# Predicting Problems Caused by Component Upgrades

Stephen McCamant and Michael D. Ernst

MIT Laboratory for Computer Science

Technical Report MIT-LCS-TR-941
March 2004

**Abstract**

This report presents a new, automatic technique to assess whether replacing a component of a software system by a purportedly compatible component may change the behavior of the system. The technique operates before integrating the new component into the system or running system tests, permitting quicker and cheaper identification of problems. It takes into account the system's use of the component, because a particular component upgrade may be desirable in one context but undesirable in another. No formal specifications are required, permitting detection of problems due either to errors in the component or to errors in the system. Both external and internal behaviors can be compared, enabling detection of problems that are not immediately reflected in the output.

The technique generates an operational abstraction for the old component in the context of the system, and one for the new component in the context of its test suite. An operational abstraction is a set of program properties that generalizes over observed run-time behavior. Modeling a system as divided into modules, and taking into account the control and data flow between the modules, we formulate a logical condition to guarantee that the system's behavior is preserved across a component replacement. If automated logical comparison indicates that the new component does not make all the guarantees that the old one did, then the upgrade may affect system behavior and should not be performed without further scrutiny.

We describe a practical implementation of the technique, incorporating enhancements to handle non-local state, non-determinism, and missing test suites, and to distinguish old from new incompatibilities. We evaluate the implementation in case studies using real-world systems, including the Linux C library and 48 Unix programs. Our implementation identified real incompatibilities among versions of the C library that affected some of the programs, and it approved the upgrades for other programs that were unaffected by the changes.

This report is a revision of the first author's Master's thesis, submitted January 2004.



Keywords: software upgrades, specification matching, software components, component frameworks, dynamic analysis, operational abstraction, dynamic library interposition, dynamic invariant detection, automated reasoning, behavioral substitutability

# Contents

# List of Figures

# Chapter 1

# Introduction

Software is too brittle. It fails too often, and it fails unexpectedly. The problem is often the use of software in unexpected or untested situations, in which it does not behave as intended or desired [Wei02, Dev99]. It is impossible to test software in every possible situation in which it might be used; in fact, it is usually impossible even to foresee every such situation.

We seek to mitigate problems resulting from unanticipated interactions among software components. In particular, our goal is to enhance the reliability of software upgrades by predicting upgrades that may cause system failure or misbehavior, as might occur when a supposedly compatible upgrade is used in a situation for which it was not designed or tested.

The key question that we seek to answer is, "Will upgrading a component, which has been tested by its author, cause a system that uses the component to fail?" In order to reduce costs by detecting problems early in the upgrade process, we wish to answer this question before integrating the new component into the system or fielding and testing the new system (though such testing is also advisable); therefore, the question must be answered based on the past behavior of the system and the component author's own tests. Because a particular upgrade may be innocuous or desirable to one user but disastrous to another, the upgrade decision must be based on the component's use in a particular system; the decision cannot be made without knowing the system-specific context. Finally, the upgrade process should warn about errors in the component (a new version violates its specification), errors in the application (it relies on behavior that is not part of the component's specification), and errors in which blame is impossible to assign (for example, because there is no formal specification).

The rest of the report introduces and evaluates an upgrade comparison technique that has these properties. Chapter 2 introduces the technique in its simplest form, in the context of a small but complete example. Chapters 3 and 4 describe the technique in a more general setting, and give examples of additional situations it is suitable for. Chapters 5 and 6 describe additional improvements to make the technique more practical, and discuss some details of our prototype implementation. Chapters 7 and 8 give our evaluation of our prototype in case studies using open-source libraries and applications written in Perl and C. Finally, Chapter 9 compares our approach to other research with similar goals and methods, and Chapter 10 concludes.

This report is a revision of the first author's Master's thesis, submitted January 2004. The research described herein was supported in part by the National Science Foundation (grants CCR-0133580 and CCR-0234651), by gifts from NTT and the Edison Design Group, by an MIT Presidential (Akamai) Graduate Fellowship, and by a National Defense Science and Engineering Graduate Fellowship.

# Chapter 2

# Basic technique

This chapter presents the outline of our approach, gives an example, and then describes the approach in detail. In the interest of concise exposition, some complicating aspects are postponed by considering only systems consisting of an application using a single component.

## 2.1 Comparing observed behavior

Our approach is to compare the observed behavior of an old component to the observed behavior of a new component, and permit the upgrade only if the behaviors are compatible, as demonstrated by testing covering the uses of the component by the application. Our method issues a warning when the behaviors of the new and old components are incompatible, but lack of such a warning is not a guarantee of correctness, nor is its presence a guarantee that the program's operation would be incorrect.

The two key techniques that underlie our methodology are formally capturing observed behaviors and comparing those behaviors via logical implication. We capture observed behavior via dynamic detection of likely program invariants [Ern00, ECGN01], which generalizes over program executions to produce an *operational abstraction*. An operational abstraction is a set of mathematical properties describing the observed behavior. An operational abstraction is syntactically identical to a formal specification — both describe program behavior via logical formulas over program variables — but an operational abstraction describes actual program behavior and can be generated automatically. In practice, formal specifications are rarely available, because they are tedious and difficult to write, and when available they may fail to capture all of the properties on which program correctness depends.

Suppose you wish to replace an old component by a new component, and the old component is used in a particular system. The "component" can be any separately-developed unit of software, such as a library or dynamically loaded object. We refer to the system that uses the component as the "application". In general, either the upgraded component or the rest of the system might consist of multiple modules, and a single module might group a number of related classes or functions. For simplicity of explanation, this chapter will consider a situation in which all of the modules to be upgraded can be thought of as a unit, which will be referred to as "the component". Chapter 3 will describe how our technique generalizes to more complex upgrade situations that must be considered at a finer granularity.

Our technique is as follows. Generate an operational abstraction for the old component, running in the context of the system. Also generate an operational abstraction for the new component, running in the context of the test suite used to validate it. Both of these steps may be performed in advance. Now, perform the upgrade only if the new component's abstraction is stronger than the old component's abstraction, considering which component behaviors were used and tested. In other words, perform the upgrade if the new component has been verified (via testing) to perform correctly (i.e., as the old component did) for at least as many situations as the old component was ever exposed to.

A key advantage of our technique is that it does not require omniscient foresight: programmers need not predict every possible use to which their software might be put. It is highly likely that some users will apply software components in situations or environments that programmers did not have in mind when

```
// Sort the argument into ascending order
static void
bubble_sort(int[] a) {
  for (int x = a.length - 1; x > 0; x--) {
    for (int y = 0; y < x; y++) {
      if (a[y] > a[y+1])
        swap(a, y, y+1);
    }
  }
}
```

Figure 2.1: A sorting method that uses `swap`

```
// Exchange the two array elements at i and j
static void
swap(int[] a, int i, int j) {
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

Figure 2.2: Version 1 of a method that swaps two array elements

designing, implementing, or testing the component. Using our upgrade technique, a user is warned when a new component has not been tested in an environment like the user's, or if the developer has inadvertently changed the behavior in the user's environment; either of these situations could be the case even if the developer conscientiously tests the component in many other situations.

The remainder of this chapter will illustrate our technique with a simple example, then describe the details of its operation.

## 2.2 Sorting example

This section gives a simple but complete example to illustrate our approach, based on a familiar small program.

### 2.2.1 The problem

Consider the sorting routine of Figure 2.1 as an application, and the `swap` subroutine of Figure 2.2 as a component it uses. Suppose `swap` is supplied by a third-party vendor and is specified to exchange the two array elements at indices $i$ and $j$. Together, these Java methods correctly sort integer arrays into ascending

```
// Exchange the two array elements at i and j
static void
swap(int[] a, int i, int j) {
  a[i] = a[i] ^ a[j]; // bitwise XOR
  a[j] = a[j] ^ a[i];
  a[i] = a[i] ^ a[j];
}
```

Figure 2.3: Version 2 of a method that swaps two array elements

```
// Sort the argument into ascending order
static void
selection_sort(int[] a) {
  for (int x = 0; x <= a.length - 2; x++) {
    int min = x;
    for (int y = x; y < a.length; y++) {
      if (a[y] < a[min])
        min = y;
    }
    swap(a, x, min);
  }
}
```

Figure 2.4: Another sorting method that uses `swap`

order. Now suppose that the vendor releases version 2 of `swap`, as shown in Figure 2.3. The component vendor asserts that the new version has been tested to meet the same specification as the previous version (perhaps using the same test suite as was used for the previous version). As the application's author, should you take advantage of this upgrade? In particular, will your `bubble_sort` application still work correctly with the new `swap` routine? Our technique automatically deduces the answer: yes, the upgrade is safe.

Now, consider a different author whose sorting application, as shown in Figure 2.4, also works correctly with the original version of the `swap` component. Should this author perform the same upgrade? In this case, the answer is no: the upgraded `swap` routine would cause this sorting application to malfunction severely, and should not be installed without a change either to it or to the application. Our technique automatically determines that this upgrade, for this application, is dangerous.

The next two sections describe how our technique reaches these conclusions.

### 2.2.2 Upgrading bubble sort

Before upgrading, the application developer constructs an operational abstraction describing how the old component works when called by the application, using an automated tool such as Daikon (Section 6.4, page 32). Figure 2.5 shows the abstraction generated for `swap` as used by the bubble sort. Figures 2.5, 2.6, and 2.7 show actual output from our system, though for brevity we have used a more compact notation and omitted a number of properties concerning subsequences of the array $a$.

When the vendor releases a new version of the component, the vendor also supplies an operational abstraction describing the new component's behavior over the vendor's test suite. Since `swap` is not specified to work for $a = \text{null}$ or for $i = j$, the vendor did not test the component for such values. The vendor's abstraction (Figure 2.6) captures the behavior of `swap` by guaranteeing that $a'[i] = a[j]$ and $a'[j] = a[i]$, subject to preconditions such as $i \neq j$.

Before installing the upgrade, the application developer uses an automated tool to compare the operational abstractions of the new component (as exercised by its test suite) and the old component (as exercised by the application). The application developer wants the new component to have the same postconditions as the old component, because other parts of the application may depend on those properties. Roughly speaking, the upgrade is safe to install if the component's abstraction logically implies the application's abstraction, showing that the component has been tested to perform as the application expects in the contexts where the application uses it. (Section 2.3.1 on page 10 explains this test in detail.)

The test has two parts: ensuring that the component precondition holds and ensuring that the application postcondition holds. In our example, the preconditions that the application establishes (Figure 2.5) imply those that the new component requires (Figure 2.6); for example, $j = i + 1 \Rightarrow i \neq j$. This means that the contexts in which the new component has been tested are a superset of those in which the application uses the component. On their own, the component postconditions (from its test suite) do not imply the application postconditions: for instance, the property $a'[i] < a'[j]$ is not true in the test suite. However,

| Preconditions for `swap` | Postconditions for `swap` |
|---|---|
| $a \neq \text{null}$ | $a'[i] = a[j]$ |
| $0 \leq i < \text{size}(a) - 1$ | $a'[j] = a[i]$ |
| $1 \leq j \leq \text{size}(a) - 1$ | $a'[i] = a'[j-1]$ |
| $i < j$ | $a'[j] = a[j-1]$ |
| $j = i + 1$ | $a'[i] < a'[j]$ |
| $a[i] > a[j]$ | |

Figure 2.5: The operational abstraction for `swap` (version 1), in the context of `bubble_sort`. Variable $a'$ represents the state of the array $a$ after the method is called.

| Preconditions for `swap` | Postconditions for `swap` |
|---|---|
| $a \neq \text{null}$ | $a'[i] = a[j]$ |
| $0 \leq i \leq \text{size}(a) - 1$ | $a'[j] = a[i]$ |
| $0 \leq j \leq \text{size}(a) - 1$ | |
| $i \neq j$ | |

Figure 2.6: The abstraction for `swap` (version 2), in the context of the vendor's test suite

this property is implied by the application precondition together with the component's tested behavior. In this example, the postcondition $a'[i] < a'[j]$ is implied by the precondition $a[i] > a[j]$, along with the fact that the elements at positions $i$ and $j$ are swapped, as captured by the test postconditions $a'[i] = a[j]$ and $a'[j] = a[i]$. Similar reasoning, easily performed by an automatic theorem prover, establishes all the other application postconditions.

The technique concludes that the upgrade is safe, up to the limits of the computed operational abstraction. In other words, bubble sort can use the new implementation of `swap`.

### 2.2.3 Upgrading selection sort

The author of the selection sort application can apply the same process. Figure 2.7 shows the operational abstraction for `swap` (version 1), in the context of `selection_sort`.

When the application developer compares this abstraction with that supplied by the vendor (Figure 2.6), the logical comparison fails. In particular, the new component's precondition $i \neq j$ is not established by the application. This suggests that the new component has not been tested in the way that `selection_sort` uses it, so the selection sort should not use the new `swap`.

In fact, testing would show that the new `swap`, when used with the selection sort, overwrites elements of the array with zeros whenever `swap` is called (contrary to its specification) to swap an element with itself. In this example, the reason for the mismatched abstraction, the special case $i = j$, also points out how the application could be fixed to work correctly with the new component: by checking that $x \neq min$ before calling `swap`.

| Preconditions for `swap` | Postconditions for `swap` |
|---|---|
| $a \neq \text{null}$ | $a'[i] = a[j]$ |
| $0 \leq i < \text{size}(a) - 1$ | $a'[j] = a[i]$ |
| $0 \leq j \leq \text{size}(a) - 1$ | |
| $i \leq j$ | |
| $a[i] \geq a[j]$ | |

Figure 2.7: The abstraction for `swap` (version 1), in the context of `selection_sort`

### 2.2.4 Who is at fault?

We have presented a scenario in which the author of selection sort is at fault for not obeying the specification of `swap`—though the selection sort happened to work fine with the first implementation of `swap`. Given the same code, one could also imagine that the specification of `swap` allowed an element to be swapped with itself, in which case the fault for the bug would lie with the vendor rather than the application developer. Or it may be that no careful specification exists at all (perhaps the documentation is ambiguous), so that the assignment of blame is unclear. Since the code is the same, our example would proceed identically in all these cases, and the dangerous upgrade would still be cautioned against, and prevented or fixed.

## 2.3 Detecting incompatibilities

This section gives a method for detecting incompatibilities between the behavior of the old version of a component and the behavior of the new version, by comparing abstractions (summaries) of the component's execution. (Section 6.4 on page 32 describes the technique for obtaining operational abstractions.) The method consists of four steps.

**(1)** Before an upgrade, when the application is running with the older version of a component, a tool automatically computes an operational abstraction from a representative subset (perhaps all) of its calls to the component. This may be done either online, to avoid recording and storing all the inputs and outputs, or offline, to use minimal CPU resources at system run time—whichever is more convenient. The result of this step is a formal mathematical description of those facets of the behavior of the old component that are used by the system. This abstraction depends both on the implementation of the component and on the way it is used by the application.

**(2)** Before distributing a new version of a component, the component vendor computes the operational abstraction of the new component's behavior as exercised by the vendor's test suite. This abstraction can be created as a routine part of the testing process. The result of this step is a mathematical description of the successfully tested aspects of the component's behavior.

**(3)** The vendor ships to the customer both the new component and its operational abstraction. The customer may either trust the accuracy of the abstraction, or may verify it in the following way. The customer uses the abstraction as an input to specification-based test suite generation [ROT89, DF93, CRS96, Meu98], computes an operational abstraction for the resulting test suite, and compares the two abstractions for inconsistencies.

**(4)** The customer's system automatically compares the two operational abstractions, to test whether the new component's abstraction is stronger than the old component's abstraction. (Our definition of "strength" appears in Section 2.3.1.) Success of the test suggests that the new component will work correctly wherever the system used the old component.

If the test does not succeed, the system might behave differently with the new component, and it should not be installed without further investigation. The tool reports the incompatibility in terms of specific procedure preconditions and postconditions. Further analysis could be performed (perhaps with human help) to decide whether to install the new component. In some cases, analysis will reveal that a serious error was avoided by not installing the new component. In other cases, the changed behavior might be acceptable:

- The change in component behavior might not affect the correct operation of the application.
- It might be possible to work around the change by modifying the application.
- The changed behavior might be a desirable bug fix or enhancement.
- The component might work correctly, but the vendor's testing might have insufficiently exercised the component, thus producing an operational abstraction that was too weak.

Of the four steps in our technique, the first three can be performed with existing tools. (We accomplished the first two using the Daikon dynamic invariant detector.) As part of this research, we implemented a tool to perform the final step, which is publicly available as part of the Daikon distribution. The details of the comparison that the tool performs are described in the following section; the details of its implementation appear in Section 6.1 on page 28.

### 2.3.1 Comparing abstractions

This section describes the test performed over operational abstractions to determine whether a new component may replace an old one, and explains why it is more appropriate than alternative conditions that are stronger or weaker.

Let $A$ be the operational abstraction describing the behavior of the old version of a component working in an application, and let $T$ be the operational abstraction describing the behavior of the new version in its test suite. $A$ is composed of preconditions $A_{\text{pre}}$ and postconditions $A_{\text{post}}$, and likewise for $T$. $T$ and $A$ are subsets of all the true statements about the tested behavior of the component and the application's use of it, limited to the grammar of the operational abstraction. $T$ approximates the specification of the new component, while $A$ approximates the old specification restricted to the functionality used by the application. Our correctness arguments are limited by this approximation. Our technique claims that the new component may be safely substituted for the old one in the application if and only if

$$A_{\text{pre}} \Rightarrow T_{\text{pre}} \qquad \text{and} \qquad (A_{\text{pre}} \wedge T_{\text{post}}) \Rightarrow A_{\text{post}} \quad . \tag{2.1}$$

#### Derivation of our condition

Our goal is to verify that the application will behave as it used to. This will be that case if, provided that the application's preconditions hold before each call to the component, its postconditions hold afterward; in other words, we want that $A_{\text{pre}} \Rightarrow A_{\text{post}}$.

We must conclude $A_{\text{pre}} \Rightarrow A_{\text{post}}$ based on the knowledge that the test abstraction accurately describes the behavior of the new component, i.e., that $T_{\text{pre}} \Rightarrow T_{\text{post}}$. Furthermore, we impose the side condition that the application's correct behavior must be achieved only by using the behavior of the component that was tested, since it is inherently unsafe to depend on code that has never been tested. The application's uses of the component are a subset of the tested uses exactly when $A_{\text{pre}} \Rightarrow T_{\text{pre}}$. Therefore, we are looking for a condition guaranteeing exactly that

$$((T_{\text{pre}} \Rightarrow T_{\text{post}}) \Rightarrow (A_{\text{pre}} \Rightarrow A_{\text{post}})) \ \wedge \ (A_{\text{pre}} \Rightarrow T_{\text{pre}}) \quad . \tag{2.2}$$

This formula is equivalent to (2.1).

#### Alternative conditions

Several conditions similar to ours have been suggested for other applications [ZW97]. We compare ours to those that are most similar, concluding that the alternatives are either too strong or too weak for our purpose.

A slightly simpler (and stronger) condition than ours is used as part of the definition of behavioral subtyping (see Section 9.1 on page 42):

$$A_{\text{pre}} \Rightarrow T_{\text{pre}} \qquad \text{and} \qquad T_{\text{post}} \Rightarrow A_{\text{post}} \quad . \tag{2.3}$$

As schematically illustrated in Figure 2.8, this condition precisely captures the operational intuition of replacing the old component with a new one (in Zaremski and Wing's terminology, "plug-in match") in guaranteeing the correct operation of the application. When the application executes, before the first use of the component, the application precondition $A_{\text{pre}}$ holds, so by the first implication, $T_{\text{pre}}$ holds. According to the component's tested behavior, $T_{\text{pre}} \Rightarrow T_{\text{post}}$, so $T_{\text{post}}$ holds. Then, by the second implication, $A_{\text{post}}$ holds, so the application's behavior remains the same. $A_{\text{pre}}$ then holds before the next call to the component, and so on for each subsequent call to the component, so that at each point the behavior of the application is the same.

Conditions (2.1) and (2.3) differ when $T_{\text{post}}$ holds, but neither $A_{\text{pre}}$ nor $A_{\text{post}}$ does. The possibility of such a situation would prohibit an upgrade according to (2.3), but not under our rule. An application may use only a subset of a component's tested behavior, so there might be possible executions of the component that are incompatible with the application's abstraction. Equivalently, we must take into account information about the particular way an application uses a component when checking that the behavior it needs is preserved, which corresponds to the addition of $A_{\text{pre}}$ to the second implication in our condition (2.1). For example,

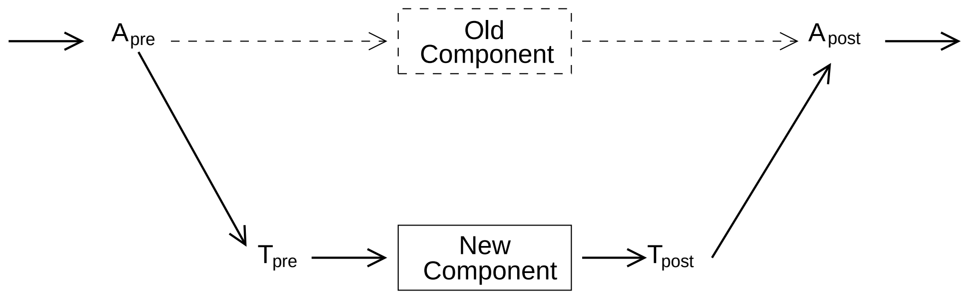


Figure 2.8: The behavioral subtyping rule. A new component whose specification guarantees $T_{\text{pre}} \Rightarrow T_{\text{post}}$ may replace an old component whose specification guaranteed $A_{\text{pre}} \Rightarrow A_{\text{post}}$ (dashed), if $A_{\text{pre}} \Rightarrow T_{\text{pre}}$ and $T_{\text{post}} \Rightarrow A_{\text{post}}$ (solid). Arrows represent both program control flow and logical implication between specifications. This rule is sufficient but too strong for validating component upgrades. Section 2.3.1 presents our alternate rule.

suppose that the component is an increment routine, and the old and new versions are behaviorally identical. Further suppose that the application happens to only increment even integers, whereas the component was tested with both even and odd inputs. Then $T_{\text{pre}}$ and $T_{\text{post}}$ might be $x$ *is an integer* and $x' = x + 1$, while $A_{\text{pre}}$ and $A_{\text{post}}$ would include $x$ *is even* and $x'$ *is odd* respectively. If $x = 5$ and $x' = 6$, the increment routine's tested abstraction would be satisfied, but both the pre- and post-conditions of the application's abstraction would be violated. However, our technique must allow an upgrade to this behaviorally-identical component. This limitation of the safety condition used in behavioral subtyping was noticed by Dhara and Leavens [DL96]; they make the same change as we do.

As seen in Section 2.3.1, our condition can also be understood as a strengthening of a weak upgrade condition,

$$(T_{\text{pre}} \Rightarrow T_{\text{post}}) \Rightarrow (A_{\text{pre}} \Rightarrow A_{\text{post}}) \quad . \tag{2.4}$$

This formula, "generalized match" in Zaremski and Wing's terminology, corresponds directly to the intuitive notion that the tested abstraction, which is an implication between pre- and postconditions, must be logically as strong as the application abstraction. However, this condition differs from (2.1) in its treatment of precondition violations. With this weaker rule, the fact that the component was never tested in some context where the application used it is not in itself a reason to reject an upgrade; it simply makes it more difficult to prove any needed aspects of the component's behavior in that context. For instance, suppose that an application used an old version of a component which took any non-zero integer as an argument, so that $A_{\text{pre}}$ is $x \neq 0$. Further suppose that the behavior of the component on negative values was well-defined but difficult to characterize as an operational abstraction, so that $A_{\text{post}}$ described only the behavior for positive arguments, say $x > 0 \Rightarrow x' = x$. Then, consider replacing this component with a new one tested only on positive inputs, but verified to work just as the old one did in that case (so that $T_{\text{pre}}$ and $T_{\text{post}}$ are $x > 0$ and $x' = x$ respectively). While the weaker condition would allow this upgrade, our condition (2.1) would prohibit it, because the new component was tested in fewer circumstances than the old one was used in. Our condition errs on the side of safety in prohibiting uses that are either not tested or are not captured in the operational abstraction. This safety is desirable in light of imperfect operational abstractions and non-functional properties like termination and exceptions.

Logically, our condition falls between the two alternatives above: it is strictly weaker than the first, and strictly stronger than the second. It is not among the conditions classified by Zaremski and Wing [ZW97]; their terminology might call it "application-guarded plug-in match." Chen and Cheng [CC00] refer to it as "relaxed plug-in match," (and to the equivalent condition (2.2) as "guarded generalized predicate match") and show that it is the weakest match condition that guarantees correctness in a relational model of program semantics.

## 2.4 Discussion

Our approach uses test suites as proxies for specifications; it could be called "behavioral subtesting" by analogy with behavioral subtyping. Our technique can be thought of as a refinement of the simple idea of directly comparing the test cases to the calls made by application. If the application's calls are a subset of those in the vendor's test suite (and the outputs are the same or are sufficiently similar), then the system with the new component will work everywhere that the system with the old component was ever used. Furthermore, the system with the new component is likely to work in new situations that are similar to the old situations. Comparing abstractions is more realistic than comparing all the underlying calls, though. Operational abstractions are usually more compact than the set of calls they abstract over, they leave out details that might be confidential or proprietary to the user or vendor, and they capture the common aspects of similar calls so that not every application call must exactly match a call in the test suite.

Our method is conservative in that it warns about all detected incompatibilities between versions. We expect that this is not a serious disadvantage in practice. In many cases the technique helps to avoid breaking a system by indicating an undesired component change. Even when the change is not catastrophic, the technique can warn about potential changes in application behavior that users might want to avoid. It can indicate why, in terms of differing properties of component behavior, an application behaves differently than before (even if the previous behavior was mistakenly believed to be correct). Alternately, the operational abstraction can help users understand that the change is an improvement and increase their confidence in it.

# Chapter 3

# A model of more general upgrades

The comparison technique described in Chapter 2 is appropriate for upgrades of a single component, containing a single method that is called from the rest of a system. It can easily be generalized to a component with several independent methods (by checking the safety of an upgrade to each method independently), or an upgrade to several cooperating components that are called by the rest of a system (by treating the components as a single entity for the purposes of an upgrade). More complicated situations, such as components that make callbacks or a simultaneous upgrade to two components that communicate via the rest of a system, require a more sophisticated approach. This section describes a model that generalizes the formulation and consistency condition for a single component as used by a single application. We consider systems to be divided into *modules* grouping together code that interacts closely and is developed as a unit. Such modules need not match the grouping imposed by language-level features such as classes or Java packages, but we assume that any upgrade affects one or more complete modules.

Our approach to upgrade safety verification takes advantage of this modular structure: we attempt to understand the behavior of each module on its own. Unlike many specification-based methods, however, the approach is not merely compositional, starting from the behavior of the smallest structures and combining information about them to predict or verify the behavior of the entire system. For our purpose of searching for differences in behavior, we examine each module of a running system to understand its workings in the context of the system, but conversely we also summarize the behavior of the rest of the system, as it was observed by that module. By combining these forms of information, we can predict problems that occur either when a module's behavior changes, or when the behavior that the system requires of a module goes beyond what the module has demonstrated via testing.

## 3.1 Relations inside and among modules

Given a decomposition of a system into modules, we model its behavior with three types of relations. *Call and return relations* represent how modules are connected by procedure calls and returns. *Internal data-flow relations* represent the behavior of individual modules, in context: that is, the way in which each output of the module potentially depends on the module's inputs. *External summary relations* represent a module's observations of the behavior of the rest of the system: how each input to the module might depend on the behavior of the rest of the system and any previous outputs of the module.

### 3.1.1 Call and return relations

Roughly speaking, each module is modeled as a black box, with certain inputs and outputs. We currently consider only modules connected by procedural interfaces, so these inputs and outputs correspond to cross-module procedure calls: when module A calls procedure `f` in module B, the arguments to `f` are outputs of A and inputs to B, while the return value and any side-effects on the arguments are outputs from B and inputs to A. In the module containing a procedure `f`, we use the symbol $f$ to refer to the input consisting of the values of the procedure's parameters on entrance, and $f'$ to refer to the output consisting of the return value and possibly-modified reference parameters. We use $f_c$ and $f_r$ for the call to and return from a procedure in the calling module. Collectively, we call these moments of execution *program points*. All non-trivial computation occurs within modules: calls and returns simply represent the transfer of information unchanged from one module to another. Each tuple of values at an $f_c$ is identical to some tuple at $f$, and likewise for $f_r$ and $f'$.

### 3.1.2 Internal data-flow relations

Internal data-flow relations connect each output of a module to all the inputs to that module that might affect the output value. (As a degenerate case, an *independent* output is one whose value is not affected by any input to the module. A constant-valued output would be independent, but an independent output might also be influenced by interactions not captured by our model: it might be the output of a pseudo-random number generator, or it might come from a file.) In a module $M$, $M(v|u_1, \ldots, u_k)$ is the data-flow relation connecting inputs $u_1$ through $u_k$ to an output $v$. A relation $M(v)$ represents an independent output $v$.

Conceptually, this relation is a set of tuples of values at the relevant inputs and at the output, having the property that on some execution of the output point, the output values might be those in the tuple, if the most recent values at all the inputs have their given values. However, each variable might have a large or infinite domain, so it would be impractical or impossible to represent this relation by a table. Instead, our approach summarizes it by a set of logical formulas that are (observed to be) always true over the input and output variables. The values that satisfy these formulas will be a (usually proper) superset of those that occurred in a particular run. This representation is not merely an implementation convenience. Generalization allows our technique to declare an upgrade compatible when its testing has been close enough to its use, without demanding that it be tested for every possible input.

The technique must be extended slightly to capture the fact that data-flow relationships may hold only after certain executions of the input points. A flow edge from an input $u$ to an output $v$ does not imply that every execution of $u$ is followed by some execution of $v$: for instance, $u$ might be the entry point of a procedure that calls another procedure at $v$ under some circumstances but not others. (For a concrete example, see Section 4.3 on page 22.) To keep track of when $u$ might be followed by $v$, our technique computes a property $\phi$ that held on executions of $u$ that were followed by executions of $v$, but did not hold on executions of $u$ that were followed by another execution of $u$ without an intervening $v$. Such a property is used to guard the statements describing a relationship between $u$ and $v$; in other words, we write those properties as implications with $\phi$ as the antecedent.

### 3.1.3 External summary relations

External summary relations are in many ways dual to internal data-flow relations. Summary relations connect each input of a module to all of the module outputs that might feed back to that input via the rest of the system. As a degenerate case, an *independent* input is not affected by any outputs. In a module $M$, we refer to the summary relation connecting outputs $u_1$ through $u_k$ to an input $v$ as $\overline{M}(v|u_1, \ldots, u_k)$, or just $\overline{M}(v)$ for an independent output. The line over the $M$ is meant to suggest that while this relation is calculated with respect to the interface of $M$, it is really a fact about the complement of $M$ — that is, all the other modules in the system.

### 3.1.4 Which relations exist

In general, only some of the possible data-flow and summary edges will appear in a model of a module. Which inputs are connected to each output via a data-flow relation is a choice that can be made to trade off the efficiency of our check against the possibility that it will discover unintended interactions. At one extreme, an author or integrator might conservatively declare that each output might be affected by any input; at the other, he or she might assume that each procedure is independent, so that the output of a given procedure depends only on its argument input, and the inputs representing the return values of other cross-module calls it makes. Because data flow relations describe only flows that occur entirely within a single module, it would also be reasonable for the module's developers to determine which relations should be examined as part of the module's development, perhaps with the help of a static program analysis.

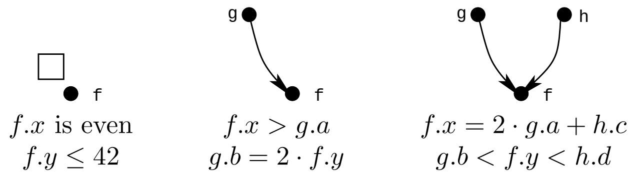


Figure 3.1: Examples of data-flow relations over one, two, and three program points.

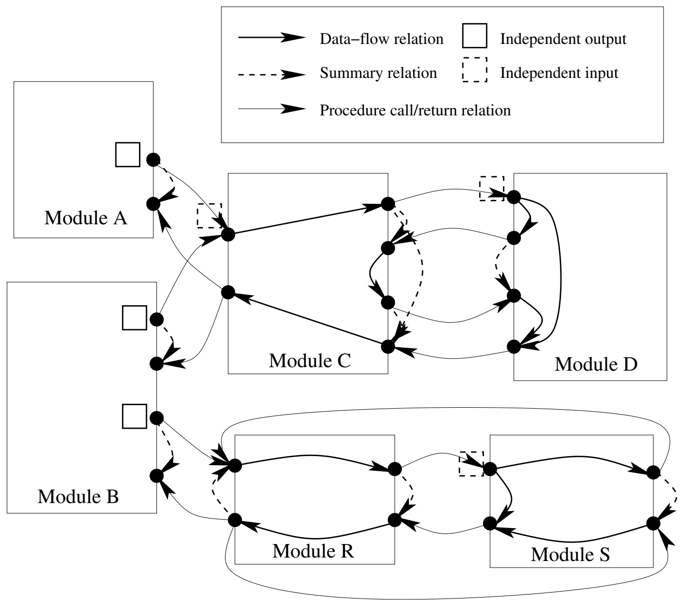


Figure 3.2: Examples of modules and relations

For summary relations, as with data-flow relations, developers may decide a priori not to consider the possibility of some outputs affecting a given input. Such restrictions are hard to justify in general: one could construct an environment for a module that realized almost any connection between module outputs and other inputs. Usually only a few such connections, however, are likely to have any influence on system correctness. Most commonly, a summary relation should exist between a call to a procedure outside the module, and the return value of that call. Other possibly relevant relations might connect the return of a method that constructs or mutates an object to the entrance of a method that accesses it, or the call to an external procedure taking a callback to the entrance of the callback procedure (when this relationship can be determined or approximated statically).

### 3.1.5 Graphical representation

In explaining which conditions must be checked to check the safety of an upgrade, it is helpful to represent the previous relational description of modules as a directed graph, in which nodes correspond to program points (module inputs and outputs), and edges correspond to relations. Specifically, each relation corresponds to zero or more edges, from each input to the output for a data-flow relation, and from each output to the input for a summary relation. We call the edges so created data-flow edges and summary edges, respectively. If an input or output is independent, then the relation is associated directly with the relevant node. Also, procedure calls and returns are represented by edges in the direction of control flow. For illustration purposes, we draw such graphs with rectangles representing modules; solid arrows between modules representing procedure call and return edges; bold arrows inside modules and bold solid boxes representing data-flow edges and independent outputs; and dotted arrows and dotted boxes representing summary edges and independent inputs. Figure 3.1 shows the representation of relations, and Figure 3.2 gives them in the context of a system composed of multiple modules.

## 3.2 Considering an upgrade

So far, we have described a model of the behavior of a modular system. For each module, the associated external summary relations represent assumptions about how the module has been used, and subject to those assumptions, its internal data-flow relations describe its behavior. Now, suppose that one or more modules in the system are replaced with new versions. We presume that each new module has been tested, and that in the context of this test suite new sets of data-flow and summary relations have been created. Under what circumstances do we expect that the system will continue to operate as it used to, using the new components in place of their previous versions? We replace the models of old components with models created during testing, and must check that this upgraded model is consistent. The key is obeying the summary relations.

In short, we must check that the assumptions embodied in each external summary relation are preserved: both those in the new component (so we know that the component will only be used in ways that exercise tested behavior) and those in other modules (so we know that the rest of the system will continue to behave as expected). Each summary relation summarizes the relationship between zero or more outputs and an input, which might be mediated by the interaction of many other relations in the system. The summary relation will be obeyed if every tuple of values consistent with the rest of the relations in the system is allowed by the summary; in other words, if its abstraction as a formula is a logical consequence of the combination of all the other relevant relation formulas. The system as a whole will behave as expected if all of the summary relations can be simultaneously satisfied given all the data-flow relations. For each summary relation, we construct a logical combination of the relevant data-flow relations, describing the states in which the data flow relations could simultaneously be satisfied. If this combination logically entails the summary relation, we can be confident that the summary relation will hold in the upgraded program.

Our algorithm for computing a consistency condition has two purposes. First, it determines how to connect data-flow relations to model a system's control flow. One might expect control flow modeling to be straightforward: for instance, sequential execution of code simply corresponds to conjunction of the corresponding flow relations. However, control flow join points (which occur at procedure entrances) require disjunction, or equivalently in our approach, the distribution of checking obligations over multiple paths. This construction of a consistency condition is similar in effect to the construction of verification conditions to check whether a program satisfies properties based on its implementation, as by weakest precondition / strongest postcondition predicates [Dij75, FS01] or symbolic evaluation [NL98]. However, we operate at the granularity of modules rather than of statements.

Second, the consistency condition includes only data-flow relations that might play a role in checking a summary relation, when deciding which assumptions to supply to a theorem prover. This is just an optimization, but it is an important one because automatic theorem provers are generally not effective at ignoring irrelevant hypotheses. This aspect of our technique resembles a backward slice [Tip95]; our use of a functional representation that combines control flow with data dependence is reminiscent of the slicing algorithm of [Ern94].

### 3.2.1 Feasible subgraphs

To describe which relations must be checked to verify that a summary relation holds, we define the concept of a *feasible subgraph* for a given summary relation. Roughly speaking, a feasible subgraph captures a subset of system execution over which a summary relation should hold. A summary relation may have many corresponding feasible subgraphs. An upgrade is safe if it allows each summary relation to hold over every corresponding feasible subgraph.

To obtain a single feasible subgraph for a given summary relation, use the following backward marking algorithm on the graph describing the relations of a system. (This algorithm, similar to a form of context-free language graph reachability [RHS95], is given merely to clarify the concept. It could be extended into a search algorithm that produces all feasible subgraphs, but below we discuss techniques for more efficient implementation.)

Starting with no nodes marked and no relations in the subgraph, mark the input of the given summary relation. Then, until no more nodes can be marked, repeat the following:

(a) If the output of a data-flow relation is marked, mark all the corresponding input nodes, and add the relation to the feasible subgraph.

(b) If the return value input node of a procedure return edge is marked, mark the exit point output node.

(c) If a procedure entry input node is marked, and a return node connected to the same procedure's exit output node is marked, then mark the procedure call output node for that procedure in the module with the return node.

(d) If a procedure entry input node is marked, and none of the corresponding call nodes or any of the return nodes connected to the same procedure's exit output node are marked, then choose one procedure call node connected to the entry and mark it.

The above algorithm describes a feasible subgraph as consisting only of data-flow relations (including independent outputs). One might also imagine including call and return edges, but we will adopt the convention that the identity between formal parameters and actual arguments entailed by the call edges, and the similar identity for return values, are represented implicitly by giving the same names to both sets of variables.

For a summary relation to be satisfied in an upgraded system, it must be guaranteed by each possible corresponding feasible subgraph. Representing each relation as a logical formula that must hold over certain variables, we can express this consistency condition for a summary relation $\overline{M}_0(v_0|u_1,\ldots,u_k)$ as

$$\bigwedge_{\substack{\text{feasible } G \\ \text{for } \overline{M}_0(v_0|u_1,\ldots,u_k)}} \left[ \left( \bigwedge_{M_i(v_i|\ldots)\in G} M_i(v_i|\ldots) \right) \Rightarrow \overline{M}_0(v_0|u_1,\ldots,u_k) \right] \tag{3.1}$$

In other words, for each feasible subgraph, the conjunction of the formulas representing data-flow relations in the subgraph must imply the formula for the summary relation.

A direct evaluation of the consistency condition for an upgrade, as described above, would be potentially inefficient, performing unnecessary logical comparisons. In particular, there may in the worst case be exponentially many feasible subgraphs, but it is not necessary to evaluate each one individually. Three techniques can be used to evaluate an upgrade's safety more efficiently. First, if all of the relations that should be checked to verify a summary relation are unchanged since the previous version of the system, they do not need to be rechecked: checks only need to consider what is being upgraded. Second, if the subgraph to be checked has a smaller subgraph that corresponds to a summary relation that has already been checked, an implementation can substitute that summary relation for the conjunction of those subgraph relations, since it has already been verified to be a consequence of them. If this implication is verified, then the summary relation is satisfied. If this implication fails, an implementation should fall back to using the conjunction of the data-flow relations, since they may be logically stronger than the summary. Such double checking should be rare in practice. Third, the feasible subgraphs for a summary relation may share some data-flow relations. Rather than evaluate each subgraph separately, the conjunctions for subgraphs that share relations can be combined into a single formula by eliminating repeated conjuncts and combining the remaining conjuncts as disjuncts, according to the identity

$$(A \wedge C \Rightarrow D) \wedge (B \wedge C \Rightarrow D) \iff ((A \vee B) \wedge C \Rightarrow D)\ .$$

This merging of feasible subgraphs is important to reduce the total number of graphs that must be evaluated.

The relation model as described is context-insensitive. A single relation includes information about all the inputs that might influence an output, even if some of them are mutually exclusive, as the different call sites of a procedure are: any particular time a procedure is invoked, its results depend upon the values at only one of its call sites. If there really is a difference in the behavior in different contexts, such context sensitivity can still be represented internally to the relation by using logical formulas that are conditional. For instance, when properties are discovered using the Daikon dynamic invariant detection tool, Daikon can search separately for properties that hold on the subsets of input values corresponding to distinct call sites, and express those differences as properties that are conditional on the values of the inputs.

A related imprecision of this model is that a single feasible subgraph cannot separately represent distinct traversals of a data-flow edge. If a procedure is used in different ways by two distinct modules within a single feasible subgraph, or if two procedures in different modules are mutually recursive, the consistency condition may contain a contradiction. A partial solution would be to duplicate a module to separate distinct uses, but duplication can potentially be expensive, it is inapplicable in the case of recursion, and simple duplication will be incorrect if multiple using modules interact via state in the duplicated module.

### 3.2.2 Special case: upgrading a functional procedure

The upgrade condition for a system of two modules and a single procedure in one module called from the other, described in Chapter 2, is a special case of the more general framework described in this chapter.

Consider a system with two modules, $U$ and $C$, where $C$ is a third-party component that defines a procedure `f`, and $U$ calls `f`. Further, suppose that each call to $f$ is independent. In our model, $C$ would have an independent input $\overline{C}(f)$ describing the preconditions of `f`, and a data-flow relation $C(f'|f)$ describing the postconditions of `f`, both based on the vendor's testing of $C$. Conversely, $U$ has an independent output $U(f)$ of preconditions describing how it calls `f`, and a summary relation $\overline{U}(f'|f)$ describing the postconditions it expects from the call.

The technique of this chapter claims that the new component may be safely substituted for the old one in the application if and only if

$$U(f) \Rightarrow \overline{C}(f) \qquad \text{and} \qquad (U(f) \wedge C(f'|f)) \Rightarrow \overline{U}(f'|f) \enspace .$$

Except for the use of a more general notation, this is the same formula as equation (2.1) on page 10.

# Chapter 4

# Examples of upgrades to more complex systems

The framework for upgrade safety checks described in Chapter 3 generalizes that of Chapter 2 to be applicable to more complex software systems. Specifically, while the previous technique is described for a system of two modules and a single procedure in one module called from the other, the new framework is more general in three aspects: it can model modules with state and multiple interacting procedures, it can model interactions between modules in which procedure calls are made in both directions, and it applies to systems with more than two modules. The following subsections illustrate these capabilities with simple concrete examples of each of these new possibilities. In each case, the determination of which relationships to check was made automatically using a implementation of the unoptimized algorithm described in Sect. 3.2; an abstraction including the properties shown was discovered by the Daikon tool; and the verification of all of the required properties, including ones not shown, was performed by the Simplify automatic theorem prover. The verification, requiring the proof of hundreds of properties, took a total of less than one second for each example. For brevity, we show shortened operational abstractions with a representative fraction of the actual properties. We also do not show the complete code, nor do we show the necessary test suites for the applications or the new modules.

## 4.1 Modules whose procedures share state

Figures 4.1 and 4.3 represent a system in which one module, $C$, makes available two procedures whose behavior is interdependent: the value returned by `get` depends on the value that was previously supplied to `set`. Such dependencies often arise when methods share state in an object instance, but our approach is

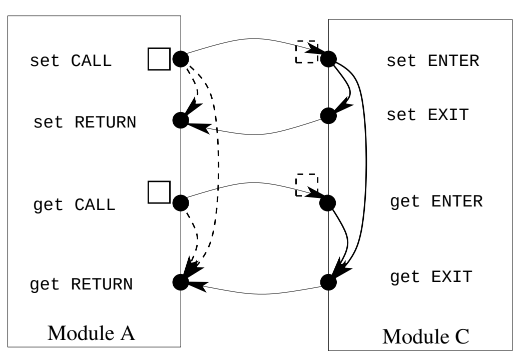


Figure 4.1: A system with a module $C$ whose procedures share state

$$
\begin{aligned}
A(s_c) &\Rightarrow \overline{C}(s) \\
A(s_c) \wedge C(s'|s) &\Rightarrow \overline{A}(s_r|s_c) \\
A(g_c) &\Rightarrow \overline{C}(g) \\
A(s_c) \wedge A(g_c) \wedge C(g'|s,g) &\Rightarrow \overline{A}(g_r|s_c,g_c)
\end{aligned}
$$

Figure 4.2: Consistency conditions, derived from equation 3.1 of Section 3.2.1, for the system shown in Figure 4.1; $s$ and $g$ represent the `set` and `get` procedures.

```
public class C {
  private int private_x;

  int set(int x) {                    int get() {
    private_x = x;                      return private_x + 1;
    return 0; // success              }
  }
}
```

Figure 4.3: Source code for a module with the structure of $C$ from Figure 4.1.

| | | | |
|---|---|---|---|
| $A(s_c)$: | $s.x$ is even | $\overline{C}(s)$: | $s.x$ is an integer |
| $\overline{A}(s_r\|s_c)$: | $s'.return = 0$ | $C(s'\|s)$: | $s'.return = 0$ |
| $A(g_c)$: | *true* | $\overline{C}(g)$: | *true* |
| $\overline{A}(g_r\|s_c,g_c)$: | $g'.return = s.x + 1$ | $C(g'\|s,g)$: | $g'.return = s.x + 1$ |
| | $g'.return$ is odd | | |

Figure 4.4: Operational abstractions for $A$ and $C$ as in Figure 4.1. Variables are prefixed according to the procedure they belong to. For instance, $s'.return$ is the return value of `set`, while $g'.return$ is the return value of `get`.

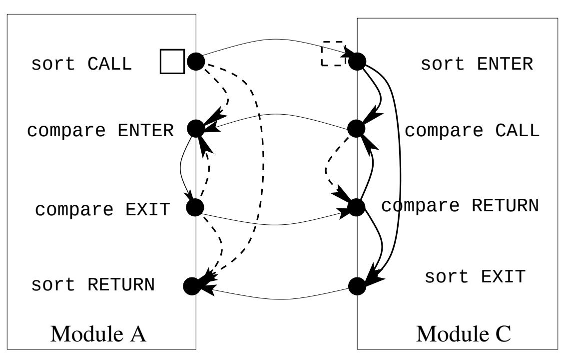


Figure 4.5: A system with a module $C$ that calls back to the using module $A$.

$$
\begin{aligned}
A(s_c) &\Rightarrow \overline{C}(s) \\
A(s_c) \wedge C(c_c|s,c_r) \wedge A(c'|c) &\Rightarrow \overline{A}(c|s_c,c') \\
A(s_c) \wedge C(c_c|s,c_r) \wedge A(c'|c) &\Rightarrow \overline{C}(c_r|c_c) \\
A(s_c) \wedge C(c_c|s,c_r) \wedge A(c'|c) \wedge C(s'|s,c_r) &\Rightarrow \overline{A}(s_r|s_c,c')
\end{aligned}
$$

Figure 4.6: Consistency conditions, derived from equation 3.1 of Section 3.2.1, for the system shown in Figure 4.5; $s$ and $c$ represent the `sort` and `compare` procedures.

```
public class A {
  private static class MyCompare
      implements Compare {
    public int compare(int x, int y) {
      return (x > y) ? 1 : (x < y) ? -1 : 0;
    }
  }
  ...
    C.sort(employee_ids, new MyCompare());
  ...
}
```

```
public class C {
  void sort(int[] a, Compare comp) {
    for (int i = a.length - 1; i > 0; i--)
      for (int j = 0; j < i; j++)
        if (comp.compare(a[j], a[j+1]) > 0) {
          int temp = a[i];
          a[i] = a[j];
          a[j] = temp;
        }
    }
}
```

Figure 4.7: Source code for a module $C$ and part of a module $A$ with the structure shown in Figure 4.5.

$$
\begin{array}{rl}
A(s_c)\colon & \forall i \in s.a\colon i \geq 1000 \\
\overline{A}(c|s_c, c')\colon & c.x, c.y \in s.a \\
 & c.x, c, y \geq 1000 \\
A(c'|c)\colon & c'.\mathit{return} \in \{-1, 0, 1\} \\
 & c'.\mathit{return} < c.x, c.y \\
\overline{A}(s_r|s_c, c')\colon & \forall i \in s'.a\colon i \in s.a \\
 & \forall i \in s.a\colon i \in s'.a \\
 & \forall i \in s'.a\colon i \geq 1000
\end{array}
\qquad
\begin{array}{rl}
\overline{C}(s)\colon & \forall i \in s.a\colon i \geq -2^{31} \\
C(c_c|s, c_r)\colon & c.x, c.y \in s.a \\
 & \\
\overline{C}(c_r|c_c)\colon & c'.\mathit{return} \in \{-1, 0, 1\} \\
 & \\
C(s'|s, c_r)\colon & \forall i \in s'.a\colon i \in s.a \\
 & \forall i \in s.a\colon i \in s'.a
\end{array}
$$

Figure 4.8: Operational abstractions for $A$ and $C$ as in Figure 4.5.

independent of how the state is recorded. To model this dependency, a data flow edge connects the entrance of the `set` procedure to the exit of the `get` procedure; symmetrically, we presume that module $A$ expects this relationship, as indicated by the summary edge connecting the call of `set` and the return of `get`. For the upgrade of module $C$ to be behavior preserving, the four implications shown in Figure 4.2 must hold. For instance, consider a behavior-preserving upgrade to $C$, which has been well-tested on its own, but suppose that module $A$ happens to only call `set` with even integers. The operational abstractions shown in Figure 4.4 describe this situation, and it can be seen that the consistency conditions shown in Figure 4.2 in fact hold. For instance, consider the last condition: if $s.x$ is even, and $g'.\mathit{return} = s.x + 1$, then $g'.\mathit{return}$ will be odd.

## 4.2 Modules with callbacks

Figure 4.7 shows an excerpt of source code from a system with two modules with calls between the modules in both directions: $A$ calls $C$'s `sort` procedure, which calls back to the `compare` procedure defined in $A$. Figure 4.5 models this system conservatively with respect to changes in $C$, by including each possible data-flow edge in $C$ and corresponding summary edge in $A$: the arguments passed to `compare` might depend on the results of the previous call, as well as the arguments to `sort`, and the results of `sort` potentially depend not only on its arguments but also on the results of the most recent call to `compare`. Here the callback is encapsulated in an object, but the same model could describe a callback passed by a function pointer. A change to this system is behavior-preserving only if the implications shown in Figure 4.6 hold. For instance, the left-hand column of Figure 4.8 gives an operational abstraction for $A$, which happens to only sort four-digit employee identification numbers. The right-hand column gives an operational abstraction for a well-tested behavior-preserving upgrade to $C$ (for instance, a change to the sorting algorithm). Note that not all of the possible relations corresponding to edges in Figure 4.5 were observed: for instance, calls to `compare` were not inter-dependent. Again, we can easily see that conditions of Figure 4.6 hold. Considering the last line, if every element of $s.a$ is at least 1000, and every element of $s'.a$ is also a member of $s.a$, then every element of $s'.a$ is also at least 1000.

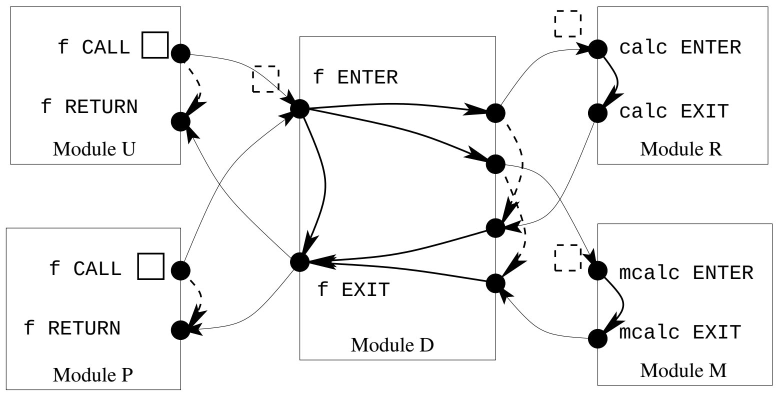


Figure 4.9: A system consisting of five modules

$$
\begin{array}{rcl}
(U(f_c) \vee P(f_c)) & \Rightarrow & \overline{D}(f) \\
(U(f_c) \vee P(f_c)) \wedge D(c_c|f) & \Rightarrow & \overline{R}(c) \\
(U(f_c) \vee P(f_c)) \wedge D(M_c|f) & \Rightarrow & \overline{M}(m) \\
(U(f_c) \vee P(f_c)) \wedge D(c_c|f) \wedge R(c'|c) & \Rightarrow & \overline{D}(c_r|c_c) \\
(U(f_c) \vee P(f_c)) \wedge D(m_c|f) \wedge M(m'|m) & \Rightarrow & \overline{D}(m_r|m_c) \\
U(f_c) \wedge D(c_c|f) \wedge R(c'|c) \wedge D(m_c|f) \wedge M(m'|m) \wedge D(f'|f, c_r, m_r) & \Rightarrow & \overline{U}(f_r|f_c) \\
P(f_c) \wedge D(c_c|f) \wedge R(c'|c) \wedge D(m_c|f) \wedge M(m'|m) \wedge D(f'|f, c_r, m_r) & \Rightarrow & \overline{P}(f_r|f_c)
\end{array}
$$

Figure 4.10: Consistency conditions, derived from equation 3.1 of Section 3.2.1, for the system shown in Figure 4.9. $f$, $c$, and $m$ represent the `f`, `calc`, and `mcalc` procedures respectively.

## 4.3 More than two modules

Figure 4.11 shows an excerpt of pseudocode from a client-server system for performing simple arithmetic. Modules $R$ and $M$ each perform two calculations in response to requests dispatched by module $D$. These services are used by two modules $U$ and $P$, making a system of five modules with the structure shown in Figure 4.9. In this example, the dispatch is performed explicitly, but a similar model could be used for dynamic dispatch as in an object-oriented language, given the sets of potential method targets. The conditions needed to verify the behavioral compatibility of a change to this system are shown in Figure 4.10 (each condition containing a disjunction was formed by combining the conditions for two feasible subgraphs). Now, suppose that we wish to upgrade module $U$, and the new version $U_2$ requires a new version $R_2$ of module $R$, in which the behavior of the rounding operation has changed to round negative values toward negative infinity rather than toward zero. Because the change to $R$ is incompatible, both modules must be replaced simultaneously. A similar simultaneous upgrade would be needed whenever two components, say a producer and a consumer of data, change the format they use without a change to the module mediating between them.

By checking the conditions of Figure 4.10 using the operational abstractions shown in Figure 4.12 (with the new $R_2$ and $U_2$), we can see that such an upgrade will be behavior preserving. $U_2$ will function correctly because $R_2$ provides the functionality it requires, and $P$ will function correctly because the functionality it uses (on non-negative integers only) was unchanged. The data-flow edges in $D$ from $f$ to $c_c$ and $m_c$ show an application of the guarding technique described in Chapter 3 on page 14: observe that the corresponding properties in Figure 4.12 are implications whose antecedents are properties over a variable of $f$.

```
public class D {  // Dispatches to M or R
  static int f(String op, int input) {
    if (op.equals("double") ||
        op.equals("triple"))
      return M.mcalc(op, input);
    else if (op.equals("increment") ||
             op.equals("round"))
      return R.calc(op, input);
  }
}
```

```
public class R {  // Rounds or increments
  static int calc(String op, int input) {
    if (op.equals("round"))
      // In version 2, changed to:
      // return 10 * (int)Math.floor(input / 10.0);
      return 10 * (input / 10);
    else if (op.equals("increment"))
      return input + 1;
  }
}
public class M {  // Multiplies by 2 or 3
  static int mcalc(String op, int input) {
    if (op.equals("double"))
      return 2 * input;
    else if (op.equals("triple"))
      return 3 * input;
  }
}
```

Figure 4.11: Java-like pseudocode for modules $D$, $R$, and $M$ as in Figure 4.9.

| | | | |
|---|---|---|---|
| $U(f_c)$: | $f.o \in \{\mathtt{d}, \mathtt{i}, \mathtt{r}\}$ | $\overline{D}(f)$: | $f.o \in \{\mathtt{d}, \mathtt{i}, \mathtt{r}, \mathtt{t}\}$ |
| $P(f_c)$: | $f.i \geq 0, f.o \in \{\mathtt{i}, \mathtt{r}, \mathtt{t}\}$ | | |
| $D(c_c \mid f)$: | $f.o \in \{\mathtt{i}, \mathtt{r}\} \Rightarrow (f.o = c.o, f.i = c.i)$ | $\overline{R}(c)$: | $c.o \in \{\mathtt{i}, \mathtt{r}\}$ |
| $D(m_c \mid f)$: | $f.o \in \{\mathtt{d}, \mathtt{t}\} \Rightarrow (f.o = m.o, f.i = m.i)$ | $\overline{M}(m)$: | $m.o \in \{\mathtt{d}, \mathtt{t}\}$ |
| $R(c' \mid c)$: | $c.o = \mathtt{r} \Rightarrow c'.return \equiv 0 \pmod{10}$<br>$c.o = \mathtt{i} \Rightarrow c'.return = c.i + 1$<br>$c.i \geq 0 \Rightarrow c'.return \geq 0$ | $\overline{D}(c_r \mid c_c)$: | $c.o = \mathtt{r} \Rightarrow c'.return \equiv 0 \pmod{10}$<br>$c.o = \mathtt{i} \Rightarrow c'.return = c.i + 1$ |
| $M(m' \mid m)$: | $m.o = \mathtt{d} \Rightarrow m'.return = 2 \cdot m.i$<br>$m.o = \mathtt{t} \Rightarrow m'.return = 3 \cdot m.i$ | $\overline{D}(m_r \mid m_c)$: | $m.o = \mathtt{d} \Rightarrow m'.return = 2 \cdot m.i$<br>$m.o = \mathtt{t} \Rightarrow m'.return = 3 \cdot m.i$ |
| $D(f' \mid f, c_r, m_r)$: | $f.o \in \{\mathtt{i}, \mathtt{r}\} \Rightarrow f'.return = c'.return$<br>$f.o \in \{\mathtt{d}, \mathtt{t}\} \Rightarrow f'.return = m'.return$ | $\overline{U}(f_r \mid f_c)$: | $f.o = \mathtt{d} \Rightarrow f'.return = 2 \cdot f.i$<br>$f.o = \mathtt{i} \Rightarrow f'.return = f.i + 1$<br>$f.o = \mathtt{r} \Rightarrow f'.return \equiv 0 \pmod{10}$ |
| | | $\overline{P}(f_r, f_c)$: | $f.o = \mathtt{i} \Rightarrow f'.return = f.i + 1$<br>$f.o = \mathtt{r} \Rightarrow f'.return \equiv 0 \pmod{10}$<br>$f.o = \mathtt{t} \Rightarrow f'.return = 3 \cdot f.i$<br>$f'.return \geq 0$ |
| $U_2(f_c)$: | $f.o \in \{\mathtt{d}, \mathtt{i}, \mathtt{r}\}$ | $\overline{R}_2(c)$: | $c.o \in \{\mathtt{i}, \mathtt{r}\}$ |
| $R_2(c' \mid c)$: | $c.o = \mathtt{r} \Rightarrow c'.return \equiv 0 \pmod{10}$<br>$c.o = \mathtt{r} \Rightarrow c'.return \leq c.i$<br>$c.o = \mathtt{i} \Rightarrow c'.return = c.i + 1$<br>$c.i \geq 0 \Rightarrow c'.return \geq 0$ | $\overline{U}_2(f_r \mid f_c)$: | $f.o = \mathtt{d} \Rightarrow f'.return = 2 \cdot f.i$<br>$f.o = \mathtt{i} \Rightarrow f'.return = f.i + 1$<br>$f.o = \mathtt{r} \Rightarrow f'.return \equiv 0 \pmod{10}$<br>$f.o = \mathtt{r} \Rightarrow f'.return \leq f.i$ |

Figure 4.12: Operational abstractions for modules in Figure 4.9. The arguments `op` and `input` are abbreviated $o$ and $i$, and the values `double`, `increment`, `round`, and `triple` are abbreviated to their initial letters. The abstractions labeled $U_2$ and $R_2$ represent a potential upgrade to the $U$ and $R$ modules respectively.

# Chapter 5

# Improvements to the technique

Even in the restricted domain considered there, the basic technique described in Chapter 2 runs into some practical limitations. In this chapter, we describe enhancements that make our technique more effective in validating upgrades to realistic software systems. We describe three broad classes of improvements: first, changes to make more information about a program's behavior available to our system, which can only improve the accuracy of its results; second, techniques that attempt to distinguish among detected behavioral differences, choosing a subset as most relevant to upgrade safety; third, a way to use our technique in the absence of a large test suite.

## 5.1 Making more information available

The enhancements described in Chapter 3, extending our technique to systems with many modules, work by propagating information along control flow paths so that it can take part in an upgrade comparison. Often, it also helps to include information associated indirectly with the objects being processed, as this section describes.

### 5.1.1 Including non-local state information

In order to verify that an upgraded module will still produce the desired outputs, our technique must capture, on at least a superficial level, how those outputs are a function of inputs. Sometimes, all the inputs that determine a subroutine's behavior are supplied as parameters, or for an object method, as fields of the object. In practice, however, the information that a routine uses may be more widely dispersed. For instance, in the Unix system-call interface, functions like `open` and `close` conceptually create and destroy stateful 'file descriptor' objects. Because the kernel and user-mode applications reside in separate address spaces, however, these file descriptors are referred to from applications not as pointers to objects, but as small integer indices, corresponding to a table that exists only in the kernel.

This sort of extra information can be thought of as residing in virtual fields, containing information that is closely related to the meaning of an object, but happens not to be kept directly in the object in a particular implementation; then it can participate in the operational abstraction just like an object's actual fields. For our technique to work well, the operational abstractions describing a method should include all the relevant information, but too much irrelevant information will slow down processing and potentially lead to reports of differences that are irrelevant to correctness. Our technique must use heuristics or human guidance to add only relevant information to operational abstractions.

In object-oriented programs, the output of the program's own (pure) accessor methods can be used to reflect application-specific semantic distinctions. (We implemented this technique for the Perl front end of Daikon, but it was not used in the case studies described in Chapter 7.) When less information is embodied in the code, it must be supplied with some manual help. For the case of Unix system calls, a simple annotation could associate the file-descriptor pseudo-datatype with `fstat`, a function that returns a variety of relevant information about a file.

## 5.2 Selecting relevant differences

This section describes two techniques to refine the set of observed behavioral differences that our tool presents to its users. These techniques reduce the number of warnings produced by our technique by discarding warnings that are less likely to represent actual incompatibilities. The first technique discards observed differences that result from behavior that is impossible for our analysis to predict. The second technique discards differences that were observed to occur identically when using both the old and new versions of a component. If developer time were unlimited, there would be no reason to omit any warning, since any warning might, under some circumstances, represent a difference that should be addressed. Given that developer time is limited, however, choosing only those differences most likely to represent serious problems makes our technique more useful.

### 5.2.1 Distinguishing non-deterministic differences

Section 5.1.1 describes how our technique can work on software whose behavior is determined by information elsewhere in the programming system. In some cases, however, a program's behavior may depend on information that is completely inaccessible. While it is relatively rare for software to be unpredictable in this way under normal operation, non-determinism is more often associated with errors.

One type of behavioral difference that might cause our technique to reject an upgrade is if a return value representing an error occurred during testing but never in an application's use. For instance, suppose that a particular method was tested to sometimes throw a divide-by-zero error, but the old version never did so in the context of an application. An upgrade will be judged unsafe unless our technique can determine, based on the method's operational abstraction, that the error was only thrown for some combination of inputs that the application never supplied. In theory, any error can be described as a special case in the behavior of a method, where that behavior is defined over a suitably broad set of inputs. For instance, if the previous contents of a hard disk are considered an input to a "write file" routine, whether that routine fails by running out of disk space can be determined based on the available space and the size of the file. This is not always the most natural description, though, and in systems that interact with the physical world, the behavior of a module might depend on facts that are completely inaccessible to the computer. For instance, if a program is reading data from the network or a disk, a physical fault like a broken cable or dust on a floppy disk might cause a method to fail under conditions that are completely unpredictable even taking into account the computer's entire state.

Because it would be unreasonable or impossible to predict when such errors might occur, they must be treated separately. Such errors represent a special case of a method's output, but rather than trying to determine the circumstances under which they might occur, as usual, we instead unsoundly assume that if they never occurred with the old component, they will never occur with the new component either. In some languages, such as Java, the circumstances that should get special treatment are analogous to a language feature: Java exceptions represent special cases that violate a method's usual contract of pre- and post-conditions, and should be treated specially by callers. For other languages, such a C, this information will not be explicit in the program structure, and must be instead be supplied by some other mechanism, such as annotations.

### 5.2.2 Highlighting cross-version differences

Even with the improvements described in the previous sections, our technique is incomplete, in the sense that there are some valid, behavior-preserving upgrades the safety of which it is unable to verify. To minimize the burden on users of the tool of such failures, we propose to effectively post-process the results of the technique as described so far, to highlight for urgent consideration those detected differences that result from a change between the module versions considered, and deemphasize those differences that may simply be a result of the technique's limitations. Such failures, in which the tool is unable to verify the safety of an upgrade that is in fact behavior-preserving, might be the result of insufficient testing, an inadequate grammar of the operational abstraction, or a theorem proving weakness.

If the vendor's test suite is inadequate, it may not describe enough of the module's behavior to assure the application's correct functioning, even if that behavior has not changed. For instance, in the example of the `swap` module from Section 2.2 (page 6), suppose that the module had been modified in a way that preserved its correctness even for attempts to swap an element with itself. As long as the vendor's test suite did not check the module's behavior in this case, the upgrade would still be considered unsafe for any application that exercised that behavior.

If the grammar of the operation abstractions is inadequate, it might be able to express a property of the module's behavior in the special case of an application, but not in the general case of the test suite. For instance, previous versions of Daikon created abstractions over slices of an array that started at the beginning of the array and stopped somewhere in the middle, and slices that started in the middle and went to the end, but not more general slices. If this version of Daikon were used to create an abstraction describing the swap routine, it would contain the facts that the routine leaves the slice $a[0 \dots i-1]$ unchanged, and the slice $a[j+1 \dots]$, but it would not record that the slice $a[i+1 \dots j-1]$ was also unchanged. If an application always used `swap` to exchange elements whose indices differed by 2, its abstraction might state that the value at $a[i+1]$ was always unchanged, but there would be no corresponding fact in the abstraction of the tested module to use to prove this statement. (This phenomenon is explored experimentally in Section 8.3 on page 39).

Finally, it may be the case that the reasoning required to determine that the upgrade is safe, particularly that the application postconditions follow from the application preconditions and the tested postconditions, is beyond the power of the particular theorem prover being used. For instance, the preconditions of swap, as used in selection sort, include that the slice $a[0 \dots i-1]$ is in order, and that the element at position $a[j]$ is at least as large as all the elements from $a[0]$ to $a[i-1]$. The application postconditions include the fact that the slice $a[0 \dots i]$ is in order. Together, the application preconditions and the fact that the `swap` routine interchanges $a[i]$ and $a[j]$ are enough to establish the application postcondition, but it may require some sophistication on the part of the theorem prover (for instance, reasoning by cases) to automatically verify this. For every automated theorem prover, there will be some logical implications it is unable to verify in a reasonable amount of time.

All of these cases represent a failure of our technique; they mean that there is some part of the module's behavior it is unable to fully reason about. However, it would be helpful to users for our tool to distinguish between cases where our technique does not have enough information to verify that an upgrade is safe, and when it has some particular information that implies an upgrade might be unsafe.

We propose performing an extra level of comparison to separate behavioral differences caused by such problems from differences caused specifically by the upgrade in question. If our technique does not approve an upgrade from an old module to a new module, then it considers an upgrade from the old module to itself (a *self-upgrade*). Such an "upgrade" is always behavior-preserving, since the behavior is unchanged. However, our technique might still fail to verify the upgrade's safety, for any of the reasons described in the above paragraphs. A failure that occurs only with the new module certainly represents a behavioral difference (we will call it a *cross-version difference*). On the other hand, if the failures for the self-upgrade are identical to the failures for the real upgrade, then those for the real upgrade may be as innocuous as those for the self-upgrade. Such failures represent an intermediate ground in which the technique was unable to support or refute the safety of an upgrade, and should lead to an appropriate intermediate level of scrutiny before the upgrade is applied. We used this technique to filter the results described in Chapter 8, but not those described in Chapter 7.

This postprocessing is effective no matter whether the abstractions describing the old component version are derived from the old or new versions of the component test suite. The operational abstraction most likely to be available for the old module is one based on the test suite current at the time of its release; in our scenario, it would have been supplied along with the old module. If the test suite has changed, however, better results can be obtained by testing the old module version with the new version's test suite. Using the new test suite with the old module allows the technique to better compensate for deficiencies existing only in the new test suite, or common to the application and the old test suite.

## 5.3 Using other applications as a test suite

Extensive testing is often an important part of software engineering practice in industry, but not all software has a large formal test suite. In particular, many open source software projects lack significant test suites: they may either perform less testing overall, or depend more on informal testing by developers and users. Even if tests are created, they may not be reliably collected and organized for future use. When an organized test suite is unavailable, the role of the "test suite" in our technique can instead be played by other applications that use a module. We use this technique in the case studies of Chapter 8. For each application, we consider whether an upgrade of the component would be safe, given as "testing" how the new version was used by all of the other applications being examined. This experiment design is analogous to the "late adopter" practice of letting of one's colleagues use a new software version, and potentially discover incompatibilities with their expectations, before using that new version oneself. This practice is only effective if your colleagues use the software in the same way you do. In the same way, it is only if the other applications have sufficiently exercised the library's behavior that an upgrade would be safe for a particular application. In addition to being useful to users, this technique lets us run experiments even in the absence of formal test suites. However, the testing achieved in this way is still less comprehensive than the results of formal testing, so the technique of Section 5.2.2 should also be used, to reduce the number of warnings that indicate only insufficient testing.

# Chapter 6

# Implementation details

This chapter provides additional information about how our prototype tool implements the comparison described in Chapter 2, about how we collected information from the components examined in the case studies, and about the tool that used that information to create an operational abstraction.

## 6.1 Implementing comparison with Simplify

Our tool uses the Simplify automatic theorem prover [NO80, DNS03] to evaluate the logical comparison formula described in Section 2.3.1 on page 10. The tool translates the operational abstractions into Simplify's input format, pushes all of the assumptions onto Simplify's background stack, and queries it regarding each conclusion property. Each class of property must be defined so that Simplify can reason about it (the basic properties of arithmetic and ordering are built in to Simplify). Most properties can be defined by rewriting them in first-order logic with uninterpreted function symbols representing operations like sequence indexing. For the most complicated properties, we supplemented Simplify with lemmas. For instance, one lemma states that a sequence $a$ is lexicographically less than a sequence $b$ if they have initial subsequences $a[0..i-1] = b[0..i-1]$ that match, followed by an element $a[i] < b[i]$. Because these lemmas are general, they need only be created once, when a new property is added to the abstraction's grammar. So far, we have had to write 52 lemmas, less than one per property in the grammar of our abstractions. Because the Daikon invariant detector generates simple properties, most of the implications Simplify must check are trivial. In our first case studies, the results of which are shown in Figure 7.1 on page 34, checking each property took only a fraction of a second.

## 6.2 Instrumenting Perl programs

For the case studies of CPAN modules, discussed in Chapter 7, we have implemented a new front end to the Daikon tool to instrument programs written in Perl. Because Perl code lacks type information, this tool operates in two passes, one to dynamically infer what type of value each variable might hold, and a second to record those values for use by Daikon. The type guessing operates in a manner somewhat analogous to Daikon itself: watching the execution of a program, the runtime system chooses the most restrictive type for a variable that is not contradicted during that execution, potentially generalizing the type as new values are seen. These types indicate, for instance, whether a scalar value always holds an integer, a possibly fractional numeric value, or a reference to another object. The ordering among these types (subset inclusion) is illustrated in Figure 6.1.

An additional quirk of Perl is that it has no specific language mechanism for named subroutine parameters: instead the front end searches for a common pattern of local variables holding parameters: a contiguous series of variables declared with `my` or `local` and, in the same expression, assigned from a value derived from a specially named array `@_`. Though inelegant, this technique works well in practice.

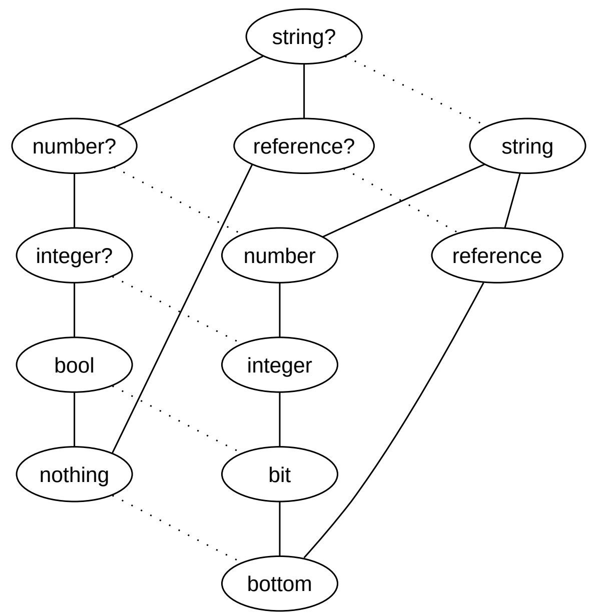


Figure 6.1: Lattice showing the inclusion ordering among subsets of the values that might occupy a Perl scalar variable. Bottom, nothing, and bit represent the sets $\varnothing$, $\{$`""`$, undef\}$ and $\{0, 1\}$. Dotted edges have the same semantics as solid ones, but emphasize that every type has both a normal version and one lifted by the addition of the nothing values.

## 6.3 Instrumenting C programs

For our case studies of C programs and libraries, discussed in Chapter 8, we put an existing tool for instrumenting C software to a new kind of use. The approach builds on previous work on instrumenting C code for use with Daikon, with the `dfec` tool, but takes a slightly different approach, because it would be impractical to instrument the entire library and all of an application using it as a closed system.

Compared to those in more constraining languages such as Java, C programs present a number of technical challenges to effective instrumentation. Three classes of difficulty are most troublesome. First, because C does not guarantee memory safety, an execution state may contain pointers that cannot be traversed. Second, because C has no requirement that data be initialized before becoming visible, an execution state may contain data that are not meaningful in the program's intended semantics. Finally, because C's basic data structures do not record their own sizes, it can be difficult to determine how much relevant information is present. A naive instrumentation tool for C programs would find meaningful relationships obscured by the second and third difficulties, but only if it was lucky enough to avoid the segmentation faults caused by the first.

Previous work on the Daikon front end for C, or `dfec` for short [Mor02], has addressed these challenges, but in way that is effective only under a closed-world assumption about the program being instrumented. The `dfec` tool works by rewriting a C program's source code into a C++ program. Built-in pointers and arrays in the C program are translated into objects with overloaded operators that simulate operations like indexing and dereferencing with additional bookkeeping and safety checks. Applied to a complete program, these techniques allow valid blocks of memory and pointers into them to be distinguished from invalid ones, uninitialized values to be distinguished from initialized ones, and the extent of valid data written into a buffer to be recorded. However, the thoroughness of this transformation makes it difficult for instrumented code to interact with uninstrumented code: instrumented pointers and arrays have a different memory layout than their pre-instrumentation counterparts, making pointer-based interfaces fail. Also, the subtle incompatibilities between C and C++ restrict what code can be instrumented.

For our case-study involving Linux applications and the Linux C library, it was impractical to use `dfec` as it was originally intended. Though `dfec` has been used successfully with moderately complex Unix applications, such as `flex`, this required manual modifications to the program's source code. Worse, the C library itself uses a number of low-level language features. Some library routines are implemented in assembly language, and depend on particular details of memory layout and the binary object format. Even before instrumentation, it appeared that it would be difficult to compile the C library with a C++ compiler. Instead, we used a technique that did not require either the library or the applications that use it to be modified.

We use shared library interposition [Cur94] to catch calls to procedures in the C library. Shared library interposition takes advantage of the fact that on modern Unix systems, libraries are usually dynamically linked. An application that uses the C library's `close` routine, for instance, does not contain the code for that routine in its binary: instead the static linker simply makes a note that the program requires a symbol with that name. At runtime, when `close` is first called, a run-time dynamic linker locates `close` in the C library, and modifies a function pointer so that subsequent calls to `close` use the library code. (Such libraries are generally referred to as "shared", referring to the memory usage benefit of this technique; for our purposes, it is more important that they are "dynamic". Libraries that are dynamic but not shared, or shared but not dynamic, are possible but rarely used.) Our interposing library also contains a routine named `close`, and we use the `LD_PRELOAD` environment variable to direct the runtime linker to load the interposition library before any libraries requested by the application. Requests by the application for procedures from the C library are directed instead to our library, which traces procedures arguments, passes them on to the real version of the procedure in the C library, traces that procedure's return value, then returns it to the application.

The use of interposition allows the instrumentation to be confined to a single library, but the interposing library would not function correctly if it were instrumented by `dfec` in the usual way, because the instrumented code expects they layout of objects in memory to be different than it is for uninstrumented code. For instance, `dfec` normally replaces each pointer with an object consisting of three word-length fields holding information such as the size of the object pointed to. In order to make the instrumentation library binary compatible with the applications and libraries it interacts with, we have created a new version of the classes used by instrumented code, redefining for instance such *smart pointers* to consist solely of a single normal pointer. Though it enables binary compatibility, this change makes it impossible for us to use `dfec`'s normal mechanisms for keeping track of memory usage (and most of them would not have operated correctly in any case when memory is modified outside their control). By maintaining the smart pointer interface, we can reuse `dfec`'s generated code for traversing objects and printing information about them to a trace file for Daikon's use, but we must replace the accessor methods that previously looked up, for instance, whether a pointer was valid or the length of the memory location it referred to.

Rather than keeping track of this information dynamically, as `dfec`'s runtime library does when used normally, we get it from two sources: the operating system and interface annotations. The most fundamental requirement for an instrumentation technique is that it not cause the instrumented code to crash, so it is important that the instrumentation library never dereference a pointer outside the process's address space. A simple way to avoid this is to simply ask the operating system whether a pointer is valid before dereferencing it. Though Linux does not provide a system call explicitly for this purpose, many system calls must check the validity of a user-supplied buffer before using it, and we take advantage of this side effect. Specifically, to test whether it would be safe to read $n$ bytes starting at a location $p$, we simply ask the kernel to do the same thing, by asking it to `write` $n$ bytes from the 'buffer' $p$ to a file. If the pointer is invalid, the system call will fail with an `EFAULT` error code; otherwise it will succeed.

The remaining information about the C library's memory use that our instrumentation library needs is provided by annotations, providing extra interface information that is not explicit in a C declaration. Because these annotations were simple and short, we found it easy enough to supply them by hand; however, similar information could also be obtained by static analysis. The memory-usage annotations we added were of two types: we marked reference parameters that were used only for input or only for output, and we marked arrays so that the instrumentation library could determine their lengths.

It is common for C library procedures to have parameters that are pointers to structures: potentially, such a structure can be used to supply information to the procedure or to allow the procedure to return extra information to the caller. In the most common cases, no annotation is required: the contents of the structure before the call are used to infer the procedure's preconditions, and its contents afterward are

| Annotation type | Number of annotations | Eligible procedures | Ratio |
|---|---|---|---|
| Total `libc` procedures | 1216 | — | — |
| Uninstrumented procedures | 227 | 1216 | 0.187 |
| Array lengths | 377 | 989 | 0.381 |
| null-terminated | 281 | 989 | 0.284 |
| other argument | 78 | 989 | 0.079 |
| constant | 14 | 989 | 0.014 |
| argument product | 4 | 989 | 0.004 |
| Argument types | 89 | 199 | 0.447 |
| file descriptor | 36 | 199 | 0.181 |
| file name | 32 | 199 | 0.161 |
| socket | 17 | 199 | 0.085 |
| time | 2 | 199 | 0.010 |
| time array | 2 | 199 | 0.010 |
| Input/output-only arguments | 18 | 199 | 0.090 |
| input-only | 2 | 199 | 0.010 |
| output-only | 16 | 199 | 0.080 |

Figure 6.2: Summary of annotations used for C, along with counts of the number of times each was used for the C library case study of Chapter 8. Each annotation count is compared to the number of procedures that were eligible for the annotation: we instrumented only a subset of the procedures, supplied array lengths only for instrumented procedures, and described argument types and input-only or output-only arguments only for procedures that were used by our applications. Uninstrumented procedures include those that use variable-length argument lists, function pointers, or complex numbers, 20 very-frequently called procedures like `strlen` that we omitted for performance reasons, and approximately 50 procedures that caused miscellaneous difficulties in our instrumentation implementation.

described by its postconditions. In particular, if the structure is not modified by the procedure, this will be reflected in the operational abstraction. However, some routines, such as `stat`, use a buffer that is allocated by the caller, but whose contents before the call are not meaningful. Since the buffer contents are often uninitialized, we annotate these procedures to avoid inferring properties that would reflect only the previous unrelated contents of such memory. A similar issue arises with procedures that take a pointer to a structure as input, operate on it, and then destroy it, say by deallocating its memory. For these procedures, we add an annotation stating that the contents of the structure after the call are to be ignored.

The most commonly required type of annotation for the C library was one describing the length of dynamic arrays passed to library procedures. Though potentially a dynamic array of any type might be passed, we only observed a need for the sizes of arrays of characters. By far the most common pattern is a character pointer representing a null-terminated string; we also instrumented several procedures for which the length of one argument is given by another argument, and four (`fread` and `fwrite`, in locked and unlocked forms) for which the length of a buffer is the product of two other arguments (`qsort` also works this way, but we did not instrument it because it uses a function pointer). Finally, because of a deficiency in the `dfec` tool, we had to annotate several arrays that had a statically specified length, supplying that length manually, because `dfec` replaces array arguments to procedures with pointers. This result could have been achieved fully automatically by a more sophisticated `dfec`.

A summary of the annotations used for C is shown in Figure 6.2. In addition to the memory safety annotations mentioned above, the table also includes annotations used to give arguments a more specific type (used for instance for the virtual fields of Section 5.1.1 on page 24), and the functions that we chose not to include in the case study of Chapter 8.

## 6.4 Generating operational abstractions

To derive an operational abstraction from the operation of a program on a test suite, we use Daikon, a tool that dynamically detects likely invariants. Dynamic invariant detection [Ern00, ECGN01] conjectures likely invariants from program executions by instrumenting the target program to trace the variables of interest, running the instrumented program, and generalizing the observed values. These conjectured invariants form an operational abstraction. In the Daikon implementation of dynamic invariant detection, the generalization step uses an efficient generate-and-test algorithm to winnow a set of possible properties; Daikon reports to the programmer those properties that it tests to a sufficient degree without falsifying them. Daikon works with programs written in C, Java, Perl, and IOA, and with input from several other sources. Daikon is available from `http://pag.csail.mit.edu/daikon/`.

Daikon detects invariants at specific program points such as procedure entries and exits; each program point is treated independently. The properties reported by Daikon encompass numbers ($x \leq y$, $y = ax + b$), collections ($x \in mytree$, *mylist is sorted*), pointers ($node = node.child.parent$), and conditionals (*if* $p \neq$ *null then* $p.value > x$). Daikon incorporates static analysis, statistical tests, logical inference, and other techniques to improve its output [ECGN00].

Generation of operational abstractions from a test suite is unsound: the properties are likely, but not guaranteed, to hold in general. As with other dynamic approaches such as testing and profiling, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases. When a reported invariant is not universally true for all possible executions, then it indicates a property of the program's context or environment or a deficiency of the test suite. In many cases, a human or an automated tool can examine the output and enhance the test suite, but this paper does not address that issue. Previous research has shown that the generated operational abstractions tend to be of good quality: they often match human-written formal specifications [Ern00, ECGN01] or can be proved correct [NE02a, NE02b], and even lesser-quality output forms a partial specification that is nonetheless useful [KEGN01], because it is much better than nothing.

# Chapter 7

# CPAN case studies

In a series of case studies, we first assessed the effectiveness of our approach for small but realistic upgrades by case studies involving pairs of Perl modules.

## 7.1 Currency case study

The first case study concerns code for manipulating monetary quantities.

### 7.1.1 Subject programs

In the first case study, the component is `Math::BigFloat` (`BigFloat` for short), a Perl module for arbitrary-precision floating point arithmetic that operates on numbers larger than the 32- or 64-bit formats provided in hardware. (`BigFloat` is bundled in a distribution, named Math-BigInt, comprising approximately 3500 lines of code, plus documentation and tests. The distribution also contains the `Math::BigInt` module for arbitrary-precision integer arithmetic, in terms of which `BigFloat` is implemented, plus several supporting modules. The modules within the distribution have their own version numbers, but for clarity we will always refer to the distribution version numbers.)

The application that uses `BigFloat` is a separately authored module, `Math::Currency`. `Currency` supports arithmetic under the conventions of monetary quantities, including special treatment of rounding and locale-specific output formats. `Currency` uses `BigFloat` to implement its underlying arithmetic operations on money quantities, taking particular advantage of `BigFloat`'s decimal, rather than binary, representation of fractions, in order to avoid rounding errors.

There are at least six bugs in various versions of `BigFloat` and its supporting modules that would lead to incorrect results being produced by the most recent version (1.39) of `Currency`, two of which are exposed by our case study. The author of `Currency` may have been aware of up to four of them: `Currency` 1.39 avoids those four by specifying that it should only be used with `BigFloat` version 1.49 or later. For the purposes of the study, we disabled this explicit check. Supplying metadata that specifies acceptable versions of a component can be effective in preventing some errors. However, such a dependence can only be found after system-scale testing, and relying on manual marking can mean some problems are missed, such as two that were fixed only in versions 1.51 and 1.55 respectively.

We investigated two pairs of versions of `BigFloat` — 1.40 with 1.42 (1.41 was not available), and 1.47 with 1.48 — to determine whether an upgrade from the earlier to the later version of the pair was permissible. We also consider a downgrade in the opposite direction. A downgrade might be desirable because of a bug in a later version, or might occur as a result of porting an application to a system that has an older version of a component installed.

| Module | Upgrade | Lines changed | Unsafe method | Props checked | Checking time (sec) |
|---|---|---|---|---|---|
| `Math::BigFloat` | 1.40→1.42 | 25 | `bcmp()` | 171 | 0.91 |
| `Math::BigFloat` | 1.42→1.40 | 25 | `bcmp()` | 163 | 1.00 |
| `Math::BigFloat` | 1.47→1.48 | 163 | — | 1130 | 4.77 |
| `Math::BigFloat` | 1.48→1.47 | 163 | — | 1130 | 4.59 |
| `Date::Simple` | 1.03→2.01 | 243 | `new()` | 12 | 1.04 |
| `Date::Simple` | 1.03→2.03 | 243 | `new()` | 12 | 1.29 |
| `Date::Simple` | 2.01→2.03 | 6 | — | 12 | 0.95 |
| `Date::Simple` | 2.01→1.03 | 243 | `new()` | 12 | 0.95 |
| `Date::Simple` | 2.03→1.03 | 243 | `new()` | 12 | 0.95 |
| `Date::Simple` | 2.03→2.01 | 6 | — | 12 | 0.94 |

Figure 7.1: Results of Perl module case studies. Changed lines include methods not exercised by our applications, but exclude documentation and tests. If no method is (potentially) unsafe, then the upgrade is judged to be behavior-preserving. Timings include translation of the properties into Simplify's input format and the running time of Simplify, on a 1.1 GHz AMD Athlon workstation.

### 7.1.2 Floating-point comparison

Between versions 1.40 and 1.42, three changes were made to `BigFloat`, all affecting the `bcmp` comparison routine. Two of these changes were bug fixes, correcting problems that caused incorrect results from currency operations. One bug caused distinct amounts having the same number of whole dollars to be considered equal, while another caused some unequal values of the same order of magnitude to have the wrong ordering.

The third change narrowed the behavior of `bcmp`: The new version of `bcmp` always returns $-1$ or 1 when its first argument is less (respectively, greater) than its second argument; by contrast, the old version of `bcmp` returns an arbitrary negative (respectively, positive) number. Both versions of `bcmp` return 0 when that their arguments are equal. This interface change was not reflected in the documentation: both before and after the change, `BigFloat`'s documentation indicated that `bcmp` could return any negative or positive value, while Perl's documentation specifies that the `<=>` operator, which `bcmp` overloads, always returns $-1$, 0, or 1.

Our approach concludes that neither an upgrade from 1.40 to 1.42 nor a downgrade from 1.42 to 1.40 is behavior-preserving. Having described the differences between the versions above, we now describe how our technique and tools discover those differences and conclude that both components are incompatible with one another. For our case study, we replaced the test suite distributed with `BigFloat` (which tests the basic routines with only a few dozen pairs of inputs) with simple randomized tests of single operators on larger numbers of inputs, and also exercised `Currency` with a simple randomized script.

The downgrade is (correctly) judged as incompatible for the following reason. The new component's abstraction restricts the comparison routine's output range ($return \in \{-1, 0, 1\}$), but no corresponding restriction appears in the abstraction generated from the old component, so the replacement is not compatible. For instance, an application that worked correctly with a later version of `BigFloat` might compare the output of `bcmp` directly to a literal `1` value, or check whether two pairs of numbers had the same ordering relationship with an expression of the form `($a <=> $b) == ($c <=> $d)`. These constructions would give incorrect results when used on a system where an earlier version of `BigFloat` was installed. Our tool does not rule out an upgrade based on the restrictions on `bcmp`'s range: the new component's properties are stronger than the old component's properties.

Our tool also (correctly) advises that the upgrade is not behavior-preserving, but for a different reason. The behavior changes it discovers are side effects of fixing a comparison bug in 1.40 and earlier versions. Specifically, the old comparison routine treated (for example) \$1.67 and \$1.75 as equal because both values were inadvertently truncated to 1. Because of an otherwise unrelated bug in the (decimal) right shift operation, this integer truncation converted values less than a dollar into a floating-point value with a corrupt mantissa, represented as an empty array rather than an array containing only a zero. Our comparison rejects the upgrade because it can tell that the application, using the old version of `BigFloat`, called the comparison routine with these malformed values, but that the test suite with the new version does not.

While the change to the return value of the comparison operator caused our tool to caution against a downgrade (unless further analysis indicates that the application does not depend on the more restricted behavior), the bug fix results in cautions against an upgrade. The bug fixes do make the two components incompatible, but the changes are improvements, and typically such changes are desirable. In the absence of a programmer-supplied specification (the presence of which, along with the operational abstraction, would have indicated the bug long before), it is impossible to know whether a change is desirable. In both cases, our tool outputs a list of program properties it cannot prove; these are the behavior changes that should be further investigated.

### 7.1.3 Floating-point arithmetic

Between versions 1.47 and 1.48, the `BigFloat` multiplication routine changed, primarily to reduce its use of temporary values. (We did not verify the behavior-preservation of the other changes to this version.) Though much of the code in the method was replaced, our tool was able to verify that these changes were behavior-preserving with respect to how the module was used by `Currency`. Most of the needed properties were verified without human intervention. We did have to slightly modify our testing script, however, to work around a deficiency in the Daikon invariant detector's handling of certain Perl global variables. We modified the tests (not `BigFloat` itself) to pass the default precision as an argument rather than requiring a runtime symbol table lookup to determine the class in which the precision should be looked up. We also added four predicates, which are hints that help Daikon to detect conditional program properties. Conditional properties are needed to capture properties of the method's behavior that are true only for a subset of its possible inputs: for instance, the multiplication routine performs differently depending on whether it receives an explicit argument specifying rounding, and whether or not one of its arguments in zero.

## 7.2 Date case study

For our second case study, the component is `Date::Simple`, a module for calendar calculations, and the application is `Date::Range::Birth`, which computes the range of birthdays of people whose age would fall into a given range on a given date. We compared three versions of `Simple` — 1.03, 2.01, and 2.03 — and the most recent available version (0.02) of `Birth`. `Simple` 1.03 consists of approximately 140 lines of code, while versions 2.01 and 2.03 consist of about 280 lines. In `Simple`, we examined all of the methods used by `Birth` (the constructor and three accessor methods), creating our own randomized tests. For our application, we supplemented the small set of tests supplied with the `Birth` module by adding four tests of invalid inputs and five test cases using a larger variety of correct inputs. The generated abstractions include uses of `Simple` both directly by `Birth` and calls via a third module `Date::Range`.

Comparing the three versions in the context of this application, our tool concludes that an upgrade or a downgrade between versions 2.01 and 2.03 would be safe. The tool warns that moving from 1.03 to a release with major version number 2, or vice versa, is potentially unsafe. An upgrade is judged unsafe because of a change in the return value of the constructor: in version 1.03, the constructor never returns a time represented as a negative value, while in versions 2.01 and 2.03 it does. Similarly, a downgrade is judged unsafe because in versions 2.01 and 2.03 the constructor never returned a time represented as zero, but it does in version 1.03. These behavior changes correspond to a bug in version 1.03: it uses an interface to the C `mktime` function to convert times into an internal representation of seconds since the Unix epoch, but the behavior of `mktime` for years before 1970 is incompletely specified. On some platforms, including the one we used for this experiment, `mktime` fails on dates before 1970, which in Perl is signified by returning a special null value `undef`. However, `Simple` fails to check for this error condition, and instead the value is treated as a zero, causing all dates prior to 1970 to be represented as December 31, 1969 (or January 1, 1970, in time zones east of UTC).

In this case study, our random tests of the Date-Simple component expose its buggy treatment of pre-1970 dates. Most likely, the `Simple` author was either unaware of this problem (perhaps because it did not occur on his platform) or would have considered use of the component for pre-1970 dates invalid.

# Chapter 8

# C library case studies

In order to test our techniques on larger examples, we performed case studies of upgrading a major software component, the Linux C library, as used by complete applications. On Unix systems a single library, traditionally named `libc`, provides the library functions required by the C standard, wrappers around the low-level system calls, and miscellaneous utility functions. Most Linux systems use version 2 of the GNU C library [Fre03], which provides a large shared library that is dynamically linked with virtually every system executable.

The authors of the GNU C library attempt to maintain compatibility, especially backward compatibility, between releases. Each procedure or global variable in the library is marked with the earliest library version it is compatible with, the library contains multiple versions of some procedures, and the static and dynamic linkers enforce that appropriate versions are used. These mechanisms assist with maintaining compatibility and avoiding incompatibility, but they are insufficient. We subverted this declarative check, and added a small number of stubs to our instrumentation library to simulate functions missing from older versions. Our experiments demonstrate that libraries marked as incompatible can be used without error by most applications, but also that some differences between procedures marked with the same version can cause errors.

Our experiments use unmodified binary versions of applications and the library. We capture an application's use of the library via dynamic interposition: a stub library wraps each function call, and records the arguments to and results from each invocation. The stubs are based only on the published library interface, as captured in header files and documentation: their creation does not require access to the library's source code. Our implementation does not support procedures taking or returning function pointers, or those with variable-length argument lists, but in total we provide stubs for approximately a thousand procedures in the C library.

## 8.1 A compatible C library upgrade

The Linux C library implements a stable API and attempts to maintain compatibility between versions. To see how well our technique validates large, but relatively safe upgrades, we compared versions 2.1.3 and 2.3.2 of the C library, as they were used by 48 applications and utility programs from version 7.3 of the standard Red Hat Linux distribution. (Normally, these applications would be used with the C library supplied with the system, based on version 2.2.5. The results from comparisons between 2.1.3 and 2.2.5, or between 2.2.5 and 2.3.2, were similar.)

We chose a suite of 48 commonplace applications, including many of the applications that the authors use in everyday work. These include a number of large graphical applications, such as text editors and a web browser, games, interface accessories, text-based application programs, and utility programs. When programs use other libraries, the uses of the C library by those other libraries are included as uses by the application. Application usage is represented by 20 minutes of scripted and recorded human usage, which exercises the programs in a fashion typical of daily use. The subject programs performed correctly, in all visible respects, with both library versions.

Because the (largely volunteer) authors of the GNU C library have provided only a limited formal test

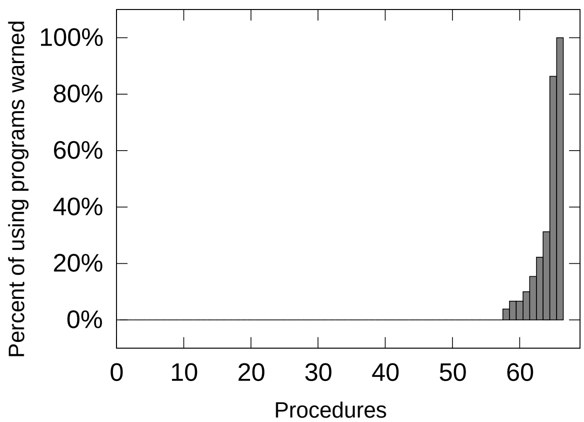


Figure 8.1: False positive incompatibility results for comparison between C library versions 2.1.3 and 2.3.2. Of 66 procedures whose behavior did not change, our tool reports no difference for 57. For each procedure, the graph shows the percentage of the programs that used that procedure for which a behavioral difference warning was reported.

suite, the role of the "test suite" in our case studies is instead played by the other applications, as described in Section 5.3 on page 26. The subject programs called 199 instrumented library procedures. Because our evaluation technique requires procedures to be tested by several clients, we restricted our attention to the 76 procedures that were used by 4 or more of the subject programs.

For the 76 procedures, our tool correctly warns of behavior differences in 10 of them and correctly approves 57 upgrades as having unchanged behavior. For 9 procedures, the tool warns (incorrectly, we believe) that the behavior differs for at least one application.

Our comparison technique discovers 10 genuine behavioral differences between the library versions; for the application programs that we examined, these differences appear to be innocuous. For example, the `dirent` structure returned by `readdir` holds information about an entry in a directory and contains a field named `d_type`. In version 2.1.3, this field was always zero, while in version 2.3.2 it took on a variety of values between 0 and 12. Though this change would theoretically be incompatible for an application that operated correctly only when the field value was zero, the value zero (also known as `DT_UNKNOWN`) is specifically documented to mean that no information about the file type is available, so well-written applications are likely to handle the change gracefully. Our tool also reports a number of behavioral differences arising from the fact that it examines the members of the `FILE` structure used by standard IO routines such as `fopen` and `fclose`. Because the definition of this structure is visible to user-written code, examining its members is conservative, but the differences it finds are not relevant to programs that correctly treat the structure as opaque.

The 9 false positive incompatibility warnings are summarized in Figure 8.1. The two tallest bars correspond to `tcgetattr` and `select`. The `tcgetattr` warnings arise because when that procedure is applied to a file descriptor that is not a terminal, it copies over a returned structure from uninitialized memory, causing spurious properties to be detected over these values. Two expected properties fail to hold for `select`: one bounding a return value indicating the number of microseconds left to wait when the procedure returns, and one concerning a field that our tool treats as an integer, though in fact it is part of a bit vector in which some bits are meaningless. On average, a user of our tool checking this C library upgrade for one of these applications would need to examine 2.69 failing procedures; of these reports, 0.75 would be spurious, and the remaining 1.94 would represent real differences, which upon examination do not affect the application in question. The distribution of numbers of procedures flagged for different programs is shown in Figure 8.2. As would be expected, larger applications have more potential for incompatibility: the two programs with the most warnings were Netscape Communicator and GNU Emacs.

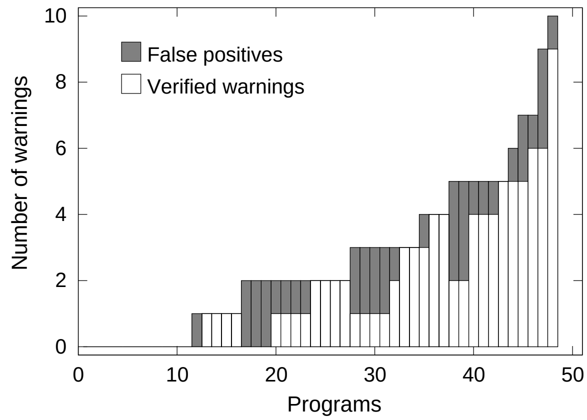


Figure 8.2: Reported incompatibilities between C library versions 2.1.3 and 2.3.2 for 48 subject programs. For each program, the graph shows how many procedures our technique warns about as potentially containing incompatibilities. White bars show incompatibilities that we have verified by hand. Shaded bars show warnings that are probably false positives.

## 8.2 C library incompatibilities

We also used our technique to examine two incompatible changes made by the authors of the GNU C library. Coincidentally, both relate to procedures that operate on representations of time; of course, our technique is not limited to such procedures. These procedures were not considered in the experiment described in Section 8.1 because they were used by too few of those programs, though one of the differences exists between the versions considered there.

### 8.2.1 The `mktime` procedure

The `mktime` procedure is specified in the C standard to convert date and time values specified with separate components (year through seconds) into a single value of type `time_t`, which in Unix is traditionally a 32-bit signed integer representing seconds since the 'epoch' of midnight UTC, January 1st, 1970. If the time cannot be so represented, a value of $-1$ is returned. Before April 2002, the GNU `mktime` converted dates between 1901 and 1970 into negative `time_t` values. In April of 2002, the C library maintainers concluded that this behavior was in conflict with the Single Unix Specification [Ope03] (the successor to POSIX), and changed `mktime` to instead return $-1$ for any time before the epoch. (Though this change was not incorporated into version 2.2.5 of the library as released by the GNU maintainers, it was adopted by Red Hat in the version of the library distributed with Red Hat 7.3, which is also labeled as version 2.2.5. This incompatibility, between two versions with the same label, underscores the dangers of relying on developers to label incompatibilities by hand.)

To see how our technique observed this change, we compared the behavior of the `mktime` function in the Red Hat-supplied version of the C library, and in a freshly compiled version of 2.2.5 as released by its maintainers. Our subject programs were `date`, `emacs`, `gawk`, `pax`, `pdfinfo`, `tar`, `touch`, and `wget`; for each program, the library was considered to be tested by the remaining programs, as described in Section 5.3.

Our tool reports that this upgrade to `mktime` would not be safe for any of the programs we examined. Though the correct behavior of `mktime` is too complex to be described in the grammar of our operational abstraction generation tool, our technique does discover differences between the old and new behaviors of `mktime` caused by the change described above. Specifically, when `mktime` completes successfully, it modifies several fields of the supplied time structure: for instance, it sets the day of the week and the offset from UTC of the timezone in effect, based on other supplied components of the date and time. Because most of the programs we tested only supplied dates within the domain of the older `mktime` version, when they were used with that version they expected the fields to be filled in with meaningful values: for instance, the operational abstraction for `touch` included an expectation that the timezone offset (in seconds) would be either $-18000$, $-14400$, or 0 (corresponding to Eastern Standard Time, Eastern Daylight Time, or UTC respectively). When the new version of `mktime` is called with a date in the year 1968 or earlier, however, it returns $-1$ immediately, without modifying these auxiliary fields, and they remain uninitialized.

Because of this phenomenon, our tool reports that a number of properties involving these fields will not hold using the upgraded version of `mktime`. In the applications we tested, the change to `mktime`'s functionality does cause user-visible functionality to be reduced. For instance, `date` with the new library refuses to operate on dates between 1901 and 1970 which would be accepted when running with the old library. This error has the same original cause as the one discovered in the Perl module case study of Section 7.2 on page 35 (though that problem was compounded by a failure to check for errors). However, this manifestation is completely different, and its effects were discovered in a different way and in different programs.

### 8.2.2 The `utimes` procedure

The C library's `utimes` procedure updates the last-modification and last-access timestamps on a file. The interface of `utimes` allows these times to specified by a combination of a number of seconds (measured since the 1970 'epoch') and microseconds. Our version of the Linux kernel stores file timestamps with one-second granularity, so the C library must convert the times to a whole number of seconds before passing them to it. During the summer of 2003, the conversion method was changed from truncation to rounding. This change was incompatible with other Unix programs: for instance, rounding up caused the `touch` command to give files a timestamp in the future, which in turn caused `make` to exit with an error message. After wide distribution of this library, including in the Debian Linux development distribution, and a month and a half of discussion, the change was reverted in response to user complaints.

Our technique recognized this change. We compared the behavior of the system version of the C library on our Red Hat 7.3 workstation (Red Hat version 2.2.5-43) with that of a version from the development CVS repository as of September 1st, 2003. Our subject programs were the standard utilities `cp`, `emacs`, `mail`, `pax`, and `touch`; for `cp`, `mail`, and `touch`, we used more recent versions (from the Debian development distribution). We wrote a short script to exercise each program's use of `utimes`; for each program, we used the other four as the test suite.

Our tool reports that an upgrade to the C library version with the round-to-nearest behavior would be unsafe for all five of the applications we considered. For each application, it reports that the new library fails to guarantee a property that the old one did, namely that the last-access timestamp of the affected file in seconds, after the call to `utimes`, should equal the seconds part of the new access timestamp passed to `utimes`. For three of the programs, it also separately warns about a similar fact regarding the last-modification timestamp, because for those programs the two timestamps passed `utimes` were not always equal. Note that the timestamps of the file are not arguments to `utimes`; they are found using the `stat` procedure as a virtual field of the filename, as discussed in Section 5.1.1 on page 24. Our tool finds no other cross-version behavior differences between the two versions of `utimes`.

For both the old and new versions, our tool also issues a handful of warnings indicating that the other applications we used were not a complete test suite for a given application's use of `utimes`; for instance, the files updated by `mail` were on a different filesystem that those updated by the other programs, and only `touch` supplied modification times with non-zero microsecond values. As these warnings occur identically for both versions, they are filtered out using the technique of Section 5.2.2 (page 25).

## 8.3 Effects of abstraction grammar

The results produced by our technique depend on the grammar of the operational abstractions used to characterize program behavior. If the grammar is too small, the technique may miss properties that are important for program correctness. However, if the abstraction grammar is extended arbitrarily, the technique's results will not necessarily improve. Specifically, our tool may produce more false positives if the grammar contains properties describing special cases, but not corresponding properties for more general

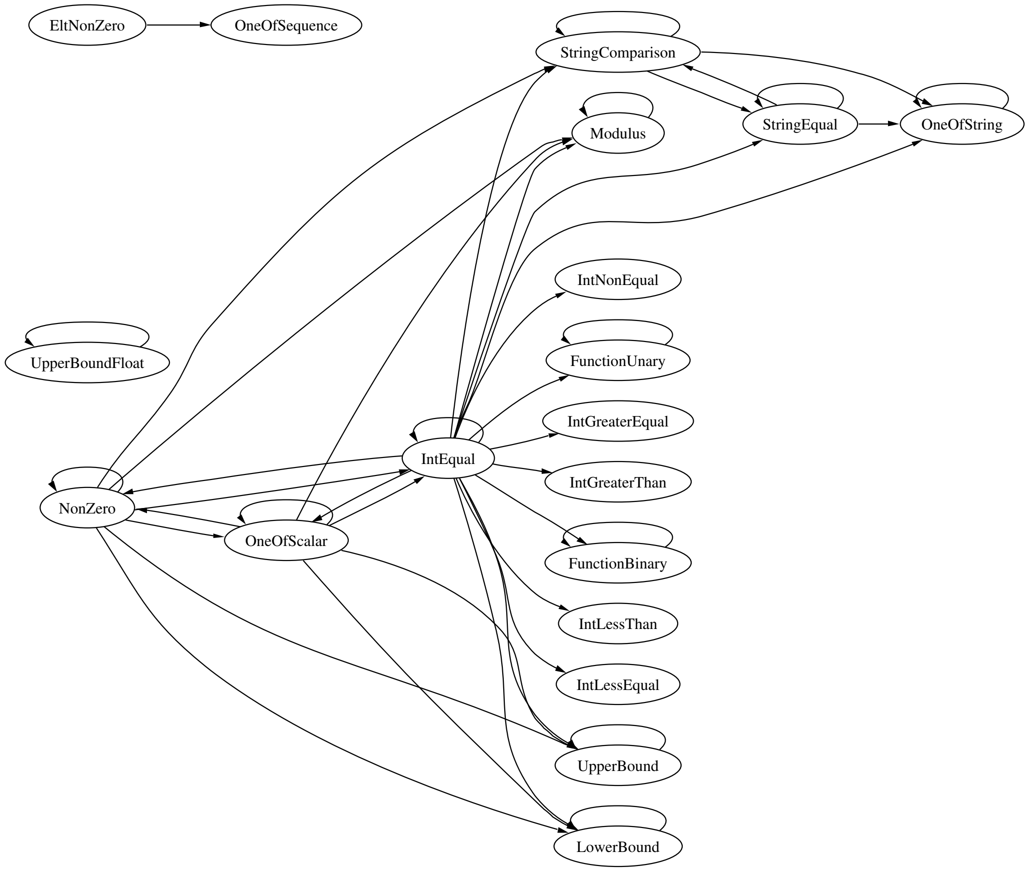


Figure 8.3: Dependencies between property types for verifying an upgrade of the C library as used by GNU Emacs. Nodes represent types of property (invariant classes in the Daikon tool). An edge connects two nodes if at least one property of the type pointed to could only be proved as a consequence of a property of the type where the arrow originates (perhaps along with other properties).

phenomena: the application's abstraction may represent some expected behavior that cannot be proved to occur based on the tested abstraction.

A simple example of the effect of increasing the abstraction grammar can be seen in the sorting example of Section 2.2 (page 6). The operational abstractions shown in Figures 2.5 and 2.6 omit, for brevity, a number of properties that Daikon finds involving subsequences of the array $a$. When these properties are added to the grammar used for both the application and component abstractions, the same consistency comparison succeeds, because while it is considering a richer model of the component's behavior, this richness applies to both the statements that must be proved and the assumptions that can be used to prove them.

To explore in more detail the effect of the abstraction grammar on which properties can be proved to hold, we repeated an experiment from our Linux C library case study, but using only subsets of the properties that the Daikon tool normally detects. For efficiency, we performed this experiment by checking separately, for each property that was proved to hold of the library for its use by a particular program (GNU Emacs), which minimal subsets of property types sufficed to verify the property. (Properties that occurred identically in the assumed abstraction and the abstraction being verified do not appear in the results, as our tool eliminates them immediately to save processing.) A property type appeared in every minimal subset if and only if without it, the target property could not be proved. The results of this experiment are shown graphically in Figure 8.3.

An edge in Figure 8.3 represents that if properties of the type indicated by the pointed to node were removed from Daikon's grammar, at least one property of the type from which the edge originates would become impossible to prove. For instance, eliminating all of the properties pertaining to strings (String-Comparison, StringEqual, and OneOfString) would not increase the number of false positive results (there are no outgoing edges from the String nodes to the rest of the graph), but removing some string properties while keeping others could mean that some true properties could no longer be proved. We can see that there is considerable freedom in the choice of an abstraction grammar for our technique to work, but that some amount of consistency (a sort of closure under consequence) is required.

# Chapter 9

# Related work

Our technique builds on previous work that formalized the notion of component compatibility, and complements other techniques that attempt to verify the correctness of multi-component systems. Our work differs in that it characterizes a system based on its observed behavior, rather than a user-written specification, and is applicable in more situations.

## 9.1 Subtyping and behavioral subtyping

Strongly typed object-oriented programming languages, such as Java, use subtyping to indicate when component replacement is permitted [SCB$^+$86, BHJ$^+$87, Car88]. If type-checking succeeds and a variable has declared type $T$, then it is permissible to supply a run-time value of any type $T'$ such that $T' \sqsubseteq T$: that is, $T'$ is either $T$ or a subtype of $T$. However, type-checking is insufficient, because an incorrect result can still have the correct type.

One approach to verifying the preservation of semantic properties across an upgrade is for the programmer to express those properties in a formal specification. This is the principle of behavioral subtyping [AvdL90, LW94]: type $T'$ is a behavioral subtype of type $T$ if for every property $\phi(t)$ provable about objects $t$ of type $T$, $\phi(t')$ is provable about objects $t'$ of type $T'$. Recently, tools have become available to enable writing and checking such properties [Sta97, LN98, LBR99, FLL$^+$02].

In practice, the requirement of behavioral subtyping is both too strong and too weak for use in validating a software upgrade. Like any condition that pertains only to a component and not the way it is used, the requirement is too strong for applications that use only a subset of the component's functionality. If a system only uses half of the APIs provided by a component, then the system remains correct even if the vendor makes incompatible changes in the behavior of the unused APIs. This inadequacy of the behavioral subtyping rule implies that the decision about whether to upgrade must be made independently for each application, based on its own use of the component. Our approach differs from behavioral subtyping in that it accounts for distinct uses of the component. Formal specifications are also too weak because a system may inadvertently depend on a fact about the implementation of a component version that is omitted (perhaps intentionally) from the specification. For example, suppose that a component's interface includes an iterator that is specified to return the elements of a collection, and that the old component happens to return the elements in order. It would be easy for the system to inadvertently depend on this property. The new version of the component may feature performance improvements — for instance, it might store the collection in a hash table, causing the iterator to return elements in an arbitrary order. The system as a whole would malfunction when using the new component. (This weakness is distinct from the limitation that any system of automatic verification has properties about which it cannot reason, which affects both our technique and ones based on the verification of human-written specifications.)

## 9.2 Specification matching

Zaremski and Wing generalize behavioral subtyping to consider several varieties of matching between specifications [ZW97]. Such comparisons can be used for a number of purposes in which the question to be answered is, broadly, whether one component can be substituted for another. Most previous research, however, has focused on retrieving components from a database, to facilitate reuse [PA97, SF97].

Though they considered a large number of possible comparison formulas, Zaremski and Wing omitted the one that we adopted for our single component upgrades. Formulas equivalent to the single-component formula have been used for reuse (sometimes called the "satisfies" match [PA97]) and in work building on behavioral subtyping [DL96]. Also, in the VDM tradition [Jon90], proof obligations analogous to the condition (with the addition of a function mapping concrete instances to abstract ones) and called the "domain rule" and the "result rule" are used to demonstrate that a concrete specification correctly implements an abstract specification. To our knowledge, no previous work considers all the issues raised by the multi-module model introduced in this paper, or uses the same formula that it does.

Ours is also not the first attempt to automate the comparison of specifications with theorem proving technology. Zaremski and Wing use a proof assistant in manually verifying a few specification comparisons [ZW97]. Schumann and Fischer use an automated theorem prover with some specialized preprocessing [SF97]. By comparison, the operational abstractions we automatically verify are significantly larger than the hand-written specifications used in previous work, though the individual statements in our abstractions are mostly simple.

## 9.3 Other component-based techniques

The increased use of black-box components in systems construction increases the need for automatically checkable representations of component behavior. Technically, our approach is most closely related to techniques based on behavioral subtyping; their use in the component-based context is well summarized by Leavens and Dhara [LD00]. A more common approach, however, has been to abstract component behavior with finite state representations such as regular languages [Nie93] or labeled transition systems [MRR03]. Like our operational abstractions, such representations can be automatically checked to determine if one component can be substituted for another. The kinds of failure found by the different techniques are complementary, though. Finite-state techniques excel at checking properties that are simple, but global; for instance that a file must always be opened before being read. Our operational abstractions can capture a richer set of properties, including infinite state ones, but only as they are localized to the pre- or postconditions on a particular interface.

## 9.4 Avoiding specifications

Ideally, a technique like the one we describe could be used with hand-written specifications in the place of operational abstractions. However, not only would the component specification need to be proved to describe the component's actual behavior, the application would have to correctly specify the particular component behaviors it relied on for its correctness. Creating and proving such comprehensive specifications would likely be too difficult and time-consuming for most software projects.

In the absence of specifications, one might also attempt to statically verify that two versions of a component produce the same output for any input. However, such checking is generally only possible when the versions are related by simple code transformations [YHR92]. For instance, techniques based on symbolic evaluation can verify the correctness of changes made by an optimizing compiler, such as common subexpression elimination [Nec00]. If a program change was subtle enough to require human expertise in its application, though, it is usually too subtle to be proved sound automatically.

## 9.5 Performing upgrades

It is sometimes possible to substitute incompatible components by wrapping them in code that translates procedure names, converts data (for instance, via an intermediate abstract representation [HL82, HWP92]), or fixes bugs. (Even compatible components may require updates to data structures or other parts of the system.) Old and new component versions can also be used simultaneously, with a specification directing when to use which: [CD99] for instance, one might wish to use a new component in circumstances where the old component has a known bug, but the old component everywhere else. Our work notifies humans of the need to cope with such incompatibilities.

Many researchers have investigated how to perform upgrades in a running system. One approach is to quiesce or "passivate" the system, in order to emulate halting and restarting it [HMN01, BISZ98, OMT98]; another is to run multiple versions of a component simultaneously (Segal [SF93] surveys techniques). Distributed systems offer special challenges [Blo83, BD92]; for instance, simultaneous upgrades are impossible. Our work addresses the complementary, and less investigated, problem of when the upgrade is permissible.

# Chapter 10

# Conclusion

We have presented a new technique to assess whether replacing a component of a software system by a purportedly compatible component may change the behavior of the system. This is necessary because component authors cannot foresee (nor test for) all uses to which their components may be put. The key idea is to compare the run-time behavior of the old component in the context of the system with the run-time behavior of the new component in the context of its own test suite. The components may be exchanged if the new one's tested run-time behavior is logically strong enough to guarantee all of the behavior of the old one as it was observed at run-time in the system. The behaviors are captured and compared as operational abstractions, which are formal mathematical descriptions that generalize over observed program executions. Strength is defined with respect to a model of behavior in a multi-module system: the technique handles multiple simultaneous upgrades, shared state, callbacks, and indirect communication through the system via a uniform framework.

This technique has a number of positive attributes, making it complementary to other approaches for determining the suitability of an upgrade. The technique is application-specific: it can indicate that an upgrade is safe for one client but unsafe for a different client. The technique does not require integrating the new component into the system or running system tests, permitting earlier and cheaper detection of incompatibilities. (In fact, an integrator could even perform the comparison before deciding whether to purchase the new component.) The technique makes minimal demands on the component vendor and the system integrator; manual intervention is required only to investigate a reported incompatibility. The technique does not require developers or users to write or prove formal specifications. The technique is blame-neutral: it warns of incompatibilities regardless of whether the component vendor or the application developer is at fault, or even if blame is impossible to assign unambiguously. The technique does not depend purely on input-output behavior nor on an oracle indicating correct behavior: where appropriate, it can also take advantage of other interfaces or of internal behavior. However, the technique works even without any access to the component source code.

We describe practical extensions of the technique to situations that arise in real-world code: non-local state, apparent non-determinism, pre-existing innocuous incompatibilities, and missing test suites. We have implemented all these enhancements, enabling us to perform a case study of upgrading the Linux C library in 48 Unix programs. Our tool approved upgrades of most parts of the library, indicated genuine behavioral differences, and had a low false positive rate. Furthermore, it also identified several differences that led to user-visible errors. Thus, we believe that this technique represents a promising approach to facilitate the construction and maintenance of large software systems, which can be adopted by developers with only minor changes to their existing practices.

# Bibliography


[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications and 4th European Conference on Object-Oriented Programming (OOPSLA/ECOOP '90)*, pages 161–168, Ottawa, Canada, October 21–25, 1990.

[BD92] Toby Bloom and Mark Day. Reconfiguration in Argus. In *International Workshop on Configurable Distributed Systems*, pages 176–187, London, England, March 1992.

[BHJ+87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distributed and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[BISZ98] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *4th International Conference on Configurable Distributed Systems*, pages 35–42, Annapolis, MD, May 1998.

[Blo83] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983. Also as Technical Report MIT/LCS/TR-303.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[CC00] Yonghao Chen and Betty H. C. Cheng. A semantic foundation for specification matching. In *Foundations of Component-Based Systems*, chapter 5, pages 91–109. Cambridge University Press, New York, NY, 2000.

[CD99] Jonathan E. Cook and Jeffery A. Dage. Highly reliable upgrading of components. In *International Conference on Software Engineering*, Los Angeles, CA, 1999.

[CRS96] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, pages 62–70, 1996.

[Cur94] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer Technical Conference*, pages 267–278, 1994.

[Dev99] Premkumar Devanbu. A reuse nightmare: Honey, I got the wrong DLL. In *ACM Symposium on Software Reusability (ACM SSR'99)*, Los Angeles, CA, USA, 1999. Panel position statement.

[DF93] J. Dick and A. Faivre. Automating the generating and sequencing of test cases from model-based specifications. In *FME '93: Industrial Strength Formal Methods, 5th International Symposium of Formal Methods Europe*, pages 268–284, 1993.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267, Berlin, Germany, 1996.

[DNS03] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, July 23, 2003.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[Ern94] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, July 26, 1994.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.

[Fre03] Free Software Foundation. GNU C library, 2003. `http://www.gnu.org/software/libc/libc.html`.

[FS01] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205, London, UK, January 17–19, 2001.

[HL82] Maurice P. Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, 1982.

[HMN01] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 13–23, Snowbird, Utah, 2001.

[HWP92] C. Hofmeister, E. White, and J. Purtilo. Surgeon: A packager for dynamically reconfigurable distributed applications. In *International Workshop on Configurable Distributed Systems*, pages 164–175, London, England, March 1992.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall, second edition, 1990.

[KEGN01] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[LD00] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, New York, NY, 2000.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In *Compiler Construction: 7th International Conference, CC'98*, pages 302–305, Lisbon, Portugal, April 1998.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Meu98] Christophe Meudec. *Automatic Generation of Software Test Cases From Formal Specifications*. PhD thesis, Queen's University of Belfast, 1998.

[Mor02] Benjamin Morse. A C/C++ front end for the Daikon dynamic invariant detection system. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2002.

[MRR03] Sabine Moisan, Annie Ressouche, and Jean-Paul Rigault. Behavioral substitutability in component frameworks: A formal approach. In *Proceedings of the 2003 Workshop of Specification and Verification of Component Based Systems*, Helsinki, Finland, 2003.

[NE02a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, Charleston, SC, November 20–22, 2002.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 83–94, Vancouver, BC, Canada, 2000.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–15, Washington, D.C., USA, 1993.

[NL98] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, 17–19 June 1998.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998.

[Ope03] The Open Group, editor. *The Single UNIX Specification, Version 3*. The Open Group, 2003. `http://www.unix.org/version3/`. Also known as IEEE Std 1003.1-2001 (POSIX.1) and ISO/IEC 9945:2003.

[PA97] John Penix and Perry Alexander. Toward automated component adaptation. In *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering (SEKE-97), Madrid, Spain, June 18-20, 1997.*, 1997.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, CA, 1995.

[ROT89] Debra J. Richardson, Owen O'Malley, and Cindy Tittle. Approaches to specification-based testing. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*, pages 86–96, December 1989.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, Portland, OR, USA, June 1986.

[SF93] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2), March 1993.

[SF97] Johann Schumann and Bernd Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In *Proceedings of the 12th Annual International Conference on Automated Software Engineering (ASE'97)*, pages 246–254, Lake Tahoe, California, 1997.

[Sta97] Raymie Stata. Modularity in the presence of subclassing. Technical Report MIT-LCS-TR-711, MIT Laboratory for Computer Science, Cambridge, MA, April 1, 1997. Revision of PhD thesis.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[Wei02] Robert K. Weiler. Automatic upgrades: A hands-on process. *Information Week*, March 2002.

[YHR92] Wuu Yang, Susan Horwitz, and Thomas Reps. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology*, 1(3):310–354, 1992.

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.