Predicting Problems Caused by Component Upgrades

Stephen McCamant Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab
200 Technology Square
Cambridge, MA 02139 USA

smcc@lcs.mit.edu, mernst@lcs.mit.edu

ABSTRACT

We present 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 generates an operational abstraction 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. 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. In case studies, the technique identified several incompatibilities among software components.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and post-conditions, Mechanical verification*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Languages, Reliability, Verification

Keywords

Software upgrades, specification matching, software components

This research was supported in part by NSF grants CCR-0133580 and CCR-0234651 and by a gift from NTT.

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

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 [36, 9]. 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 and prevent 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).

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, modulo irrelevant properties of the execution environment. 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 [13, 14], 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 formulae over program variables — but an operational abstraction describes actual program behavior and

```
// 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 1: A sorting method that uses `swap`.

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.

Our technique is as follows. 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".) 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, after accounting for properties of the execution context that are maintained by the old component. 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 proposed 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 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 paper is organized as follows. Section 2 illustrates our technique by means of a simple example. Section 3 details our approach to detecting component incompatibilities. Section 4 describes case studies that suggests that the technique may be useful in practice, and Section 5 provides further perspective on the results. Section 6 surveys related work, and Section 7 concludes.

2. SORTING EXAMPLE

This section gives a simple example to illustrate our approach.

2.1 The problem

Consider the sorting routine of Figure 1 as an application, and the `swap` subroutine of Figure 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 order. Now suppose that the vendor releases version 2 of `swap`, as shown in Figure 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

```
// 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: Version 1 of a method that swaps two array elements.

```
// 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 3: Version 2 of a method that swaps two array elements.

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 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 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 6.3). Figure 5 shows the abstraction generated for `swap` as used by the bubble sort. Figures 5, 6, and 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 = null or for $i = j$, the vendor did not test the component for such values. The vendor's abstraction (Figure 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 desires 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 3.1 explains this test in detail.)

```
// 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 4: Another sorting method that uses `swap`.

Preconditions for `swap`	Postconditions for `swap`
$a \neq$ 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 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`
$a \neq$ 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 6: The operational abstraction for `swap` (version 2), in the context of the vendor's test suite.

Preconditions for `swap`	Postconditions for `swap`
$a \neq$ 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 7: The operational abstraction for `swap` (version 1), in the context of `selection_sort`.

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 5) imply those that the new component requires (Figure 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, 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`.

2.3 Upgrading selection sort

The author of the selection sort application can apply the same process. Figure 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 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`.

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.

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 6.3 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, the system 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, requiring 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. This step results in 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 [32, 10, 7, 26], 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 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. 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 problem 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.

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_{pre} and postconditions A_{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

$$A_{\mathrm{pre}} \Rightarrow T_{\mathrm{pre}} \qquad \text{and} \qquad (A_{\mathrm{pre}} \wedge T_{\mathrm{post}}) \Rightarrow A_{\mathrm{post}} \ . \qquad (1)$$

3.1.1 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 $A_{\mathrm{pre}} \Rightarrow A_{\mathrm{post}}$.

We must conclude $A_{\mathrm{pre}} \Rightarrow A_{\mathrm{post}}$ based on the knowledge that the test abstraction accurately describes the behavior of the new component, i.e., that $T_{\mathrm{pre}} \Rightarrow T_{\mathrm{post}}$. 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 $A_{\mathrm{pre}} \Rightarrow T_{\mathrm{pre}}$. Therefore, we are looking for a condition guaranteeing exactly that

$$((T_{\mathrm{pre}} \Rightarrow T_{\mathrm{post}}) \Rightarrow (A_{\mathrm{pre}} \Rightarrow A_{\mathrm{post}})) \ \wedge \ (A_{\mathrm{pre}} \Rightarrow T_{\mathrm{pre}}) \ .$$

This formula is equivalent to (1).

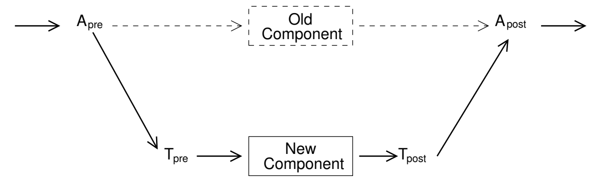

Figure 8: The behavioral subtyping rule. A new component whose specification guarantees $\mathbf{T}_{\mathrm{pre}} \Rightarrow \mathbf{T}_{\mathrm{post}}$ may replace an old component whose specification guaranteed $\mathbf{A}_{\mathrm{pre}} \Rightarrow \mathbf{A}_{\mathrm{post}}$ (dashed), if $\mathbf{A}_{\mathrm{pre}} \Rightarrow \mathbf{T}_{\mathrm{pre}}$ and $\mathbf{T}_{\mathrm{post}} \Rightarrow \mathbf{A}_{\mathrm{post}}$ (solid). Arrows represent both program control flow and logical implication between specifications. This rule is sufficient but too strong for validating component upgrades. Section 3.1 presents our alternate rule.

3.1.2 Alternative conditions

Several conditions similar to ours have been suggested for other applications [38]. 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 6.1):

$$A_{\mathrm{pre}} \Rightarrow T_{\mathrm{pre}} \qquad \text{and} \qquad T_{\mathrm{post}} \Rightarrow A_{\mathrm{post}} \ . \qquad (2)$$

As schematically illustrated in Figure 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_{pre} holds, so by the first implication, T_{pre} holds. According to the component's tested behavior, $T_{\mathrm{pre}} \Rightarrow T_{\mathrm{post}}$, so T_{post} holds. Then, by the second implication, A_{post} holds, so the application's behavior remains the same. A_{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 (1) and (2) differ when T_{post} holds, but neither A_{pre} nor A_{post} does. The possibility of such a situation would prohibit an upgrade according to (2), 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_{pre} to the second implication in our condition (1). For example, 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_{pre} and T_{post} might be x *is an integer* and $x' = x + 1$, while A_{pre} and A_{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.

As seen in Section 3.1.1, our condition can also be understood as a strengthening of a weak upgrade condition,

$$(T_{\mathrm{pre}} \Rightarrow T_{\mathrm{post}}) \Rightarrow (A_{\mathrm{pre}} \Rightarrow A_{\mathrm{post}}) \ . \qquad (3)$$

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 (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_{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_{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_{pre} and T_{post} are $x > 0$ and $x' = x$ respectively). While the weaker condition would allow this upgrade, our condition (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 [38]; their terminology might call it "application-guarded plug-in match."

3.2 Implementation of comparison

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.

Our tool uses the Simplify automatic theorem prover [28, 8] to evaluate the logical comparison formula described in Section 3.1. 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 30 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. As shown in Figure 9, checking each property takes only a fraction of a second.

3.3 Meta-comparison

Our technique may fail to prove equation 1 of Section 3.1 even when an upgrade is behavior-preserving. Such failures might be the result of insufficient testing, a theorem prover weakness, or an inadequate grammar of the operational abstraction (for further discussion, see Section 5.1). We propose *meta-comparison* to work around such problems. If our technique does not approve an upgrade from an old component to a new component, then it considers an upgrade from the old component to itself. Such an upgrade is always behavior-preserving. However, the theorem prover might not be powerful enough to verify some condition, or a property that is required to prove the condition might not appear in the operational abstraction (because of limitations of the test suite or of the invariant detector). Any theorem-proving failure that occurs only with the new component certainly represents a behavioral difference. If the theorem-proving failures for the self-upgrade are identical to the theorem-proving 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 did not use meta-comparison in any of the case studies reported in Section 4.

4. CASE STUDIES

We assessed the effectiveness of our approach by case studies involving pairs of Perl modules.

4.1 Currency case study

The first case study concerns code for manipulating monetary quantities.

4.1.1 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 aren't caught, 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

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 9: 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.

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 installed version of a component.

4.1.2 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 varieties of input, 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 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.

4.1.3 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 (these are discussed further in Section 5.1.1). 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.

4.2 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.

5. 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.

5.1 Limits of abstraction comparison

Though the case studies described in section 4 are relatively small, we believe that performance problems do not represent the main threats to the scalability of the technique. Dynamic invariant detection can construct operational abstractions efficiently [14], and we have found empirically that abstractions can also be efficiently compared. Instead, the key challenge for our technique is to avoid spuriously rejecting upgrades given the variety of behavior found in real programs.

5.1.1 Finding and proving implications

When a tested component has several discrete modes of operation, of which an application uses only a subset, the tested abstraction often needs a conditional property to express this special case. For instance, if the tested component computes the absolute value of an input, but the application only gives negative values for that input, the application will have the properties $x < 0$ (in the prestate) and $x' = -x$ (in the poststate). To prove the latter, the tested component abstraction needs to contain a conditional property such as $(x \leq 0) \Rightarrow (x' = -x)$.

The Daikon invariant detector can find conditional properties, but it requires hints in the form of predicates specifying how to subdivide its input into classes in which conditional facts might hold. (The left-hand side of implications discovered from such subsets is either the supplied predicate, or another property that was discovered to hold on the same subset.) Daikon has heuristics to generate some predicates, and others can be found by a static analysis of program source code, by clustering of program data points, and by several other techniques [12, 11]. For the bubble sort example of Section 2.1, statistical clustering found the relevant predicate (whether i was less than or greater than j in the swapping routine). Further work is also needed in pruning the set of conditional invariants produced, since their processing increases running time and the probability of false positives (properties that are false in general but are not falsified by a given test suite).

5.1.2 Testing is required

Because our technique characterizes a component in terms of the way it is tested (by its vendor) and used (by an application), the technique requires both tests and uses of the component.

The technique presumes that the vendor considers the behavior exhibited during testing to be correct; the technique is aimed at detecting behavioral differences that seem acceptable in isolated testing but are inappropriate in a particular context. Component testing is already performed in practice; our technique simply distills its results so that they can be verified by a component's users. Other research has demonstrated that the number of tests needed to produce accurate operational abstractions is comparable to that needed to achieve traditional goals such as statement or branch coverage, and that tests that generate accurate abstractions can be automatically minimized to reduce their running time [17].

The technique only advises about the safety of future component uses that are similar to past uses by the application, as captured by the operational abstraction. (Thus, the application operational abstraction should represent all the ways that the application typically exercises the component.) If a user runs the application in a new domain, then the application may invoke the component in ways that the application never used it before. In such a situation, there is no guarantee about how either the old or the new component would behave, so users are no worse off with the new component.

5.1.3 Limitations of the abstraction grammar

The effectiveness of our technique is limited by the grammar of properties that can be recognized by the abstraction generation tool. Any tool is oblivious to program properties outside its grammar. In practice, our tool often recognizes another property that is related to, or a side effect of, the particular behavior change that a programmer might cite as the bug. A complete description of the problematic behavior might be a complex logical expression, but often Daikon's grammar of mainly simple expressions includes one that separates correct from incorrect behavior. (This can be seen in the examples of Sections 4.1.2 and 4.2.)

A related danger is that the grammar captures a special case property in an application's behavior, but not a more general fact about the component's tested behavior that is needed to prove it. Intuitively, the abstraction tool's grammar needs to be closed under a kind of special-case intersection. For instance, we encountered such a problem when examining a variant of the example in Section 2.1, trying to prove that selection sort could be upgraded to a version of swap that was tested to behave correctly when swapping an element with itself. Previously, Daikon had considered array subsequences whose first index was at the start of the array, or whose final index was at the end. To completely capture the fact that only the two elements specified as arguments to swap are modified, we needed to enhance Daikon to also consider subsequences in the middle of an array.

We believe that if the abstraction grammar is "closed" in this sense of not missing properties related to ones it contains, it can be effective at both rejecting and approving upgrades. For instance, the example of Section 2.1 works just as well without any subsequence properties: our tool is unable to prove more complex properties of the program's behavior, but it does not have to because the properties to check are also correspondingly simpler. In practice, the technique can accurately predict whether an upgrade will be safe using only a finite subset of the possible true properties of a program.

6. RELATED WORK

Our technique builds on previous work that formalized the notion of component compatibility. Our work differs in that it characterizes a system based on its runtime behavior, rather than a user-written specification. Our research is complemented by other techniques which, once an upgrade is deemed safe, can be used to perform it with minimal disruption. This section also discusses how to generate an operational abstraction.

6.1 Static safety checks

Strongly typed object-oriented programming languages, such as Java, use subtyping to indicate when component replacement is permitted. 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. Such an execution is guaranteed not to result in a type error. The subtype relationship requires contravariance of argument types and covariance of result types [33, 3, 6]. 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 [1, 25]: 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 [35, 24, 23, 16].

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 for validating a software upgrade. Specifications abstract over the component's behavior, but a system may inadvertently depend on a fact about the implementation of a component version that is omitted in 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.)

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.

Zaremski and Wing's specification matching [38] gives a framework of relations that generalize behavioral subtyping, and they indicate uses for many of the resulting relations, one of which is substituting a component for another. Zaremski and Wing use a proof assistant in manually verifying a few specification comparisons; by contrast, we use a theorem prover to automatically compare more comprehensive operational abstractions. They considered only programmer-written specifications, and their framework does not include the test performed by our technique. Our comparison formula (Section 3.1) differs from theirs by "guarding", or restricting the behavior compared, to the subset of behaviors used by a particular application. Section 3.1.2 relates our test to the two most similar ones among those they define.

Formulas with the same structure as the one we suggest have been used in proving that a specification for a particular data representation correctly implements an abstract specification, which is the problem of data refinement or data reification. For instance in the VDM tradition [21], proof obligations analogous to our conditions (with the addition of a function mapping concrete instances to abstract ones) are called the "domain rule" and the "result rule".

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 [37]. For instance, techniques based on symbolic evaluation can verify the correctness of changes made by an optimizing compiler, such as common subexpression elimination [27]. If a program change was subtle enough to require human expertise in its application, though, it is probably too subtle to be proved sound automatically.

6.2 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 [18, 20]), or fixes bugs. (Even compatible components may require updates to data structures or other parts of the system.) 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 [19, 2, 31]; another is to run multiple versions of a component simultaneously (Segal [34] surveys techniques). Distributed systems offer special challenges [4, 5]; for instance, quiescing and simultaneous upgrade are impossible. Our work addresses the complementary, and less investigated, problem of when the upgrade is permissible.

6.3 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 [13, 14] 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.lcs.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 ($x \in \mathit{mytree}$, $\mathit{mylist\ is\ sorted}$), pointers ($\mathit{node} = \mathit{node.child.parent}$), and conditionals ($\mathit{if}\ p \neq \mathit{null\ then}\ p.\mathit{value} > x$). Daikon incorporates static analysis, statistical tests, logical inference, and other techniques to improve its output [15].

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 [13, 14] or can be proved correct [29, 30], and even lesser-quality output forms a partial specification that is nonetheless useful [22], because it is much better than nothing.

7. 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 stronger (according to a novel test we have introduced) than the old one's run-time behavior in the system. The behaviors are captured and compared as operational abstractions, which are formal mathematical descriptions that generalize over observed program executions.

This technique has a number of positive attributes that complement 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. The technique requires no manual effort by the component vendor, nor by the system integrator (except perhaps to investigate a reported incompatibility). Developers and users are not required 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 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.

Preliminary case studies suggest that the technique effectively identifies some incompatibilities in software components, permitting informed decisions about whether to perform an upgrade.

8. REFERENCES

[1] P. America and F. 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, Oct. 21–25, 1990.

[2] 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.

[3] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distributed and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, Jan. 1987.

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

[5] T. Bloom and M. Day. Reconfiguration in Argus. In *International Workshop on Configurable Distributed Systems*, pages 176–187, London, England, Mar. 1992.

[6] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, Feb./Mar. 1988.

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

[8] D. Detlefs, G. Nelson, and J. Saxe. Simplify theorem-prover. `http://research.compaq.com/SRC/esc/Simplify.html`.

[9] P. 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.

[10] 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.

[11] N. Dodoo. Selecting predicates for conditional invariant detection using cluster analysis. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, Sept. 2002.

[12] N. Dodoo, A. Donovan, L. Lin, and M. D. Ernst. Selecting predicates for implications in program analysis, Mar. 16, 2002. Draft. `http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.ps`.

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

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb. 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.

[15] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. 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.

[16] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. 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.

[17] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.

[18] M. P. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Trans. Prog. Lang. Syst.*, 4(4):527–551, 1982.

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

[20] 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, Mar. 1992.

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

[22] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, Nov. 6–10, 2001.

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

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

[25] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

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

[27] G. 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.

[28] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Palo Alto, CA, 1980. Also published as Xerox Palo Alto Research Center Research Report CSL-81-10.

[29] J. W. Nimmer and M. 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.

[30] J. W. Nimmer and M. 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, Nov. 20–22, 2002.

[31] 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, Apr. 1998.

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

[33] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to Trellis/Owl. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, Portland, OR, USA, June 1986.

[34] M. E. Segal and O. Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, 10(2), Mar. 1993.

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

[36] R. K. Weiler. Automatic upgrades: A hands-on process. *Information Week*, Mar. 2002.

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

[38] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, Oct. 1997.