Using Component Metadata to Regression Test Component-Based Software [*]

Alessandro Orso
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
orso@cc.gatech.edu

Hyunsook Do
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
dohy@cse.unl.edu

Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
grother@cse.unl.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
harrold@cc.gatech.edu

David S. Rosenblum
London Software Systems
Department of Computer Science
University College London
Gower Street
London, United Kingdom
d.rosenblum@cs.ucl.ac.uk

Abstract

Increasingly, modern-day software systems are being built by combining externally-developed software components with application-specific code. For such systems, existing program-analysis-based software-engineering techniques may not directly apply, due to lack of information about components. To address this problem, the use of component metadata has been proposed. Component metadata are metadata and metamethods provided with components, that retrieve or calculate information about those components. In particular, two component-metadata-based approaches for regression test selection are described: one using code-based component metadata and the other using specification-based component metadata. The results of empirical studies that illustrate the potential of these techniques to provide savings in re-testing effort are provided.

Keywords: component-based software, component metadata, regression testing, regression test selection

[*]Based in part on "Using component metacontents to support the regression testing of component-based software" by A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do, which appeared in Proceedings of the International Conference on Software Maintenance, Florence, Italy, November, 2001, pages 716-725 [1]. Copyright 2001 IEEE.

1 Introduction

Component-based software technologies are increasingly viewed as essential for managing the growing complexity of software systems [2, 3]. Using component-based technologies, software engineers build systems by integrating independently developed software components with application-specific code.

Component-based engineering provides many advantages [3], but it may also complicate software engineering tasks. For example, the use of components complicates system evolution [4] and threatens the ability to validate software [5, 6]. In part, these problems arise because application developers lack information about the components they utilize. Component developers are often reluctant to provide source code for components due to intellectual property issues. Without source code, many types of information that could be used to validate and maintain software, such as dependence information, cannot be calculated. Furthermore, even if source code is available, computing some kinds of information may require complex analyses or analysis tools not available to component users. Finally, some types of information that engineers might wish to use, such as specifications, cannot easily be derived from source code alone.

This article investigates an alternative method for supporting software-engineering tasks on component-based applications, based on the concept of component metadata. *Component metadata* can be either *metadata* about components, or *metamethods*, associated with those components, that retrieve or calculate metadata. In general, metadata consist of various forms of static data, such as data or control dependencies, abstract representations of source code, or complexity metrics, and metamethods compute or retrieve such information, as well as collect dynamic information such as execution traces or assertions about security properties. As such, component metadata can conceivably support a wide range of software-engineering tools and tasks that depend on such data.

This article explores the application of component metadata to one particular testing and maintenance problem: the problem of regression testing an application that uses components after those components have been modified (modifications include both code changes and interface changes in components). As software is maintained, software engineers regression test it to validate new features and detect whether corrections and new features have introduced new faults into previously tested code. Such regression testing plays an integral role in maintaining the quality of subsequent releases of software, but it is also expensive, accounting for a large proportion of the costs of maintenance [7, 8]. For this reason, many researchers have addressed regression testing problems and proposed various techniques for improving the cost-effectiveness of regression testing (e.g., [9, 10, 11, 12, 13, 7, 8, 14, 15, 16]). Little of this work, however, has addressed the growing need to provide methods for regression testing component-based software.

One particular regression testing technique, *regression test selection* [8, 17], involves selecting tests from an existing test suite for use in revalidating a modified version of a software system. This technique is the focus here, as framed by the following research question:

> Given an application A that uses a set of externally-developed components C, and has been tested with a test suite T, and given a new version of this set of components C', is it possible to exploit component metadata to select a test suite $T' \subseteq T$ that can reveal possible regression faults in A due to changes in C' and that does not exclude test cases impacted by such changes?

In practice, the many different business and engineering models under which components are developed, distributed, and employed cause component-based systems to be built from components provided with varying forms of information, ranging from source code to byte code to various forms of specifications retrievable through various means, such as documentation or reflection (a mechanism that allows for inspecting a component's structure and interface at run-time). Furthermore, in practice, different types of testing techniques have different, and often complementary, strengths and weaknesses. No single technique will be most appropriate in all situations. Thus, two different types of component-metadata-based techniques for performing regression test selection on component-based software are investigated, that differ in terms of the types of information they utilize. The first type of technique involves code-based regression test selection, based on statement-level, method-level, and component-level regression test selection algorithms

[18, 11, 12, 14, 19, 20]. The second type of technique involves specification-based regression test selection, based on a statechart diagram representation of components [21, 22, 23].

The next section of this article provides further background information and motivation relevant to the study of component metadata. Section 3 presents a general framework for supporting the use of component metadata. Sections 4 and 5 describe specific component-metadata-based regression test selection techniques, together with empirical results investigating their application. Finally, Section 6 concludes and discusses future work.

2 Background and Motivation

This section provides background information on related research on metadata, potential uses of metadata and basic regression testing techniques and terminology.

2.1 Related Work on Component Metadata

The notion of providing information with components is not new. Existing component standards and environments, including DCOM [24], Enterprise JavaBeans [25], and .NET Common Language Runtime (CLR) [26] already supply some information through data packaged with components and through reflection. (*Reflection*, also called *introspection*, lets users gather information about a component's structure and interface at run-time, and use that information to interact with the component [27].) The information supplied so far, however, is typically restricted to uses in compile-time and run-time type-checking (e.g., the name of the component's class, the names of its functions, or the types of the functions' parameters), or design-time customization (e.g., the shape or color of a graphical user interface component, or the maximum size of the internal buffer of a data storage component).

Researchers have proposed extending these uses of metadata and reflection for certain specific tasks [28, 29, 30, 31, 32]. However, the varieties of information currently considered address only a limited range of software-engineering problems, such as providing deployment descriptions of components [28], enhancing self-documentation [32], providing information about a component's testing history [29], and identifying components' misuses [30]. None of this previous work has focused on software engineering problems currently addressed, for procedural-language programs, by program-analysis-based tools and techniques, which typically require information on source code. Moreover, none of these existing approaches have explored the possibility of dynamically building metadata at runtime.

2.2 On the Potential Uses of Component Metadata

An examination of the uses of metadata to support program-analysis-based software-engineering tasks, including metadata dynamically constructed at runtime, might provide opportunities for aiding those tasks. Earlier work by the authors of this article in the area of component metadata [1, 33] has suggested several specific ways in which component metadata could be used. This article focusses on the problem of using metadata to regression test evolving component-based systems, and on regression test selection in particular.

A general question relating to component metadata is whether it is realistic to expect component providers to provide metadata with their components. Certainly, the provision of extra data and methods with components would require additional effort on the part of component developers, including potential changes to development processes. There are reasons, however, for believing that component-metadata-based approaches could eventually be adopted in practice.

For example, the requirement that particular data and information be provided with software to aid in its validation is standard practice for high-integrity aviation system software, and companies supplying that software comply with this requirement in order to obtain contracts. Further, if it were demonstrated empirically that provision of component metadata could significantly enhance the quality of applications built from components, component developers might be motivated to provide such metadata as an optional value-added feature for which they could charge a fee, thus enhancing the value of their components. Finally, for many types of component metadata, calculation and provision of relevant metadata and metamethods can be automated, facilitating inclusion of metadata in new components, and easing retrofitting of metadata into existing components.

The applications of component-metadata-based engineering techniques are not limited only to cases in which components are provided to an "external" world; they also apply to cases in which components are shared within an organization. For example, software test engineers could create test-related component metadata to facilitate their organizations' testing efforts, just as they currently create test plans, drivers, and stubs. Eventually, the availability of quality-enhancing mechanisms could change business and engineering practices, just as, in the past, adoption of new language paradigms has changed development practices.

Before any of these potential benefits can be realized, however, it must be determined whether component-metadata-based engineering processes could, if employed, support effective engineering techniques, and this is the goal of the research detailed in this article.

2.3 Regression Testing and Regression Test Selection

This section briefly describes the regression testing and regression test selection activities that are the focus in this article; a more substantive description is presented by Rothermel and Harrold [17].

Let P be a program, let P' be a modified version of P, and let T be a test suite developed for P. Regression testing attempts to validate P'. To facilitate regression testing, test engineers typically attempt to re-use T to the maximum extent possible. However, rerunning all the test cases in T can be expensive, and when only small portions of P have been modified, may involve unnecessary work.

Regression test selection (RTS) techniques attempt to reduce unnecessary regression testing, increasing the efficiency of revalidation. RTS techniques (e.g., [9, 10, 11, 34, 12, 35, 14, 36, 16, 20]) use information about P, P', and T to select a subset of T with which to test P'. Most of these techniques are code-based, using information about code changes to guide the test selection process. A few techniques [10, 16], however, are specification-based, relying on some form of specification instead of code. (Rothermel and Harrold [17] survey RTS techniques.) Empirical studies [11, 37, 19, 38] have shown that these techniques can be cost-effective.

One important facet of RTS techniques involves *safety*. *Safe* RTS techniques (e.g., [11, 12, 14, 36]) guarantee that, given that certain preconditions are met, test cases not selected could not have exposed faults in P' [17]. Informally, these preconditions require that: (1) the test cases in T are expected to produce the same outputs on P' as they did on P; i.e., the specifications for these test cases have not changed; and (2) test cases can be executed deterministically, holding all factors that might influence test behavior constant with respect to their states when P was tested with T. This notion of safety is defined formally, and these preconditions are expressed more precisely by Rothermel and Harrold [17].

Two other facets of RTS techniques involve *precision* and *efficiency*. Precision concerns the extent to which techniques correctly deduce that specific test cases need not be re-executed. Efficiency concerns the cost of collecting the data necessary to execute an RTS technique, and the cost of executing that technique. RTS techniques that operate at different levels of granularity — for example, analyzing code and test coverage at the level of functions rather than statements — exploit tradeoffs between precision and efficiency, and their relative cost-benefits vary with characteristics of programs, modifications, and test suites [18].

Finally, note that in regression testing, in general, simply reusing existing test cases is not sufficient; new test cases may also be required to test new functionality. There are also many other regression testing problems that have been addressed in the research literature (see [8, 17]). Many of these problems could also conceivably be addressed through the use of component metadata. In this paper, however, attention is restricted to regression test selection.

3 Component Metadata

Component metadata consists of metadata and metamethods; *metadata* are information about components and *metamethods* are methods, associated with components, that can compute or retrieve metadata.

1. Get the list of types of profiling metadata provided by component c:
   `List lo = c.getMetadata(‘‘analysis/dynamic/profiling’’)`
2. Check whether `lo` contains the metadata needed (e.g., “analysis/dynamic/profiling/method”);
3. Get information on how to gather profiling data:
   `MetadataUsage ou = c.getMetadataUsage(‘‘analysis/dynamic/profiling/method’’)`
4. Based on information in `ou`, gather the profiling data by first enabling built-in coverage facilities:
   `c.enableMetadata(‘‘analysis/dynamic/profiling/method’’)`
5. The built-in profiling facilities provided with `c` are now enabled, execute the application that uses `c`.
6. Disable the built-in coverage facilities:
   `c.disableMetadata(‘‘analysis/dynamic/profiling/method’’)`
7. Get the profiling data:
   `Metadata md = getMetadata(‘‘analysis/dynamic/profiling/method’’)`

Figure 1: Steps for gathering method profiling metadata.

Component metadata can provide a wide range of static and dynamic information about a component, such as coverage information, built-in test cases, abstract representations of source code, or assertions about security properties. Thus, different types of component metadata and different protocols for accessing them can be envisioned. A distinction is made between *a priori* and on-demand metadata. In the first case, *a priori metadata* provide information about a component that can be computed beforehand and attached to the component (e.g., a component's version number). In the second case, *On-demand metadata* provide information that either (1) can be gathered only by execution or analysis of the component in the context of the application (e.g., the execution profile of the component in the component-user's environment), or (2) are too expensive to compute exhaustively *a priori* (e.g., metadata for impact analysis that provide forward slices on all program points in a component).

Like metadata, metamethods are highly dependent on the characteristics of the data involved. Metamethods for *a priori* metadata are in general simple, returning static metadata. Metamethods for on-demand metadata, in contrast, usually involve an interaction protocol between the component user (which could be another component, a tool, or a human being) and the component. Consider, for example, metadata by which self-checking code obtains a log of assertion violations during an execution. In this case, the user may have to invoke a metamethod for enabling assertion checking and logging, a metamethod for disabling assertion checking and logging, and a metamethod for retrieving log information.

In the metadata framework that underlies this work [33], each component provides, in addition to specific metamethods, two generic metamethods that let the user gather information about the metadata available for the component: `getMetadata` and `getMetadataUsage`. Method `getMetadata` provides a list of metadata available for the component, and `getMetadataUsage` provides information on how to gather a given type of metadata. To illustrate, Figure 1 shows a possible interaction with a component to obtain method profiling information.

This example assumes the existence of some hierarchical scheme for naming and accessing available metamethods, as described by Orso et al. [33]. Obviously, many mechanisms for identifying and retrieving information about metadata may be available, and the mechanism shown here is just one possible alternative. In the remainder of this article it is assumed that a mechanism such as this is available, and this mechanism is used to describe metadata-based techniques.

4 Component-Metadata-Based Regression Test Selection

This section shows how metadata can be used to support regression test selection for component-based software. The section first presents an example used to illustrate the approach, and then presents the approach itself.

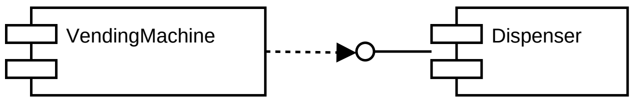

Figure 2: Component diagram for `VendingMachine` and `Dispenser`.

4.1 Example

The example represents a case in which application A uses a component C. (The approaches presented also handle the case in which an application uses multiple components, but this single-component example simplifies the presentation.) The application models a vending machine. A user can `insert` coins into the machine, ask the machine to `cancel` the transaction, which results in the machine returning all the coins inserted and not consumed, or ask the machine to `vend` an item. If an item is not available, a user's credit is insufficient, or a selection is invalid, the machine prints an error message and does not dispense the item, but instead returns any accumulated coins.

Because the techniques presented involve code-based and specification-based testing techniques, both an implementation and a possible specification for the example are provided, expressing the latter using component diagrams and statechart diagrams in UML (Unified Modeling Language [21]) notation.

Figure 2 shows a component diagram that represents application `VendingMachine`, component `Dispenser`, and their interactions. `Dispenser` provides an interface that is used by `VendingMachine`. `VendingMachine` uses the services provided by `Dispenser` to manage credits inserted into the vending machine, validate selections, and check for availability of requested items.

Figure 3 shows a statechart specification of `VendingMachine`. The machine has five states (`NoCoins`, `SingleCoin`, `MultipleCoins`, `ReadyToDispense`, and `Dispensing`), accepts five events (`insert`, `cancel`, `vend`, `nok`, and `ok`), and produces three actions (`setCredit`, `dispense`, and `returnCoins`). If the machine receives an `insert` event while in state `NoCoins`, it reaches state `SingleCoin`, from which an additional `insert` event takes it to state `MultipleCoins`; if the machine receives a `vend` or `cancel` event while in state `NoCoins`, it remains in that state. In states `SingleCoin` and `MultipleCoins`, a `vend` event triggers action `setCredit` (with different parameters in the two cases: 1 and $2..max$, respectively) and brings the machine to state `ReadyToDispense`. (The notation $2..max$ is used to indicate that the value of the parameter can vary between 2 and some predefined maximum.) In state `ReadyToDispense`, the machine produces action `dispense` and enters state `Dispensing`, from which it returns to state `NoCoins` when it receives a `nok` or `ok` event; in both cases, any remaining coins are returned through a `returnCoins` event.

Figure 4 shows a statechart specification of `Dispenser`. The machine has three states (`Empty`, `Insufficient`, and `Enabled`), accepts two events (`setCredit` and `dispense`), and produces two actions (`nok` and `ok`). In state `Empty`, the machine accepts event `setCredit` and stays in state `Empty`, reaches state `Insufficient`, or reaches state `Enabled` based on the value of the credit, as specified by the guards in the figure. In all three states, the machine accepts event `dispense`. Event `dispense` triggers a `nok` or `ok` action depending on the availability of the requested item.

Figure 5 shows a Java implementation of `VendingMachine`. Methods `insert` and `cancel` increment and reset the counter of inserted coins. Method `vend` calls `Dispenser.setCredit` and `Dispenser.dispense`. If the value returned by `Dispenser.dispense` is greater than 0 (which means that the selection is valid, the item is available, and the credit is sufficient), the item is dispensed and the change, if any, is returned to the user; otherwise, an error message is displayed and credit is reset to 0.

Figure 6 shows a Java implementation of `Dispenser`. The code contains an error: after a successful call to `Dispense.dispense` (i.e., a call resulting in the dispensing of the requested item), the value returned to the caller is the actual cost of the item (`COST`), rather than the number of coins consumed (`COST` divided by `COINVALUE`). As a result, `VendingMachine` fails in computing the change to be returned. For example, the sequence of calls: `v.insert()`; `v.insert()`; `v.insert()`; `v.vend(4)`, where `v` is an instance of `VendingMachine`, causes item 4 to be dispensed, but the one-coin change is not returned to the user.

This example is presented solely to illustrate the techniques. Thus, in both the code and the specification,

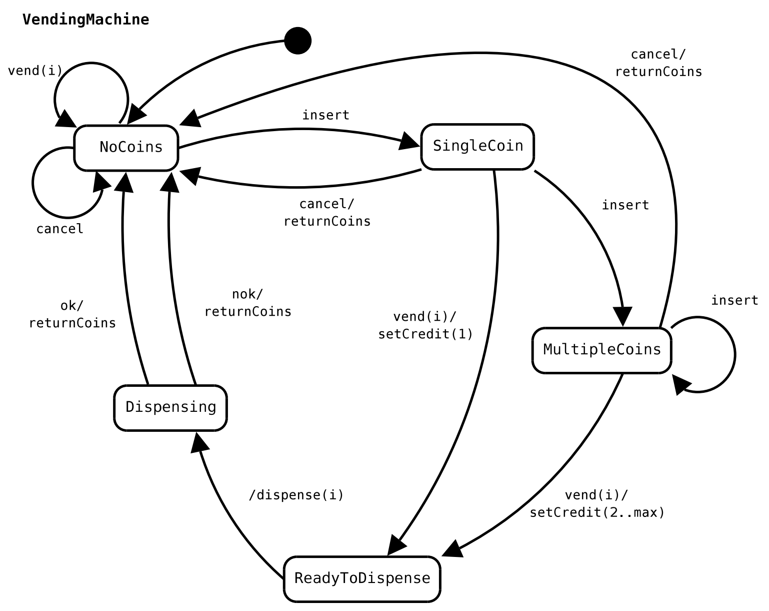

Figure 3: Statechart specification of `VendingMachine`.

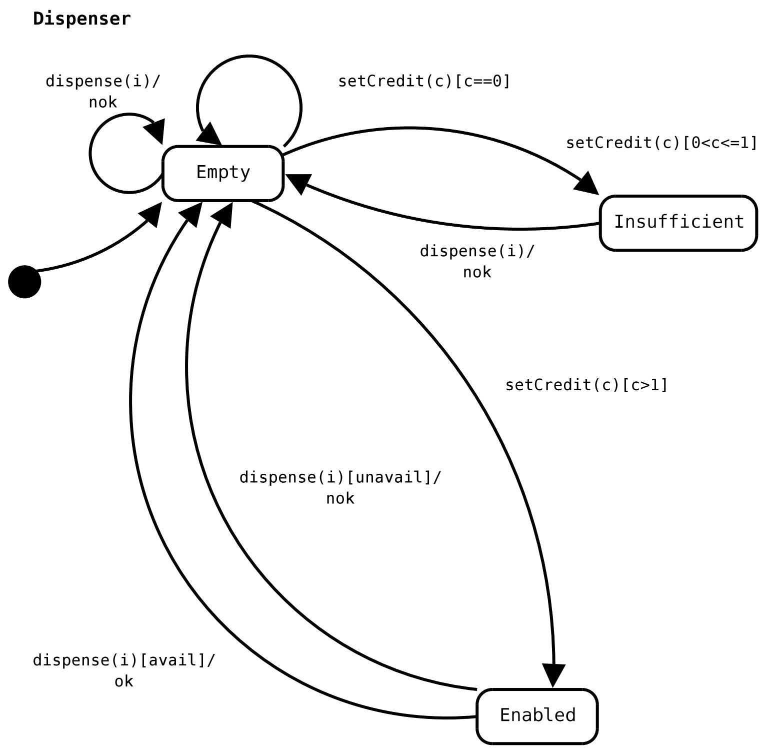

Figure 4: Statechart specification of `Dispenser`.

```
public class VendingMachine {
  private int coins;
  private Dispenser d;

  public VendingMachine() {
    coins = 0;
    d = new Dispenser();
  }
  public void insert() {
    coins++;
    System.out.println("Inserted coins = "+coins );
  }
  public void cancel() {
    if( coins > 0 ) {
      System.out.println("Take your change" );
    }
    coins = 0;
  }
  public void vend( int item ) {
    if( coins == 0 ) {
      System.out.println("Insufficient credit");
      return;
    }
    d.setCredit(coins);
    int result = d.dispense( item );
    if( result > 0 ) {      \\ event OK
      System.out.println("Take your item" );
      coins -= result;
    }
    else switch( result ) { \\ event NOK
      case -1: System.out.println("Invalid selection "+item);
               break;
      case -2: System.out.println("Item "+item+" unavailable");
               break;
      case -3: System.out.println("Insufficient credit");
               break;
    }
    cancel();
  }
} // class VendingMachine
```

Figure 5: Application `VendingMachine`.

```
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() {
    credit = 0;
  }
  public void setCredit(int nOfCoins) {
    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 unavailble
    else if ( credit < COST )
      val = -3; // Insufficient credit
    else {
      val = COST;
      itemsInStock[selection]--;
    }
    credit = 0;
    return val;
  }
} // class Dispenser
```

Figure 6: Component `Dispenser`.

Table 1: A Test Suite for `VendingMachine`.

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

unnecessary details are omitted, and only the parts of the code or specifications that are involved in the interactions between the application and the component are shown. For example, the set of available items is hard-coded in the `Dispenser` component and no method is provided for updating it.

To complete the example, Table 1 shows a test suite for `VendingMachine`, created to cover usage scenarios of `VendingMachine` based on knowledge of its behavior. Each test case in this test suite is a sequence of method calls. For brevity, initial calls to the constructor of class `VendingMachine`, which is implicitly invoked when the class is instantiated, are omitted. The test cases are grouped into three sets (1–14, 15–19, 20–24) based on the value of parameter `selection` that is passed to method `VendingMachine.vend`. The table indicates whether each test case passes or fails. Test cases 6 and 12 fail due to the error in `Dispenser.dispense`.

4.2 Code-Based RTS Using Component-Metadata

The first type of approach to be presented involves code-based RTS techniques. This section begins by describing traditional code-based RTS techniques, and then presents corresponding techniques that utilize metadata.

4.2.1 Code-based Regression Test Selection

Code-based RTS techniques select test cases from an existing test suite based on a coverage goal expressed in terms of some measurable characteristic of the code. There are many such characteristics that can be

Table 2: Branches for `VendingMachine` and `Dispenser`.

	VendingMachine	**Dispenser**
Method Entries	(4,5), (8,9), (12,13), (18,19)	(46,47), (49,50), (55,56)
Branches	(13,14), (13,16), (19,20), (19,23) (25,26), (25,29), (29,30), (29,32) (29,34), (29,37)	(50,51), (50,53), (57,58) (57,59), (59,60), (59,61) (61,62), (61,64)

considered, including statements, branches, paths, methods, and classes.

In particular, for techniques that use branch-coverage information, the program under test is instrumented such that, when it executes, it records which branches (i.e., method entries and outcomes of decision statements) are traversed by each test case in the test suite. Given coverage of branches, coverage of all other edges (i.e., transitions between individual statements)[1] in the program can be inferred. For example, coverage of edge (26,27) in `VendingMachine` (Figure 5) is implied by coverage of branch (25,26). For the example application and component, the branches are shown in Table 2.

Suppose the developer of `VendingMachine` runs the test suite shown in Table 1 on that system, with the faulty version of `Dispenser` incorporated. In this case, test cases 6 and 12 fail. Suppose the component user communicates this failure to the component developer, who fixes the fault by changing statement 64 to "`val = COST / COINVALUE;`", and releases a new version `Dispenser′` of the component.

When the component user integrates `Dispenser′` into `VendingMachine`, it is important to regression test the resulting application. For efficiency, the component user could choose to rerun only those test cases that exercise code modified in changing `Dispenser` to `Dispenser′`. However, without information about the modifications to `Dispenser` and how they relate to the test suite, the component user is forced to run all test cases that exercise the component (20 of the 24 test cases – all except test cases 1, 2, 3, and 7).

Code-based RTS techniques (e.g., [11, 12, 14, 19, 20]) construct a 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 entities (i.e., nodes, branches, or edges) 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 compare the representations for P and P' to select test cases from T for use in testing P'. The selection is based on differences between such representations with respect to the entities considered and on information about which test cases cover the modified entities.

Consider the DEJAVU approach [14], which utilizes control-flow graph representations of the original and modified versions of the program, treating edges in the graph as entities. To select test cases, DEJAVU performs a synchronous traversal of the control-flow graph (CFG) for P and the control-flow graph (CFG′) for P', and identifies *affected edges*, that is, edges that lead to statements that have been added, deleted, or modified from CFG to CFG′. Then, the algorithm uses these affected edges to infer a set of dangerous branches. *Dangerous branches* are branches that control the execution of affected edges (i.e., branches that, if executed, may lead to the execution of affected edges). For example, if edge (20,21) in Figure 5 were identified as an affected edge, branch (19,20) would be the corresponding dangerous branch. Finally, the algorithm selects the test cases in T that cover dangerous branches in P as test cases to be rerun on P'. Following the terminology of Section 2.3, DEJAVU is a *safe* RTS technique. Because code-based RTS techniques consider only changes in code, if there are no such changes, the technique selects no test cases.

When applied to systems built from components for which code is not available, DEJAVU makes conservative approximations. For example, to perform RTS on `VendingMachine` when `Dispenser` is changed to `Dispenser′`, DEJAVU constructs control-flow graphs CFG′ for methods in `VendingMachine′`. However, because the code for `Dispenser` is unavailable to the developer of `VendingMachine`, DEJAVU cannot con-

[1]Technically, edge coverage measures coverage of edges in a program's control-flow graph, hence the use of the term "edge." To simplify the discussion, edges are referred to as if they were in the program, implicitly assuming the mapping of edges from the flow graph to the code.

struct control-flow graphs for the methods in `Dispenser`. Therefore, DEJAVU can select test cases based on the analysis of CFG and CFG′ for `VendingMachine` only by conservatively considering each branch that leads to a call to component `Dispenser` as dangerous. In this case, when DEJAVU performs its synchronous traversal of CFG and CFG′, it identifies branch (19,23) as dangerous because it leads to a call to component `Dispenser`, and selects all test cases that exercise this branch – test cases 4–6 and 8–24. (This result is identical to the result, discussed above, in which the component user selects all test cases that exercise the component. In this context, in fact, DEJAVU is an automated approach to identifying that set of test cases.)

4.2.2 A Component-Metadata-Based Approach

To achieve more precise regression test selection results in situations such as the foregoing, component metadata can be used. To do this, three types of information are required for each component:

1. coverage of the component achieved by the test suite for the application, when the component is tested within the application;
2. the component version;
3. information on the dangerous branches in the component, given the previous and the current version of the component.

The component developer can provide the last two forms of information with the component, retrievable through metamethods. In particular, they can compute the set of dangerous branches using the DEJAVU approach discussed above and provide this set as *a priori* metadata in `Dispenser′`. Coverage information, however, can be provided only as on-demand metadata; it must be collected by the application developer while testing the application. Therefore, the component developer must provide built-in instrumentation facilities with the component, as metamethods packaged with the component. For example, the component can be equipped with an instrumented and an uninstrumented version of each method; a test at the beginning of each method would decide which version to execute (similar to the approach presented by Arnold and Ryder [39]).

Given the foregoing metadata and metamethods, when the component user acquires and wishes to regression test `Dispenser′`, they begin by constructing a coverage table for `Dispenser`, as follows:

1. Verify that coverage metadata are available for `Dispenser`
2. Enable the built-in instrumentation facilities in `Dispenser`
3. For each test case t in T
    (a) run t and gather coverage information
    (b) use coverage information for t to incrementally populate the coverage table
4. Disable the built-in coverage facilities in `Dispenser`

(In cases in which more than one component is involved, the same process would be applied simultaneously for all components of concern.) Table 3 shows the coverage information that would be computed by this approach for component `Dispenser` and the test suite in Table 1.

Given this coverage information, the component user invokes a metadata-aware version of DEJAVU, DEJAVU$_{MB}$, on their application. DEJAVU$_{MB}$ proceeds as follows:

1. On methods contained in the user's application, for which source code is available, DEJAVU$_{MB}$ performs its usual actions, previously described
2. On methods contained in `Dispenser′` DEJAVU$_{MB}$ performs the following actions:

Table 3: Branch Coverage for Component `Dispenser`.

TC#	Branches Covered
1	none
2	none
3	none
4	(49,50) (50,53) (55,56) (57,59) (59,61) (61,62)
5	(49,50) (50,53) (55,56) (57,59) (59,61) (61,64)
6	(49,50) (50,53) (55,56) (57,59) (59,61) (61,64)
7	none
8	(49,50) (50,53) (55,56) (57,59) (59,61) (61,62)
9	(49,50) (50,53) (55,56) (57,59) (59,61) (61,64)
10	(49,50) (50,53) (55,56) (57,59) (59,61) (61,62)
11	(49,50) (50,53) (55,56) (57,59) (59,61) (61,64)
12	(49,50) (50,53) (55,56) (57,59) (59,61) (61,64)
13	(49,50) (50,53) (55,56) (57,59) (59,61) (61,64)
14	(49,50) (50,53) (55,56) (57,59) (59,61) (61,62)
15	(49,50) (50,53) (55,56) (57,59) (59,60)
16	(49,50) (50,53) (55,56) (57,59) (59,60)
17	(49,50) (50,53) (55,56) (57,59) (59,60)
18	(49,50) (50,53) (55,56) (57,59) (59,60)
19	(49,50) (50,53) (55,56) (57,59) (59,60)
20	(49,50) (50,53) (55,56) (57,58)
21	(49,50) (50,53) (55,56) (57,58)
22	(49,50) (50,53) (55,56) (57,58)
23	(49,50) (50,53) (55,56) (57,58)
24	(49,50) (50,53) (55,56) (57,58)

(a) retrieve `Dispenser`$'$'s version number

(b) use this information to query `Dispenser`$'$ about the dangerous branches with respect to `Dispenser`

(c) select the test cases associated with dangerous branches, referencing the coverage table.

In the example, differences between `Dispenser` and `Dispenser`$'$ cause branch (61,64) to be the only dangerous branch reported. This branch is exercised by test cases 5, 6, 9, 11, 12, and 13, so these six test cases are selected for re-execution – many fewer test cases than would be required without component metadata.

4.3 Empirical Study of Code-Based Regression Test Selection

For component-metadata-based approaches to regression test selection to be useful, they must be able to reduce testing costs. RTS algorithms have shown potential for reducing testing costs in prior empirical studies [11, 37, 35, 19, 38]; however, these studies have not involved component-based software systems, and it is not appropriate to conclude that their results will generalize to such systems. Moreover, the tradeoffs that exist among different types of metadata should be investigated for such systems. Thus, an empirical study was performed examining the results of applying several different code-based RTS techniques to a non-trivial component-based system. The remainder of this section describes the study design and results.

4.3.1 Independent Variable

The independent variable in this study is the particular code-based RTS technique utilized. In the following, let A be an application that uses a set of externally-developed components C, and that has been tested with a test suite T. Let C' be a new version of set of components C. Based on the discussion in Section 4.2, four safe RTS techniques are considered:

- **No component metadata.** The developer of A knows only that one or more of the components in C have been modified, but not which ones. 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. This is referred to as the NO-META technique.

- **Component-level RTS.** The developer of A possesses component metadata provided by the developer of C, supporting selection of test cases that exercise *components* changed in producing C' from C. This is referred to as the META-C technique.

- **Method-level RTS.** The developer of A possesses component metadata provided by the developer of C, supporting selection of test cases that exercise *methods* changed in producing C' from C. This is referred to as the META-M technique.

- **Statement-level RTS.** The developer of A possesses component metadata provided by the developer of C, supporting selection of test cases that exercise *statements* changed in producing C' from C. This is referred to as the META-S technique.

The NO-META technique does not use component metadata and serves as a control technique. The other three techniques do use metadata, and rely on different levels of information about changes between versions of components: component-level, method-level, or statement-level. These three techniques are referred to collectively as META techniques. By observing the application of these techniques it is possible to investigate the effectiveness of component-metadata-based techniques generally, along with the further question of whether the level of information about changes between versions of components can affect the performance of such techniques.

4.3.2 Dependent Variable and Measures

The dependent variable involves technique effectiveness in terms of savings in testing effort. Two measures are utilized for this variable: reduction in test suite size and reduction in test-execution time.

RTS techniques provide savings by reducing the effort required to regression test a modified program. Thus, one method used to compare such techniques [18] considers the degree to which the techniques reduce test-suite size for given modified versions of a program. Using this method, for each RTS technique R considered, and for each (version, subsequent-version) pair (P_i,P_{i+1}) of program P, where P_i is tested by test suite T, the percentage of T selected by R to test P_{i+1} is measured.

The fact that an RTS technique reduces the number of test cases that must be run does not guarantee that the technique will be cost-effective. That is, even if the number of test cases that need to be rerun is reduced, if this does not produce savings in testing time, the reduction in number of test cases will not produce savings. Moreover, savings in testing time might not be proportional to savings in number of test cases — for example, consider the case in which the test cases excluded are all inexpensive, while those not excluded are expensive. (See the work of Leung and White [40] for an applicable cost model.) Thus, to further evaluate savings, for each RTS technique, the time required to execute the selected subset of T on P_{i+1} was measured.

4.3.3 Study Subject

As a subject for the study, several versions of the Java implementation of the SIENA server [41] were utilized. 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 accepting notifications from publishers and for selecting notifications that are of interest to subscribers and delivering those notifications to the clients via access points.

To investigate the effects of using component metadata, it was necessary to obtain an application program 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.

The source code for eight different sequentially released versions of Siena (versions 1.8 through 1.15) was obtained. Each version provides enhanced functionality or corrections with respect to the preceding 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 sequence 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 the study.

To investigate the impact of component metadata on regression test selection it was also necessary to obtain a comprehensive test suite for the 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 considered, so one was created. To do this in an unbiased manner, the fifth author, who had been involved in requirements definition and design of Siena, independently created a black-box test specification using the category-partition method and TSL test specification language [42]. Individual test cases were created for these testing requirements. The resulting suite contains 567 test cases and served as the subject regression test suite for the study.

4.3.4 Procedure

Because the implementation of component metadata and support tools for directly applying the techniques would be expensive, a way was needed to study the use of metadata, initially, without creating such infrastructure. This approach makes sense from the standpoint of research methodologies because, if there is no evidence that component metadata can be useful for regression test selection, there may be no reason to create infrastructure to support direct experimentation with the use of component metadata. Furthermore, if a study conducted without formal infrastructure suggests that component metadata are useful, its results can help direct the subsequent implementation effort.

A procedure was thus designed by which it could be determined precisely, for a given test suite and (program, modified-version) pair, which test cases would be selected by the four target techniques. For each (program, modified-version) pair (P_i, P_{i+1}), the Unix `diff` utility and inspection of the code were used to locate differences between P_i and P_{i+1}, including modified, new, and deleted code. In cases where variable or type declarations differed, the components, methods, or statements in which those variables or types were used were determined, and those components were treated as if they had been modified. This information was used to determine (for the META-C, META-M, and META-S techniques, respectively) the components, methods, or statements in P_i that would be reported changed for that technique.

For each of the META techniques, the changed code (component, method, or statement) was instrumented for each version of the subject program so that, when reached, the instrumentation code outputs the text "selected," and then executables of the application were constructed from this instrumented code (one for each META technique). Given this procedure, to determine which test cases in T would be selected by the META techniques for (P_i, P_{i+1}) it was sufficient to execute all test cases in T on the 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 that META technique.

Determining the test cases that would be selected by the NO-META technique required a similar, but simpler approach. The application developer's portion of the code for P was instrumented, inserting code that outputs "selected" prior to any invocation of any method in C, and then the test cases in T were executed on that instrumented version.

The foregoing process requires that all test cases in T be executed to determine which would be selected by an actual RTS tool, and thus is useful only for experimentation. However, the approach supports the determination of exactly the test cases that would be selected by the techniques, without providing full implementations.

This approach was applied to each of the seven (program, modified-version) pairs of the Siena system with the given test suite, and used to record, for each of the four RTS techniques, the percentage of the test suite selected by that technique for that (program, modified-version) pair, and the time required to run the selected test cases. These percentages and times served as the data sets for the analysis.

4.3.5 Threats to Validity

Like any empirical study, this study has limitations that must be considered when interpreting its results. The application of four component-metadata-based RTS techniques to a single program and test suite and seven subsequent modified versions of the components that make up that program have been considered, but it cannot be claimed that these results generalize to other programs and versions. On the other hand, the program and versions used are part of an actual implementation of a non-trivial software system, and the test suite represents a test suite that could realistically be used in practice. Nevertheless, additional studies with other subjects are needed to address such questions of external validity.

Other limitations involve internal and construct validity. The tests used for Siena were created by one of the authors; however, this activity was performed prior to any exposure of that author to the particular algorithms, and the authors' familiarity with the system was necessary for external validity. Only two measures of regression test selection effectiveness have been considered: percentage reduction in test suite size and percentage reduction in test-execution time. Other costs, such as the cost of providing component metadata and performing test selection, may be important in practice. Also, the execution times reported do not factor in the cost of the analysis required to perform test selection, which would add costs to test selection; however, in other studies of test selection, those costs have been shown to be quite low [38]. Finally, the execution times include only the times required to execute, and not to validate, the test cases. Measuring validation costs would further increase test execution time, and increase savings associated with reductions in test-suite size.

4.3.6 Results and Analysis

Figure 7 depicts the test selection results measured. In the graph, each modified version of Siena's components occupies a position along the horizontal axis, and the test selection data for that version are represented by a vertical bar, black for the NO-META technique, dark grey for the META-C technique, light grey for the META-M technique, and white for the META-S technique. The height of the bar depicts the percentage of test cases selected by the technique on that version.

As the figure shows, the NO-META technique always selected 99.2% of the test cases. Only 0.8% of the test cases for Siena do not exercise components in C (the set of external components), and thus all others must be re-executed. Also, because the NO-META technique selects all test cases that execute any components in C, and the test cases in the test suite that encounter C did not vary across versions, the NO-META technique selected the same test cases for each version.

As the figure also shows, the three META techniques always selected a smaller subset of the test suite than the NO-META technique. For the META-C technique, the selected subset did not differ greatly from the subset selected by the NO-META technique. The META-C technique always selected 98.9% of the test cases, a difference of only 0.3% in comparison with the NO-META technique.

The META-M and META-S techniques provided greater savings. In the case of version C7, the differences were extreme: the META-M technique selected only 1.4% of the test cases in the test suite and the META-S technique selected none of the test cases, whereas the NO-META technique selected 99.2% of the test cases. (The fact that META-S selected no test cases on version C7 is not a drawback of that technique: it simply shows that existing test cases were inadequate for testing the code that was modified. Running existing test cases cannot help in testing this code, and re-using such test cases in this case is wasted effort. This case does suggest, however, that the META-S technique can help testers identify areas of the modified system requiring additional testing.) This large difference arose because the changes within C7 involved only a few methods and statements, where these methods were encountered by only a few test cases, and these statements were encountered by none of the test cases. On other versions, for the META-M technique, differences in selection were more modest, ranging from 0.7% to 32.6% of the test suite. The overall savings for the META-S technique, however, were greater, ranging from 11.8% to 84.6%.

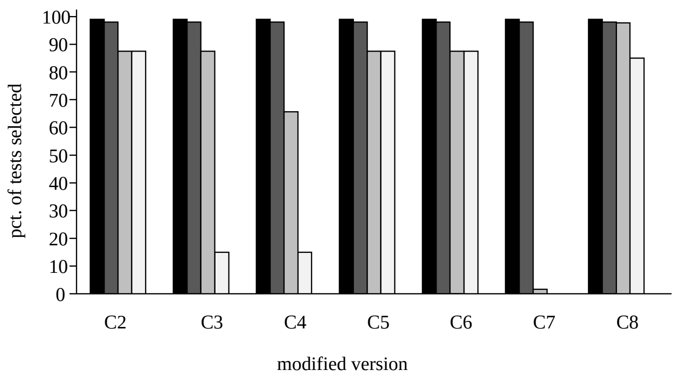

Figure 7: Test selection results for the NO-META (black), META-C (dark grey), META-M (light grey), and META-S (white) techniques.

Table 4: Execution Times (Hours:Minutes:Seconds) for Test Cases Selected by Techniques.

Version	NO-META	META-C	META-M	META-S
C2	2:20:23	2:20:05	2:04:44	2:04:01
C3	2:20:37	2:19:49	2:01:19	0:20:43
C4	2:20:19	2:19:50	1:34:49	0:20:30
C5	2:20:37	2:19:58	2:02:13	2:02:02
C6	2:20:30	2:19:51	2:02:13	2:02:03
C7	2:20:16	2:20:06	0:03:15	0:00:00
C8	2:20:19	2:19:47	2:19:33	1:57:26
average	2:20:26	2:19:55	1:43:59	1:15:15
cumulative	16:23:01	16:19:26	12:08:06	8:46:45

Note that on versions C3, C5, and C6, inspection of the data shows that the META-M 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. Similarly, inspection shows that the META-S technique selected identical test cases on versions C5 and C6. This occurred because the code changes involved the same sets of statements.

Next, test-execution times are considered. Table 4 shows, for each version of Siena's components considered, the hours, minutes and seconds required to test that version. The columns show the version number, and the time required to run the test cases selected by the NO-META, META-C, META-M, and META-S techniques, respectively. The last two rows of the table show average and cumulative times.

On average over the seven modified versions, the META-C technique produced little reduction in testing time: from 2 hours, 20 minutes and 26 seconds to 2 hours, 19 minutes and 55 seconds. The META-M technique did better, reducing average regression testing time to 1 hour, 43 minutes and 59 seconds, and META-S reduced average testing time to 1 hour, 15 minutes and 15 seconds.

Because regression testing is repeated over sequences of releases, savings over sequences are also important, so these are also considered. For the META-C technique, cumulative time savings were only 3 minutes and 35 seconds (0.37% of total time.) For the META-M and META-S techniques, however, cumulative savings were larger: 4 hours, 14 minutes and 55 seconds (25.9% of total time) in the former case, and 7 hours, 36 minutes and 16 seconds (46.4% of total time) in the latter case. Since the runtime for each test case was roughly the same across test cases, the percentage of cumulative time savings were about the same as the percentage of cumulative numbers of test cases saved, for all techniques; 0.4% savings of test cases for

Table 5: ANOVA for Test Time

Source	Sum of squares	d.f	Mean square	p-value
Technique	74803768	3	24934589	0.0035
Version	50215323	6	8369220	0.0924
Residuals	68811162	18	3822842	

Table 6: Comparisons Between NO-META and each META.

multiple comparison with a control by Dunnett's method					
Comparison	Estimate	Std. Error	Lower Bound	Upper Bound	
META-C : NO-META	-30.7	1050	-2710	2650	
META-M : NO-META	-2180.0	1050	-4860	494	
META-S : NO-META	-3910.0	1050	-6590	-1230	****

META-C, 25.9% for META-M, and 46.4% for META-S.

Considering results for META-M and META-S on individual versions, in their worst cases, the META-M technique saved only 46 seconds (0.55%) of testing time (on version C8), and the META-S technique saved 16 minutes and 22 seconds (11.7%) of testing time (on version C2). In their best cases, both on version C7, the META-M technique saved 2 hours, 17 minutes and 1 second (97.7%) of testing time and the META-S technique saved 2 hours, 20 minutes and 16 seconds (100%) of testing time. Here too, similar to the case with the cumulative results, savings in time are similar to savings in numbers of tests executed.

Of course, savings of a few hours or 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 days, or even weeks of effort, and much of this effort may be human-intensive. For the META-C technique, a savings of 0.37% of the overall testing effort for a sequence of seven releases would most likely be unimportant. For the META-M and META-S techniques, however, results suggest the possibility of achieving meaningful savings. If results such as those demonstrated by this study scale up, a savings of 25.9% of the overall testing effort for a sequence of seven releases using the META-M technique may be worthwhile, and a savings of 97.7% of the testing effort for a version may be substantial.

To determine whether the impact of RTS technique on test-execution time in the study was statistically significant, an ANOVA test [43] was performed on the test-execution time data in Table 4. Table 5 shows the results of this analysis. The results indicate that there is strong evidence that at least one of the techniques' test-execution times differ from one of the other techniques' test-execution times (p-value = 0.0035), for a significance level of 0.05.

Next, a multiple comparison with a control technique (the NO-META technique) was performed, to investigate whether there is a difference between each META technique and the NO-META technique, using Dunnett's method [44]. Table 6 presents the results of these comparisons. In the table, cases that were statistically significant are marked with "****" (which indicates confidence intervals that do not include zero), with a 95% confidence interval. The results indicate that the differences between the NO-META and META-C techniques, and between the NO-META and META-M techniques, are not statistically significant. However, the results show that there is a statistically significant difference between the META-S and the NO-META techniques. (The average test-execution time for the META-S technique is 3910.0 seconds less than the average test-execution time for the NO-META technique)

An all-pair comparison for all the META techniques was also performed, to investigate whether there was a difference between META techniques, using Tukey's method [44]. Table 7 presents the results of these comparisons. Also in this case, statistically significant cases, with a 95% confidence interval, are marked

Table 7: Comparisons Between Pairs of META Techniques.

all pair comparison by Tukey's method					
Comparison	Estimate	Std. Error	Lower Bound	Upper Bound	
META-M : META-S	1730.0	1110	-1250	4700	
META-M : META-C	-2150.0	1110	-5130	819	
META-S : META-C	-3880.0	1110	-6850	-907	****

with "****". The results show that there is a statistically significant difference between the META-S and META-C techniques.

Therefore, these results provide evidence that testing with component metadata could provide savings in testing costs, and that component metadata could be useful for regression test selection in component-based software. Furthermore, the differences among the META techniques in savings in test-execution time indicate that the level of information about changes can affect the degrees of savings in test-execution time, and suggest that a proper focus of implementation efforts would be development of the META-S technique.

5 Specification-Based RTS Using Component Metadata

This section describes and investigates an approach to using metadata to support regression test selection for component-based software, in the case in which specification-based testing is involved.

5.1 Specification-based Regression Test Selection

Specification-based RTS techniques [10, 16] select test cases based on some form of functional specification of a system, such as natural language specifications, FSM diagrams, or UML diagrams [21], and are complementary to code-based techniques.

An approach to regression test selection based on UML statecharts is considered. The technique applies to software that integrates an application A with a set of software components C by (1) combining specifications in the form of statecharts for A and C to build a global behavioral model, (2) identifying differences in the global behavioral model when a new version of C is integrated, and (3) selecting the test cases that exercise changed sections of the model. Note, however, that a naive algorithm for step 1 could result in a global statechart of size exponential in the numbers of states and transitions in the individual statecharts, so a heuristic for cost-effectively combining statecharts is needed. (One could also directly compare statecharts for an original and new version of a component. However, the focus here is on the interactions between applications and components, and the combined behavioral model facilitates addressing this issue.)

To illustrate, consider again the case in which a component user is integrating a set of components C with their application A. In step 1 of the approach, the user constructs a global behavioral model GB for their application (including components) by composing the statecharts for components incrementally, using an heuristic reduction algorithm and composition rules that reduce the size of a composed statechart by eliminating some unreachable states.

To perform this step, a version of the technique for composing general communicating finite state machines presented by Sabnani, Lapone, and Uyar [23] and specialized for integration testing by Hartmann, Imoberdorf, and Meisinger [22] was utilized. These approaches, which are targeted primarily at communicating sequential processes, are generalized here to extend the applicability of the technique to a wider set of systems. To this end, "simple" parameterized events and actions, and the use of a constrained form of guards, are allowed for. Further, scalar parameters and guards containing conditions either related to a parameter and composed of a single clause, or unrelated to parameters, are allowed for. Note that this is just one possible technique for composing specifications at the component level to derive a specification for the overall system. There exist other, related techniques that use different notations with similar goals (e.g., [45, 46]).

Given a global behavioral model GB, the component user can generate a set of *testing requirements* for the application using any testing technique based on state-machine coverage, such as Binder's adaptation [47] of Chow's method [48]. These testing requirements, which are typically paths through the state-machine, are then used by the tester to generate test cases, resulting in test suite T.

When the statechart specifications for one or more components in C are changed, and new versions C' of these components incorporating these changes are released, the component user must retest their application with these changed components. To perform this task using the model composition approach, the component user does the following:

1. generate a new global behavioral model GB' by composing statecharts for A with components in C';
2. compare GB and GB' by performing a pairwise walk of the two models, marking *dangerous transitions*, that is, edges that have been added, deleted, or modified in the new model, or edges leading to states that differ;
3. select all test cases in T that traverse at least one dangerous transition.

The algorithm for performing the pairwise walk in step 2 of this process is similar to that used by DEJAVU on control-flow graphs for code-based selection, and follows a similar motivation – the notion that test cases that reach changes should be selected, because these changes indicate places in which program behavior might change and erroneous behavior might be revealed by tests. (In this case, however, the graphs model required behavior, and the changes involve changes in requirements.) The algorithm begins at initial states, comparing states reached along identically-labeled outgoing edges, until differences are found. Then the dangerous transitions that are defined in step 2 are identified; detailed descriptions of the process of identifiying dangerous transitions can be found in [14]. When new edges out of a node are found, two approaches are possible: they can be ignored for purposes of test selection (under the assumptions that previous testing requirements cannot include them, and that new test cases will be created to exercise them), or all transitions into their source node can be identified as dangerous.

Because testing requirements are generated based on the statechart specification of the system, and defined as sequences of states and transitions, once the dangerous transitions have been identified, selecting the test cases associated with dangerous transitions requires a simple set union over entries in the test coverage table.

The presentation of an example of this technique is deferred to the next section, where the technique is illustrated using a metadata-based adaptation of the approach.

5.2 A Component-Metadata-Based Approach

The technique just described requires UML statecharts for components, and is applicable only in cases in which the component developer can provide them (e.g., when sharing components in-house). The component developer can provide these statecharts as component metadata, by encoding them in appropriate data structures accessed through metamethods. One simple approach utilizes statecharts output by Rational Rose [49]; these can be parsed by a statechart composition tool, and the results rendered in the same format, for use in generating testing requirements and test cases and selecting test cases.

Analogous to the code-based approach, if a new version of C, C', is released, and no information about the changes between C and C' is available, the component user may need to execute all of the test cases for A that exercise components in C. If the specification for the components in C' is available, however, the component user can exploit it to perform regression test selection, as is now illustrated using the vending machine example. (Note that the process can also be applied when components in A are modified, but here the focus is on the case in which C alone is changed for simplicity.)

Consider the application `VendingMachine` and its statechart (Figure 3). When the component user first acquires `Dispenser`, they retrieve its statechart using a metamethod. Next, they use an implementation of the composition algorithm discussed above to compose that statechart with the statechart for `VendingMachine`, obtaining the global statechart `VendingMachine-Dispenser` shown in Figure 8.[2]

As described above, using the global statechart, the component user can create a set of test cases by applying various state-machine-based testing approaches (e.g., [50, 48, 22]). Table 8 shows one possible set of testing requirements for `VendingMachine`, designed to cover each path $p = (s_0, s_1, \ldots, s_n)$ in the global statechart such that (1) $n = 0$ and $s_0 = s_n =$ `NoCoins_Empty`, (2) $n \neq 0$ and $s_i \neq$ `NoCoins_Empty` for $i = 1, \ldots, n-1$, and (3) p does not traverse the same edge twice. (The user would create test cases for each of these requirements, using appropriate combinations of test drivers and inputs.)

Suppose the component developer releases a new version of `Dispenser`, `Dispenser′`, in which the specification is changed by inserting a new intermediate state, `Preparing`, between states `Enabled` and `Empty` (see Figure 9). When `Dispenser′` is acquired, the user uses a metamethod to retrieve its new specification, and build a new global behavioral model (Figure 10).

Finally, the user applies the statechart-based version of Dejavu. When this algorithm processes the models in Figures 9 and 10, it identifies the transition from `ReadyToDispenseEnabled` to `DispensingEnabledAvail` as "dangerous", because it leads to states that differ in the two models. This transition is exercised by test cases 8 and 10, so these two testing requirements (or ultimately, their associated test cases) are selected.

5.3 Empirical Study of Specification-Based Regression Test Selection

To investigate whether the use of component metadata can benefit specification-based regression test selection for applications built with external components, an empirical study similar to the first study was performed, but focusing on the statechart-based technique.

5.3.1 Techniques

The independent variable is again RTS technique; two specific techniques are considered:

- **UML-state-diagram-based component metadata.** Components in C are provided with UML statecharts, in the form of metadata, sufficient for the developer of A to (1) build a global behavioral model of their application, (2) identify dangerous transitions, and (3) select test cases through dangerous transitions, following the procedure described in Sections 5.1 and 5.2. This is referred to as the META-U technique.
- **No component metadata.** The developer of A knows only that the statecharts for 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. This technique is the same control technique used in the first study, and is referred to (as previously) as the NO-META technique.

5.3.2 Measures

The dependent variable is a single measure of efficiency. Analogous to the measures used to investigate code-based techniques, RTS techniques' abilities to reduce retesting effort by reducing the number of requirements needing retesting is considered. Specifically, for each (version, subsequent-version) pair (GSD_i,GSD_{i+1}) of global statechart diagrams for program P, where a set of testing requirements TR_i can be generated from GSD_i, the percentage of requirements in TR_i selected by the technique as necessary to test P_{i+1}, given its statechart diagram GSD_{i+1}, is measured.

[2]Because a global statechart is constructed after normalizing the individual statecharts to contain only one event or action on each transition, some of the states in the global statechart are composed of states that are not present in the original statecharts. For example, the two composing states of state `Dispensing_InsufficientDispensing` are `Dispensing` and `InsufficientDispensing`; the latter is a state of the normalized statechart for `Dispenser`. The normalized statecharts for `VendingMachine` and `Dispenser` are provided in the Appendix.

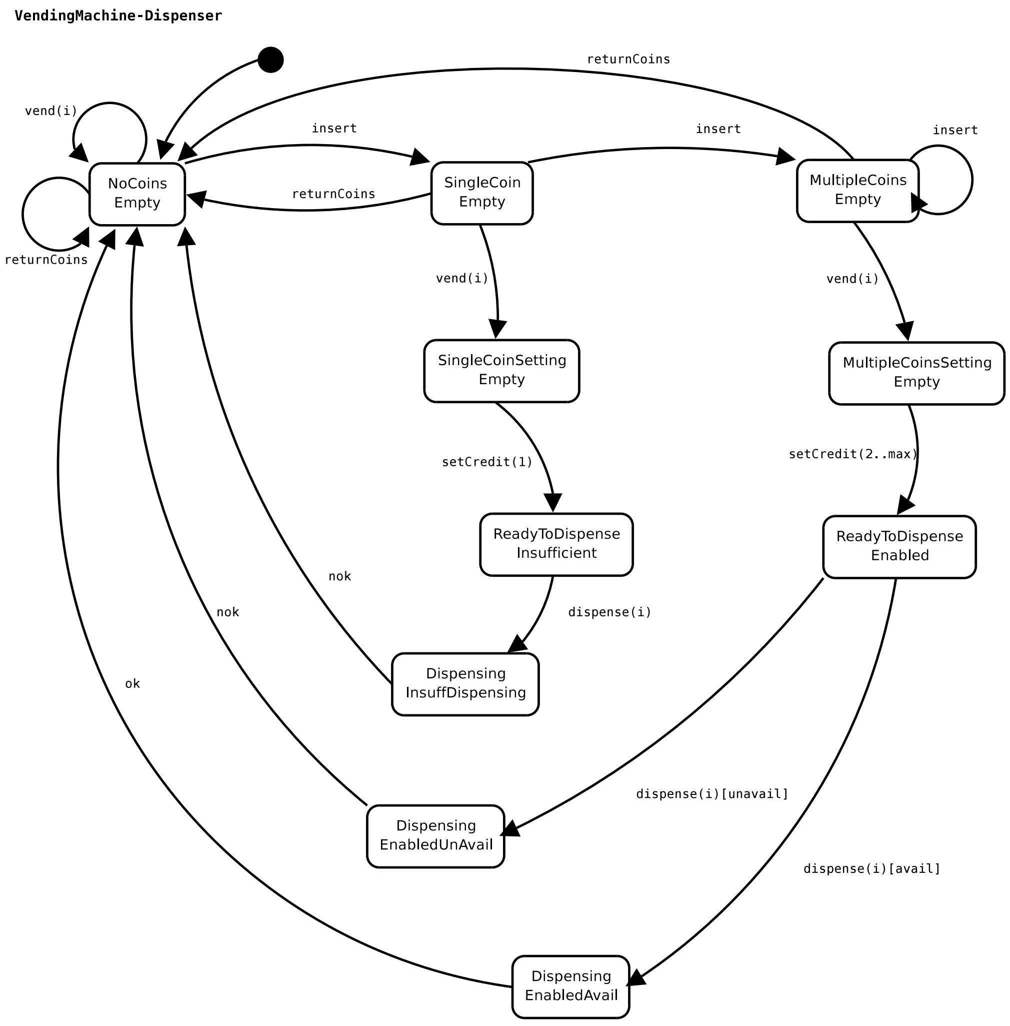

Figure 8: Global statechart for `VendingMachine` and `Dispenser`.

Table 8: Testing Requirements for `VendingMachine-Dispenser`.

TR#	Testing Requirement
1	vend(i)
2	returnCoins
3	insert, returnCoins
4	insert, insert, returnCoins
5	insert, insert, insert, returnCoins
6	insert, vend(i), setCredit(1), dispense, nok
7	insert, insert, vend(i), setCredit(2..max), dispense[unavail], nok
8	insert, insert, vend(i), setCredit(2..max), dispense[avail], ok
9	insert, insert, insert, vend(i), setCredit(2..max), dispense[unavail], nok
10	insert, insert, insert, vend(i), setCredit(2..max), dispense[avail], ok

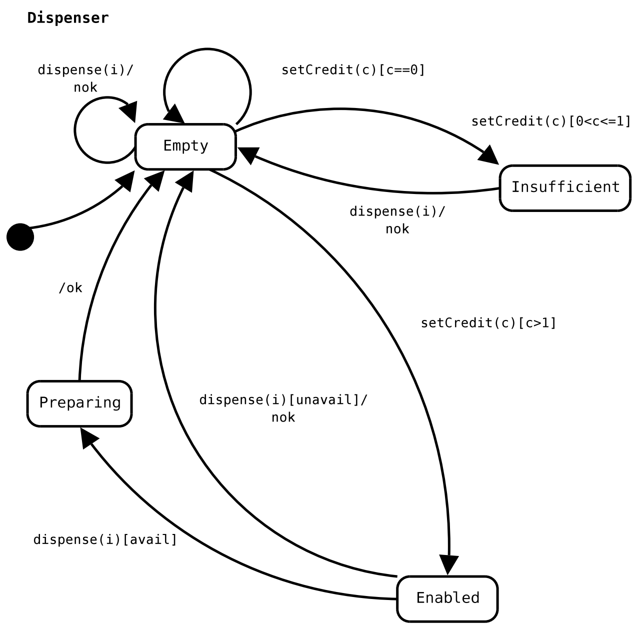

Figure 9: Statechart specification of `Dispenser′`.

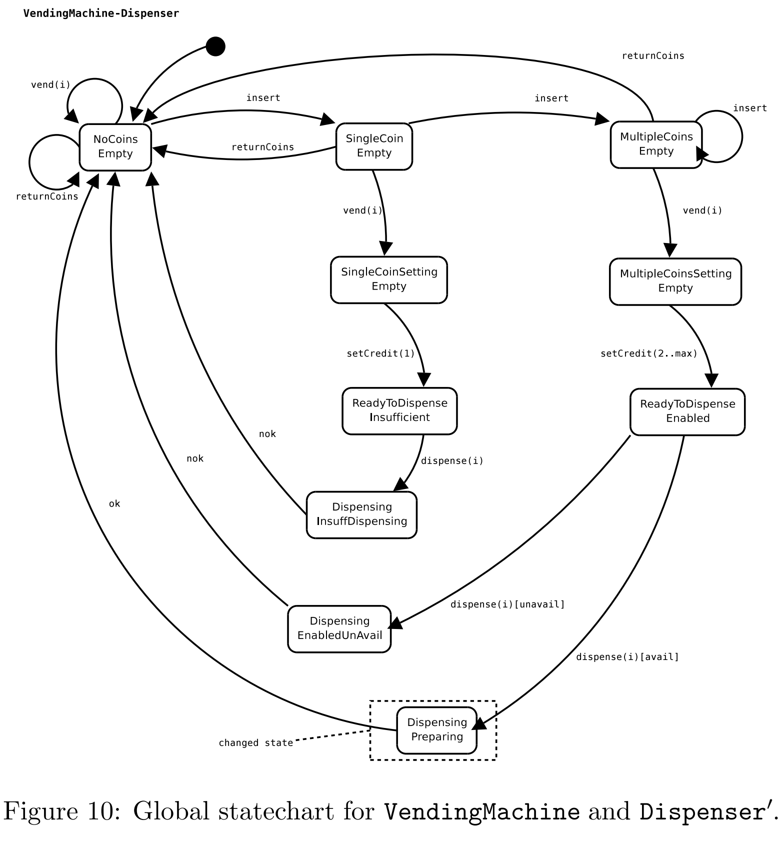

Figure 10: Global statechart for `VendingMachine` and `Dispenser′`.

Table 9: The Number of States and Transitions for Individual and Global Statechart Diagrams Per Version. Each Entry Takes the Form: Number_of_states (Number_of_transitions).

	V_1	V_2	V_3	V_4	V_5	V_6
App	8(13)	8(13)	8(13)	8(13)	8(13)	8(13)
Comp1	14(22)	16(26)	16(26)	16(25)	16(25)	16(25)
Comp2	5(12)	5(14)	5(14)	5(14)	5(14)	5(14)
Comp3	11(18)	11(19)	11(19)	11(19)	11(19)	11(19)
Comp4	3(10)	3(10)	3(10)	3(10)	3(11)	3(11)
Global	87(262)	89(285)	89(285)	84(277)	84(277)	84(277)

5.3.3 Study Subject

As mentioned in Section 1, different techniques are expected to be of different appropriateness for application to different programs. The statechart-based technique presented here is not appropriate for Siena — Siena involves only a single connection between application and component, which is not adequate to illustrate general software behavior where more than one interface between application and components exist. Thus, as a subject for this study, six sequentially released versions of the Java implementation of an XML parser, NanoXML [51], a component library consisting of 17 classes and 226 methods, were selected. An application program, JXML2SQL, was also obtained, which uses this library to read XML documents and respond to user queries about those documents, generating either an HTML file (showing its contents in tabular form) or an SQL file.

Because neither a textual specification nor a statechart diagram for NanoXML or JXML2SQL was available, UML statechart diagrams for these systems were constructed by reverse engineering from their code; this step was performed by several graduate students experienced in UML, but unacquainted with the plans for this study or the technique being investigated. The classes in the NanoXML library were grouped into four logical groups: Parser, Validator, Builder, and XMLElement handler. One statechart diagram was obtained for the application, and four statechart diagrams (one per logical group) were obtained for each of the six different sequentially released versions of the NanoXML library. The Rational Rose Case tool, which complies with UML notation, was used to build these statechart diagrams.

5.3.4 Procedure

For each version of statechart diagrams $SD_{i1}, SD_{i2}, \ldots, SD_{i5}$, $1 \leq i \leq 6$, a global statechart diagram GSD_i was constructed by composing the statechart diagrams incrementally using a composition tool implementing the technique described in Section 5.1. Table 9 shows the number of states and edges in each version of the individual component statechart diagrams and global statechart diagrams for the subject program.[3] Note that by inspection, it was determined that versions 3 and 6 contained no functional (statechart-level) changes with respect to preceding versions, whereas other versions did contain changes (this includes V_5 which, though possessing the same total number of states and transitions in the global statechart as V_4, did contain different transitions, due to differences in the statecharts for component 4).

To generate testing requirements from the global statechart diagrams, a tool to generate linearly independent test paths was constructed. A *linearly independent path* is a path that includes at least one edge that has not been traversed previously (in a given set of paths under construction) [52], and such sets of linearly independent paths are used by testers as requirements for test cases [50]. Applying this tool to the statecharts produced 54 testing requirements for the global statechart for version 1, and 66 testing requirements for each of the global statecharts for versions 2 through 5.

[3]By applying the heuristics described in Section 5.1, the number of states in global statechart diagrams can be reduced well beyond the number that might be present given a naive approach. For instance, for V_1, the total number of states in the global statechart diagram is 87 instead of 18480, which is the number that would have been obtained simply by naively composing all the statecharts.

Table 10: Testing Requirement Selection Rates

Version	NO-META	META-U
C_2	100.0%	83.33%
C_3	0.0%	0.0%
C_4	100.0%	86.36%
C_5	100.0%	39.39%
C_6	0.0%	0.0%

Given the global statechart diagrams and testing requirements for each version, for each pair (GSD_i, GSD_{i+1}) of sequential global statechart diagrams, an implementation of the DejaVu algorithm for statecharts to locate differences between GSD_i and GSD_{i+1} was used. This implementation outputs *dangerous transitions* – all testing requirements containing these transitions are selected.

These procedures were applied to each of the five (GSD_i, GSD_{i+1}) pairs of NanoXML and its application, and the percentage of testing requirements selected by the spec-based RTS technique for that (GSD_i, GSD_{i+1}) pair. These percentages served as the data sets for the subsequent analysis.

5.3.5 Results

Table 10 shows the testing requirement selection rates observed in this study. On versions 3 and 6, no testing requirements were selected by either technique because there were no changes made to the statecharts for these versions: the code changes for these versions did not involve functional changes. Because the main goal of statechart-based regression test selection is to identify test cases addressing changed system requirements, these results are appropriate.

On versions 2, 4, and 5, selection was able to reduce the number of requirements needing retesting, with respect to those identified by the NO-META technique. Selection rates for versions 2 and 4 were 83.33% and 86.36%, respectively, and the selection rate for version 5 was 39.39%. On versions 2 and 4, in fact, only a few (in each case, 4) dangerous edges were identified, but these edges were present in most testing requirements. On version 5, 12 dangerous edges were identified, but these were present in fewer requirements.

Like the study of code-based regression, this study has limitations. Only a single program has been considered, and statechart diagrams have been constructed through reverse engineering, and it cannot be claimed that these results will generalize to other systems and diagrams. Additional studies with other subjects that can provide statechart diagrams as component metadata are needed to address such questions of external validity. These results do provide, however, an initial look at the feasibility of statechart-based regression test selection using metadata, and suggest the potential utility of further work.

6 Conclusion and Future Work

In this paper, the results of an investigation of whether component metadata can be leveraged to support and improve the cost-effectiveness of software engineering tasks for component-based applications has been presented. In particular, the paper has focused on regression testing. Two new techniques for performing regression test selection based on component metadata have been introduced. Being code-based and specification-based, the two techniques are potentially complementary on systems to which both can be applied.

The application of these techniques has been illustrated on examples, and initial empirical results of applying them to two real Java systems have been provided. These results show that component metadata can feasibly be used to produce savings in retesting cost through regression test selection; such a demonstration is a prerequisite for arguing for the utility of further research on component metadata.

Having fulfilled this prerequisite, however, there are many open questions that must be addressed and that suggest a number of directions for future research.

A first set of questions continues this paper's focus on regression testing. Here, the problem of re-testing applications as the components they use evolve has been considered. The re-testing of applications as they use entirely new variants of particular components is another problem worth addressing. Discussion is also limited to selection of existing test cases, but identification of situations where new test cases are needed is also important. Finally, one mechanism for combining one particular form of specification has been considered, but many alternatives could also be investigated.

A second set of questions involves whether other uses of component metadata may provide cost-effective solutions to software-engineering problems involving components. Can component metadata aid maintainers in judging the impact of changes in components on their applications? Can component metadata help engineers verify security properties for their applications that use components? If so, can component metadata be guaranteed to reflect actual properties of the component in a manner similar to that achieved by proof-carrying code? Also, can component metadata be retrofitted with new information after the component has been deployed?

A third set of questions involve the mechanics of component metadata. What is the overhead in terms of component size and component execution time of incorporating metadata and metamethods into a component? How does this overhead vary as the amount and variety of metadata and metamethods are increased? How can component-metadata-based techniques function in the presence of deep nestings of components? For example, suppose application A depends on component $C1$, and component $C1$ depends on component $C2$, but $C1$ and $C2$ are developed by different (or even multiple) vendors, and it is the developer of A who decides which flavor of $C1$ and $C2$ to integrate. In this case, what mechanisms are needed to allow $C1$ to detect and extract metadata from C2 as part of its metadata generation for A? What happens if the developer of A decides to switch vendors from one version of a component to the next?

A fourth set of questions involve trust and privacy. What kind of software-engineering tasks can be addressed using component metadata without revealing too much information about the component? For the tasks for which component metadata must contain sensitive information (e.g., a dependence graph), can the information be encoded so that the component user cannot understand it, but the technique can still use it? In general, is there some way of mathematically showing that adding a certain kind of metadata does not reveal too much about a component?

The results of the research presented in this paper provide evidence that component metadata can have value, and thus, that these sets of questions are worth addressing. Through such consideration, it may be possible to find new ways to address some of the critical software-engineering problems raised by component-based software systems.

Acknowledgements

Antonio Carzaniga provided the source code for the Siena system and helped with its installation. Fred Ramsey of the Oregon State University Statistics Department provided assistance with our statistical analysis. Mary Lou Soffa contributed to initial work in this area. This work was supported in part by grants from Boeing Aerospace Corp., by the state of Georgia under the Yamacraw Mission, and by NSF awards EIA-0196145 and CCR-0096321 to Georgia Institute of Technology, by NSF award CCR-9703108 to Oregon State University 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. David S. Rosenblum holds a Wolfson Research Merit Award from the Royal Society.

References

[1] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metacontents to support the regression testing of component-based software. In *Proceedings of the*

International Conference on Software Maintenance, pages 716–725, November 2001.

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

[3] Clemens Szyperski. *Component Oriented Programming*. Addison-Wesley, 1st edition, 1997.

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

[5] P. Brereton and D. Budgen. Component-based systems: A classification of issues. *IEEE Computer*, 33(11):54–52, Nov. 2000.

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

[7] H. K. N. Leung and L. J. White. Insights into regression testing. In *Proceedings of the International Conference on Software Maintenance*, pages 60–69, October 1989.

[8] K. Onoma, W-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, May 1988.

[9] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, August 1997.

[10] L. C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on UML design. In *Proceedings of the International Conference on Software Maintenance*, pages 252–261, October 2002.

[11] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–220, May 1994.

[12] 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*, pages 312–326, October 2001.

[13] P. Hsia, X. Li, D. Kung, C-T. Hsu, L Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of OO software. *Software Maintenance: Research and Practice*, 9(4):217–233, 1997.

[14] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[15] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *ENCRESS '97, Third International Conference on Reliability, Quality, and Safety of Software Intensive Systems*, May 1997.

[16] A. von Mayrhauser and N. Zhang. Automated regression testing using DBT and Sleuth. *Journal of Software Maintenance*, 11(2):93–116, 1999.

[17] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

[18] John Bible, Gregg Rothermel, and David 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, April 2001.

[19] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *Journal of Software Testing, Verification, and Reliability*, 10(2):77–109, June 2000.

[20] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the International Conference on Software Maintenance*, pages 262–270, November 1992.

[21] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, Reading, Massachusetts, USA, first edition, 1999.

[22] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-based integration testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 60–70, August 2000.

[23] K. K. Sabnani, A. M. Lapone, and M. Ü. Uyar. An algorithmic procedure for checking safety properties of protocols. *IEEE Transactions on Communications*, 37(9):940–948, September 1989.

[24] Nat Brown and Charlie Kindel. Distributed component object model protocol: Dcom/1.0. `http://www.microsoft.com/com/default.mspx`, January 1998. Microsoft Corporation, Redmond, WA.

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

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

[27] JavaBeans Documentation. `http://java.sun.com/products/javabenas/index.jsp`, Oct. 2000.

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

[29] C. Liu and D. Richardson. Software components with retrospectors. In *International Workshop on the Role of Software Architecture in Testing and Analysis*, Marsala, Italy, July 1998.

[30] C. Liu and D. Richardson. Towards discovery, specification, and verification of component usage. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, pages 331–334, 1999.

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

[32] XOTcl - extended object Tcl. `http://media.wu-wien.ac.at/`, November 2000.

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

[34] K.F. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference B-6-3*, pages 1–6, November 1981.

[35] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering* , pages 241–251, November 2004.

[36] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance*, pages 44–53, November 1998.

[37] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the International Conference on Software Engineering*, pages 188–197, April 1998.

[38] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[39] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *ACM SIGPLAN Notices*, volume 36(5), pages 168–179, May 2001.

[40] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Proceedings of the International Conference on Software Maintenance*, pages 201–208, Oct. 1991.

[41] 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.

[42] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[43] Fred L. Ramsey and Daniel W. Schafer. *The Statistical Sleuth*. Duxbury Press, 1st edition, 1997.

[44] J.C. Hsu. *Multiple Comparisons: Theory and Methods*. Chapman & Hall, London, 1996.

[45] Michael Barnett, Colin Campbell, Wolfram Schulte, and Margus Veanes. Specification, simulation and testing of COM components using abstract state machines. In *Proceedings of the International Workshop on Abstract State Machines (ASM 2001)*, pages 266–270, February 2001.

[46] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Testing with abstract state machines. In *Proceedings of the International Workshop on Abstract State Machines (ASM 2001)*, pages 257–261, February 2001.

[47] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Reading, MA, 2000.

[48] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[49] IBM Rational Software, 2003. `http://IBM.com/rational`.

[50] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1999.

[51] Marc De Scheemaecker. NanoXML: A small XML parser for Java. `http://nanoxml.sourceforge.net`, 2006.

[52] R. S. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, 5th edition, 2001.

A Normalized Statecharts for `VendingMachine` and `Dispenser`

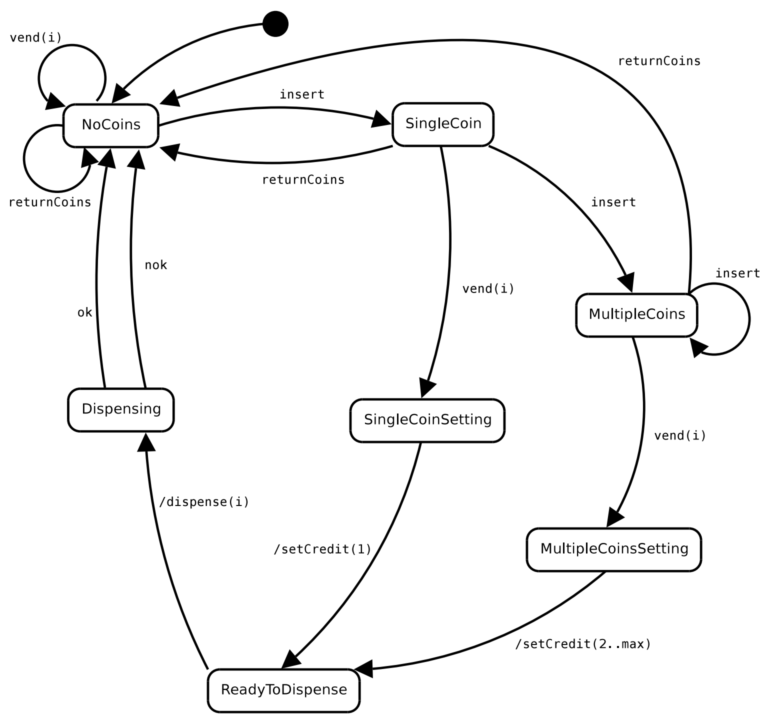

Figure 11: Normalized statechart for `VendingMachine`.

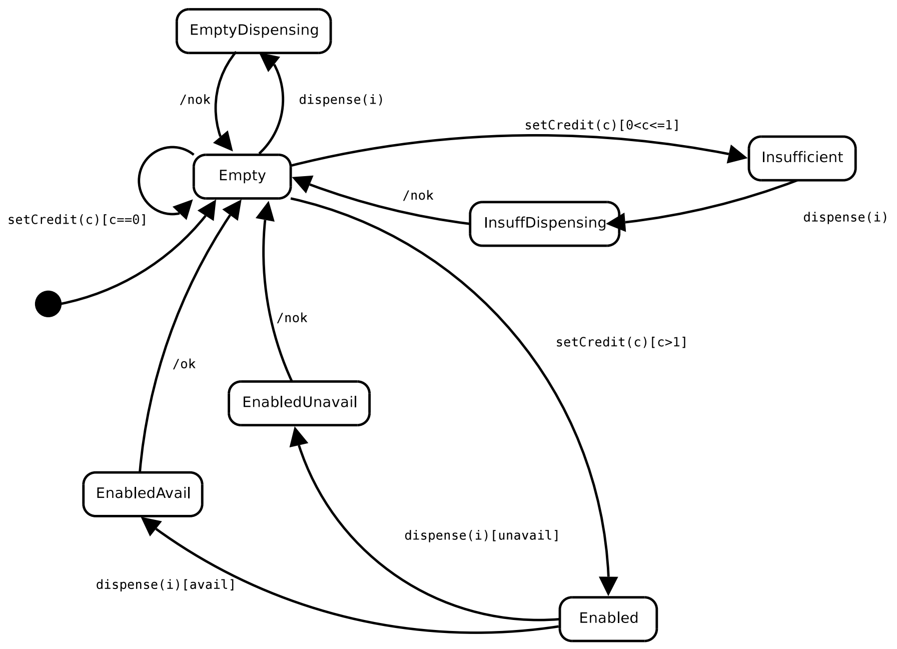

Figure 12: Normalized statechart for `Dispenser`.