but greater than zero, then variable `value` (set to `VAL` at line 52) is not reset to zero; consequently, when control returns from `Dispense.dispense` to `VendingMachine.vend`, `currValue` is erroneously decremented.

Suppose that the component developer finds and fixes this error in `Dispenser` by adding statement "`val = 0;`" after statement 54, and releases a new version `Dispenser`′ of the component. When we integrate `Dispenser`′ into `VendingMachine`, we want to regression test the resulting application. For efficiency, we want to rerun only those test cases in our test suite that exercise modifications from `Dispenser` to `Dispenser`′. However, without information about the modifications to `Dispenser` and how they relate to our test suite, we are forced to run all or most of the test cases in the test suite. At a minimum, we must select all test cases in the test suite that exercise the component (20 of the 25 test cases).

2.1 Code-based Selection Metacontents

We now illustrate a metacontent-based technique for regression test selection for use with code-based testing techniques. 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 cri-

Table 2. Edge coverage for `VendingMachine`.

Test Case #	Relevant Edges Covered
1	(9,10), (20,21), (21,22)
2	(9,10), (28,29)
3	(9.10), (15,16), (20,21), (21,23)
4	(9,10), (15,16), (28,29)
5	(9,10), (15,16), (20,21), (21,23)
6	(9,10), (15,16), (28,29)
7	(9,10), (15,16), (20,21), (21,23)
8	(9,10), (15,16), (28,29)
9	(9,10), (15,16), (20,21), (21,23)
10	(9,10), (15,16), (28,29)
11	(9,10), (15,16), (20,21), (21,23), (28,29)
12	(9,10), (15,16), (28,29)
13	(9,10), (15,16), (20,21), (21,23), (28,29)
14	(9,10), (15,16), (28,29)
15	(9,10), (15,16), (20,21), (21,23), (28,29)
16	(9,10), (15,16), (28,29)
17	(9,10), (28,29)
18	(9,10), (15,16), (28,29)
19	(9,10), (15,16), (20,21), (21,23), (28,29)
20	(9,10), (15,16), (28,29)
21	(9,10), (28,29)
22	(9,10), (15,16), (28,29)
23	(9,10), (15,16), (28,29)
24	(9,10), (15,16), (28,29)
25	(9,10), (15,16), (28,29)

terion 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 traversed by each test case in the test suite T. *Relevant edges* are method entries (edges (9,10), (15,16), (20,21), and (28,29) for `VendingMachine`) and edges from decision statements (edges (21,22) and (21,23) for `VendingMachine`).[4] With this information, we can associate the relevant edges in P, the program under test, with each test case in T. Table 2 shows the relevant edges in `VendingMachine` exercised by each of our test cases.

Code-based regression test selection techniques (e.g., [7, 10, 18, 20, 25]) construct some representation, such as a control-flow graph, a call graph, or a 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 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.[5]

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 [18]. In particular, we consider a specific implementation of Rothermel and Harrold's approach: the DEJAVU tool. DEJAVU uses a control-flow graph as the representation, and the entities are the edges in the graph. To select test cases to be rerun, DEJAVU 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$'$, and selects the test cases that cover such edges as the test cases to be rerun.

For example, to perform regression test selection on application `VendingMachine` when component `Dispenser` is changed to `Dispenser`$'$, DEJAVU constructs a control-flow graph CFG$'$ for `VendingMachine`$'$. However, because the code for `Dispenser` is unavailable to the developer of `VendingMachine`, DEJAVU cannot construct control-flow graphs for any of the methods in `Dispenser`. Therefore, DEJAVU can only select test cases based on the analysis of CFG and CFG$'$ for `VendingMachine` by conservatively considering each edge that represents a call to component `Dispenser` to be modified. In this case, when DEJAVU performs its synchronous traversal of CFG and CFG$'$, it finds that edge (28,29) is affected by the change, and selects all test cases that exercise this edge—{2,4,6,8,10–25}.

To achieve better regression test selection when the source code of the component is unavailable, we can use component metacontents. To support test selection for code-based regression testing, we need three types of metacontents 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 could then construct, for example, a metacontent-aware version DEJAVU$_{MA}$ of DEJAVU. This tool would build the matrix "test cases"–"edges covered" by gathering the component coverage data for each test case. According to the framework presented in Reference [15], a possible interaction of DEJAVU$_{MA}$ with component c for incrementally populating the matrix "test cases"–"edges covered" could consist of the following steps:[6]

[4]Given the coverage of relevant edges, we can infer coverage of all other edges in the program. For example, coverage of edges (29,30), (30,31), (31,32), and (32,33) is implied by coverage of (relevant) edge (28,29).

[5]Regression test selection selects test cases from the original test suite for use in testing the modified program P'. However, modified or new code from P to P' may not be exercised by test cases in T. In this case, the test suite must be augmented by developing new test cases that cover these unexercised parts of the program.

[6]This scenario assumes the existence of some hierarchical scheme for naming and accessing available metacontents, as described in [15].

1. Get the list of types of coverage metacontents provided by the component:
   List lmd = c.getMetacontents("analysis/dynamic/coverage")
2. Check whether lmd contains the metacontent needed (i.e., "analysis/dynamic/coverage/edge"); assume that it does.
3. Get information on how to access the metadata through metamethods:
   MetacontentUsage mu = c.getMetacontentUsage("analysis/dynamic/coverage/edge")
4. Based on information in mu, fetch the coverage metadata by first enabling the built-in coverage facilities:
   c.enableCoverage("analysis/dynamic/coverage/edge")
5. At this point, the built-in coverage facilities provided with component c are enabled, so we can start producing coverage information; for each test case t in the test suite:
   - Reset the built-in coverage to get the coverage for t:
     c.resetCoverage("analysis/dynamic/coverage/edge")
   - Run test case t
   - Get the coverage for t:
     Metadatum md = getCoverage("analysis/dynamic/coverage/edge")

Now, when `Dispenser`$'$ is acquired, DEJAVU$_{MA}$ (1) retrieves from `Dispenser` its version, (2) using this information, queries `Dispenser`$'$ about which edges are affected by the changes between it and `Dispenser`, and (3) selects the test cases to rerun, based on the affected edges and the matrix. In this case, the differences between `Dispenser` and `Dispenser`$'$ affect only edge (53,54), which is exercised only by test cases 4 and 14. Therefore, only test cases 4 and 14 are selected to be rerun, which is a substantial savings over the approach that does not use metacontents.

The technique for code-based regression testing that we have just illustrated is defined at the edge level. When the size of the code increases, the edge-level approach may become impractical. However, the technique can be defined at different levels of granularity. In particular, possible alternatives are to define the technique at the method level, at the class level, or at the subsystem level. In such cases, both the coverage and change information provided through metamethods would be defined at the method, class, or subsystem levels, respectively, rather than at the edge level. In our experiments, we used the method-level approach, as described in Section 3.

2.2 Specification-based Selection Metacontents

We now illustrate a second metacontent-based technique for regression test selection, defined for a specification-based approach. Specification-based testing techniques develop test cases based on a functional description of the system. One such technique, the *category-partition method* [16], produces *test frames* that represent a test specification for the functional units in the system. The technique is composed of several phases. In the first phase, the tester analyzes the specification to identify the individual functional units in the system; for each unit, the tester identifies parameters (inputs to the unit) and environment factors (elements outside of the code that affect the behavior of the unit). In the second phase, the tester partitions each parameter and environment entity into mutually exclusive choices. In the third phase, the tester identifies constraints among choices, based on their mutual interactions. Finally, in the fourth phase, the tester develops a set of test frames for each unit by computing the cross-product of the different choices; in this phase, the constraints among choices are used to eliminate meaningless or contradictory combinations and to reduce the number of frames, possibly through re-iteration of the third phase.

Analogous to code-based regression test selection techniques, specification-based techniques record the coverage of the original test suite T with respect to entities in the functional representation. In the case of the category-partition method, we can consider the test frames as the entities to be covered. A test case in T covers a test frame *tf* if (1) the parameters of calls to single functionalities match the corresponding choice in *tf*, and (2) the state of the component matches the environment characteristics in *tf*. To compute the coverage of the component achieved by a given test case in terms of test frames, the code must be instrumented according to the identified test frames. In this way, for each test case in T we can identify the test frames that it covers. Therefore, we are able to associate a subset of T with each test frame. This information can be used when performing regression testing of component P'. If we know which frames are affected by the changes, then we can rerun only the test cases associated with such frames. Each test frame identifies a family of test cases that satisfy it. Such test cases, in turn, identify a family of paths within the component—the paths traversed by the execution of the test cases. These paths can therefore be associated with the frame. We say that a change *affects* a test frame *tf* if at least one of the paths associated with *tf* traverses a statement either changed or eliminated in the new version of the component. The component developer can, based both on analysis of the component and on his or her knowledge, identify which frames are affected by the changes between two versions of a given component.

Figure 2 illustrates, for method `dispense`, a possible set of categories, choices, and constraints on the choices derived by applying the category-partition method to the component `Dispenser`. Figure 3 shows a set of test frames derived from the test specifications in Figure 2.

For the specification-based approach, to perform regression test selection when `Dispenser` is modified, we need to (1) know the test frames for the component, (2) have a way of computing which test cases for application `VendingMachine` cover which test frames of component `Dispenser`, and (3) have information about the test frames *functionality* **dispense**

- Params:
  credit
  - zero [if Available]
  - insufficient [if Available]
  - sufficient [if Available]
  - over [if Available]

  selection
  - correct [property Correct]
  - incorrect [error]
- Environment
  availability
  - available [if Correct] [property Available]
  - unavailable [if Correct] [error]

Figure 2. A possible set of categories, choices, and constraints for the component $Dispenser$.

functionality **dispense**

1 selection: incorrect, availability: X, credit: X
2 selection: correct, availability: unavailable, credit: X
3 selection: correct, availability: available, credit: zero
4 selection: correct, availability: available, credit: insufficient
5 selection: correct, availability: available, credit: sufficient
6 selection: correct, availability: available, credit: over

Figure 3. Test frames for component `Dispenser` (value "X" indicates "don't care" values).

affected by the changes in the component. When a specification for the component is available, we can define test frames for the component. However, because the code for `Dispenser` is unavailable, we cannot compute the coverage information (because we need access to the state of the component to check which environmental conditions are satisfied by each test) and cannot identify which test frames are affected by the changes in `Dispenser`.

Therefore, to support test selection for specification-based regression testing, we need three types of metacontents. First, we need to know the coverage achieved by the test suite with respect to the test frames for the component, so that we can associate test cases with frames. Second, we need to know the component version. Third, we need a way to query the component to retrieve the test frames affected by the changes in the component between two versions. Again, the component developer will provide this information in the form of metadata and metamethods, packaged with the component.

We could now construct a metacontent-aware tool, analogous to the DEJAVU$_{MA}$ tool of Section 2.1, that would build the matrix "test cases"–"test frames covered" by enabling the coverage computation and gathering the data for each test case. Like DEJAVU$_{MA}$, this tool would be based on the framework presented in Reference [15], and could consists of the following steps:

1. Get the list of types of coverage metacontents provided by the component:
   List lmd = c.getMetacontents("analysis/dynamic/coverage")
2. Check whether lmd contains the metacontent needed (i.e., "analysis/dynamic/coverage/testframes"); assume that it does.
3. Get information on how to access the metadata through metamethods:
   MetacontentUsage mu = c.getMetacontentUsage("analysis/dynamic/coverage/testframes")
4. Based on information in mu, fetch the coverage metadata by first enabling the built-in coverage facilities:
   c.enableCoverage("analysis/dynamic/coverage/testframes")
5. At this point, the built-in coverage facilities provided with component c are enabled, so we can start producing coverage information; for each test case t in the test suite:
   - Reset the built-in coverage facilities to get the coverage for t:
     c.resetCoverage("analysis/dynamic/coverage/testframes")
   - Run test case t
   - Get the coverage for t:
     Metadatum md = getCoverage("analysis/dynamic/coverage/testframes")

Table 3. Test frames for the `Dispenser` component covered by the test cases for the vending machine.

Test Case #	Test Frames Covered	Test Case #	Test Frames Covered
1		14	4, 6
2	3	15	3
3		16	3, 6
4	4	17	2
5		18	2
6	5	19	2
7		20	2
8	6	21	1
9		22	1
10	6	23	1
11	3	24	1
12	3, 5	25	1
13	3		

When a new version of the component is acquired, the tool (1) gathers the metadatum about the version from the old component, (2) using this information queries the new component for the test frames affected by the changes between its version and the version currently in the system, and (3) selects the test cases to rerun, based on the affected frames and the matrix.

Suppose we apply this technique to the `Vending-Machine` example. First, we run the 25 test cases for the application and gather the test-frame coverage information; Table 3 shows how the different test frames in Figure 3 are covered by the test cases for the vending machine. Second, when we acquire `Dispenser`$'$, we determine which test frames are affected by the changes between the two versions of the component; we discover that only test frame 4 is affected. Finally, we use the matrix "test cases"–"test frames covered" to select the test cases to be rerun; according to the information in Figure 3, we select test cases 4 and 14.

As with the code-based approach, the specification-based approach provides a meaningful reduction in the number of test cases to be rerun for the new version of `VendingMachine`.

3 Case Study

To investigate whether the use of metacontents can benefit regression test selection of applications built with external components, we performed a case study. Specifically, we investigated the following research question:

> Let A be a program created by an application developer using a set of externally-developed components C. Let T be a test suite created to test A. Suppose a new version C' of C is created through modifications to one or more of the components in C, and suppose the developer of A then wishes to adopt C' for use in A. If metacontents are available with C and C', can the developer reuse T to regression test A more efficiently than if metacontents are not available?

In this study, we restricted our attention to the use of metacontents for code-based regression test selection techniques, as described in Section 2, and we focused on two specific regression test selection techniques:

No metacontents. The developer of A knows only that one or more of the components in C have been modified, but not which. Therefore, to selectively retest A safely, the developer must rerun any test case in T that exercises code in one or more of the components in C. We refer to this as the NOMETA technique.

Metacontents for method-level regression test selection. The developer of A possesses metacontents provided by the developer of C, sufficient to support selection of test cases that exercise methods changed in producing C' from C using the procedure described in Section 2. We refer to this as the META technique.

3.1 Measures

Regression test selection techniques achieve savings by reducing the effort required to regression test a modified program. Thus, one method used to compare such techniques [2] is to measure and compare the degrees to which the techniques reduce test suite size for given modified versions of a program. We adopt this approach. For each regression test selection technique R that we consider, and for each (version, subsequent-version) pair (P_i,P_{i+1}) of program P, where P_i is tested by test suite T, we measure the percentage of T selected by R to test P_{i+1}.

3.2 Study Subject

As a subject for our study we used several versions of the Java implementation of the SIENA server [6]. SIENA (Scalable Internet Event Notification Architecture) is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks, responsible for selecting notifications that are of interest to clients (as expressed in client subscriptions) and then delivering those notifications to the clients via access points.

To investigate the effects of using component metacontents for regression test selection, we required an application program that was constructed using external components. SIENA is logically divided into a set of six components (consisting of nine classes of about 1.5KLOC), which constitute "a set of external components C," and a set of 17 other classes of about 2KLOC, which constitute an application that could be constructed using C.

We obtained the source code for all the different versions of SIENA, from its first to its last release (about 15 different releases), in the form of an RCS repository. From the SIENA repository, we extracted eight different sequentially released versions of C (versions 1.8 through 1.15), which we refer to as $C_1, C_2, \ldots, C_8$, respectively. Each version provides enhanced functionality or bug fixes over the previous version. The net effect of this process was the provision of eight successive versions of SIENA, $A_1, A_2, \ldots, A_8$, constructed using $C_1, C_2, \ldots, C_8$, respectively. These versions of SIENA represent a succession of versions, each of which the developer of A would want to retest. The pairs of versions (A_k, A_{k+1}), $1 \leq k \leq 7$, formed the (version, modified-version) pairs for our study.

To investigate the impact of metacontents on regression test selection we also required a test suite for our base version A_1 of SIENA that could be reused in retesting subsequent versions. Such a test suite did not already exist for the SIENA release we considered, so we created one. To provide test cases in an unbiased manner, one of the authors of this paper, who is involved in defining the requirements and design of SIENA but is unfamiliar with its implementation details, independently created a black-box test suite, based on the functionality of SIENA, that consists of 138 test cases. This set of test cases served as the subject regression test suite for our study.

3.3 Procedure

Because the creation of metamethods and support tools for directly implementing our target techniques would be expensive, our goal was to discover a way to address our research question without creating such infrastructure. We designed a procedure by which we could determine precisely, for a given test suite and (program, modified-version) pair, which test cases would be selected by our two target techniques. For each (program, modified-version) pair

(P_i, P_{i+1}), we used the Unix `diff` utility and inspection of the code to locate differences between P_i and P_{i+1}, including modified, new, and deleted code. In cases where variable or type declarations differed, we found the methods in which those variables or types were used, and treated those methods as if they had been modified. We used this information to determine the methods in P_i that would be reported changed for the META technique. We instrumented each such method so that, when executed, the method outputs the text "selected", and we then constructed an executable of the application from this instrumented code.

Given this procedure, to determine which test cases in T would be selected by the META technique for (P_i, P_{i+1}) it was sufficient to execute all test cases in T on our instrumented version of P_i, and record which test cases caused P_i to output (one or more times) the text "selected". By construction, these are exactly the test cases that would be selected by an implementation of the META technique.

Determining the test cases that would be selected by the NOMETA technique required a similar, but simpler approach. We instrumented the application developer's portion of the code for P, inserting code that outputs "selected" prior to any invocation of any method in C, and then executed the test cases in T on that instrumented version.

The foregoing procedures require us to execute all test cases in T to determine which test cases would be selected by an actual regression test selection tool; thus, the approaches are of use only for experimentation. However, the approaches let us determine exactly the test cases that would be selected by the techniques.

We applied this approach to each of the seven (program, modified-version) pairs of the SIENA system with our given test suite, and recorded, for each of the two regression test selection techniques, the percentage of the test suite selected by that technique for that (program, modified-version) pair. These percentages served as the data set for our analysis.

3.4 Results and Discussion

Figure 4 depicts the test selection results obtained in this study. In the graph, each modified version of SIENA occupies a position along the x-axis, and the test selection data for that version are represented by a vertical bar, black for the NOMETA technique and grey for the META technique. The height of the bar depicts the percentage of tests selected by the technique on that version.

As the figure shows, the NOMETA technique always selected 97% of the test cases. Only 3% of the test cases do not exercise components in C (the set of external components), and thus all others must be re-executed. Also, because the NOMETA technique selects all test cases that executed any components in C, and the test cases in our test suite that encounter C do not vary across versions, the NOMETA technique selected the same test cases for each version.

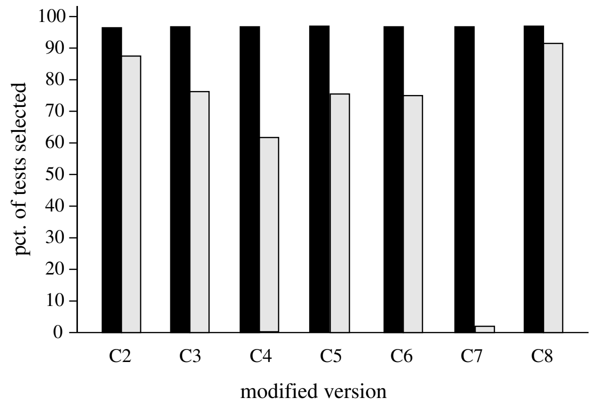

Figure 4. Test selection results for the NOMETA (black) and META (grey) techniques.

As the figure also shows, the META technique always selected a smaller subset of the test suite than the NOMETA technique. In the case of version C7, the difference was extreme: the META technique selected only 1.5% of the test cases in the test suite, whereas the NOMETA technique selected 97% of the test cases. This large difference arose because the changes within C7 are minor, involving few methods, and methods encountered by only a few test cases. On the other versions, differences in selection were more modest, ranging from 6% to 37% of the test suite.

On versions C3, C5 and C6, the META technique selected identical test cases, even though the code changes in those versions differed. This occurred because the code changes involved the same sets of methods. Theoretically, if we had used edge-level regression test selection, the test cases selected for these versions could have differed.

The fact that a regression test selection technique reduces the number of test cases that must be run does not guarantee that the technique will be cost-effective. That is, even though we reduce the number of test cases that need to be rerun, if this does not produce savings in testing time, the reduction in number of test cases will be meaningless. Moreover, savings in testing time might not be proportional to savings in number of test cases (if, for example, the test cases excluded are all inexpensive, while those not excluded are expensive). (See [11] for an applicable cost model.)

In the absence of implementations of the testing techniques and measurements of analysis costs, we cannot determine such savings precisely for this case study; however, we can still gain some insights by considering test execution times. Thus, we recorded execution times for the test cases selected by each technique on each version. Table 4 shows these times.

The table shows, for each version, the minutes and seconds required to test that version. The columns show the

Version	NOMETA Test Execution Time	META Test Execution Time
C2	19:44	18:45
C3	19:51	16:57
C4	19:51	13:07
C5	19:52	17:44
C6	19:52	16:40
C7	19:51	00:15
C8	19:49	19:26
average	19:50	14:07
total	138:50	102:54

Table 4. Execution times (minutes:seconds) for test cases selected by the NOMETA and META techniques.

version number, the time required to run the test cases selected by the NOMETA technique, and the time required to run the test cases selected by the META technique. The last two rows show average and total times. On average over the seven modified versions, the META technique reduced testing time from 19 minutes and 50 seconds to 14 minutes and 7 seconds. The total time savings over the sequence of seven versions was 35 minutes and 59 seconds (26% of total time.) In the worst case, for version C8, the META technique saved only 23 seconds (2%) of testing time. In the best case, for version C7, it saved 19 minutes and 36 seconds (99%) of testing time.

Note that these times do not factor in the cost of the analysis required to perform test selection, but in other studies of test selection, those costs have been shown to be quite low [19]. Furthermore, these times include only the times required to execute, and not validate test cases; validation would further inflate the times, and increase the savings.

Of course, savings of a few minutes and seconds, such as those exhibited in the differences in testing time seen in this study, may be unimportant. In practice, however, regression testing can require hours, days, or even weeks of effort, and much of this effort may be human-intensive. If results such as those demonstrated by this study scale up, a savings of 26% of the overall testing effort for a sequence of seven releases may be substantial, and a savings of 99% of the testing effort for a version may be a huge win. These results thus provide evidence that testing with metacontents could save significant costs and that metacontents could be useful for regression test selection in component-based software.

3.5 Limitations of this Study

Like any empirical study, this study has limitations. We have considered the application of only two regression test selection techniques to a single program and test suite and seven subsequent modified versions of the components that make up that program. Furthermore, we have considered only one measure of test selection effectiveness: percentage reduction in test suite size (although we have buttressed this measure by also considering test execution cost data). Other costs, such as the cost of providing metacontents and performing test selection, may be important in practice.

On the other hand, the program and modified versions we used are derived from an actual implementation, and our specification-based test suite represents a test suite that could be used in practice. Furthermore, previous work [2, 19] has illustrated the applicability of our cost measure. Our results thus support an "existence argument": Cases exist in which metacontents can produce benefits in regression testing. Thus, these results motivate further research, and the implementation of tools to support the techniques, followed by carefully controlled experimentation, to investigate whether such results will generalize.

4 Conclusion

We have introduced two new techniques for regression testing of component-based applications. The first technique is code-based, and the second technique is specification-based. Both techniques are based on the use of metadata and metamethods to package additional information together with a component. The presence of metacontents 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 be provided for the techniques to be applicable.

To assess the applicability and potential effectiveness of the proposed techniques in practice, we have presented a case study performed on a real system using the code-based approach. Although there are some limitations to the results of our study, the study does show that cases exist in which the use of metacontents can reduce the costs of regression testing component-based applications. In particular, our code-based technique resulted in an average savings of 26% of the testing effort over seven subsequent releases of the considered set of components, with a maximum saving of 99% of the testing effort for one of the versions.

Because of these promising initial results, we plan to perform further research on the use of metacontents for regression testing. Our first goal is to build a set of tools that allow us to automate the application of the presented techniques and to integrate them into the ARISTOTLE analysis system [1]. As a first step in this direction, we are currently developing DEJAVU$_{MA}$, the metacontent-aware version of the DEJAVU tool. In this way, we will be able to run extensive experiments to further validate the code-based approach. In parallel, we will study the applicability of the specification-based approach on real examples. Finally, we will study other applications of component metacontents and their effectiveness for software engineering tasks.

Acknowledgements

Antonio Carzaniga provided the source code for the Siena system and helped with its installation. This work was supported in part by grants from Boeing, and by NSF awards EIA-0196145 and CCR-0096321 to Georgia Institute of Technology, by NSF Award CCR-9703108 to Oregon State University, by NSF Award CCR-9808590 to the University of Pittsburgh, and by NSF Award CCR-9701973 to UC Irvine. The work was also supported by the ESPRIT Project TWO (EP n.28940), by the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO Project. This effort was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061 to UC Irvine.

References

[1] Aristotle Research Group. ARISTOTLE: Software engineering tools. `http://www.cc.gatech.edu/aristotle/`, 2000.

[2] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):149–183, Apr. 2001.

[3] P. Brereton and D. Budgen. Component-Based Systems: A Classification of Issues. *IEEE Computer*, 33(11):54–52, November 2000.

[4] N. Brown and C. Kindel. *Distributed Component Object Model protocol: DCOM/1.0*. January 1998.

[5] C. Canal, L. Fuentes, J. Troya, and A. Vallecillo. Extending CORBA interfaces with p-calculus for protocol compatibility. In *Technology of Object-Oriented Languages and Systems (TOOLS'00)*, pages 208–225, June 2000.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[7] Y. Chen, D. Rosenblum, and K. Vo. TestTube: A system for selective regression testing. In *16th Int'l. Conf. Softw. Eng.*, pages 211–222, May 1994.

[8] R. Cherinka, C. M. Overstreet, and J. Ricci. Maintaining a COTS integrated solution — Are traditional static analysis techniques sufficient for this new programming methodology? In *Int'l. Conf. Softw. Maint.*, pages 160–169, November 1998.

[9] Enterprise JavaBeans technology. `http://java.sun.com/products/ejb/index.html`, October 2000.

[10] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*. ACM Press, October 2001. (to appear).

[11] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Conf. Softw. Maint.*, pages 201–208, October 1991.

[12] U. Lindquist and E. Jonsson. A map of security risks associated with using COTS. *IEEE Computer*, 31(6):pages 60–66, June 1998.

[13] P. M. Maurer. Components: What if they gave a revolution and nobody came. *IEEE Computer*, 33(6):28–34, June 2000.

[14] Microsoft .NET Platform. `http://www.microsoft.com/net/`, February 2001.

[15] A. Orso, M. J. Harrold, and D. S. Rosenblum. Component metadata for software engineering tasks. In W. Emmerich and S. Tai, editors, *EDO '00*, volume 1999 of *Lecture Notes in Computer Science*, pages 126–140. Springer-Verlag / ACM Press, November 2000.

[16] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Comm. ACM*, 31(6), June 1988.

[17] G. Piccinelli and S. Lynden. Concept and tools for e-service development. In *7th Workshop HP Openview Univesity Association (OVUA'00)*, June 2000.

[18] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng.and Meth.*, 6(2):173–210, April 1997.

[19] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. Softw. Eng.*, 24(6):401–419, June 1998.

[20] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *J. Softw. Testing, Verif., and Rel.*, 10(2), June 2000.

[21] C. Szyperski. *Component Oriented Programming*. Addison-Wesley, first edition, 1997.

[22] J. Troya and A. Vallecillo. On the addition of properties to components. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology: ECOOP'97 Workshop Reader*, volume 1357 of *Lecture Notes in Computer Science*, pages 374–378. Springer, 1997.

[23] J. Voas. The challenges of using COTS software in component-based development. *IEEE Computer*, 31(6):44–45, June 1998.

[24] E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, Sept–Oct 1998.

[25] L. White and H. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Conf. Softw. Maint.*, pages 262–270, November 1992.

[26] XOTcl - extended object Tcl. `http://nestroy.wi-inf.uni-essen.de/xotcl/`, November 2000.