Early Identification of Incompatibilities in Multi-component Upgrades

Stephen McCamant and Michael D. Ernst

Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge MA USA
smcc@csail.mit.edu, mernst@csail.mit.edu

Abstract. Previous work proposed a technique for predicting problems resulting from replacing one version of a software component by another. The technique reports, before performing the replacement or integrating the new component into a system, whether the upgrade might be problematic for that particular system. This paper extends the technique to make it more applicable to object-oriented systems and real-world upgrades. First, we extend the theoretical framework to handle more complex upgrades, including components with internal state, callbacks, and simultaneous upgrades of multiple components. The old model is a special case of our new one. Second, we show how to handle four real-world situations that were not addressed by previous work: non-local state, non-determinism, distinguishing old from new incompatibilities, and lack of test suites. Third, we present a case study in which we upgrade the Linux C library, for 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.

1 Introduction

A frequent cause of software failures is the use of software in unexpected or untested situations, in which it does not behave as intended or desired. Such problems are inevitable because it is impossible to foresee, much less to test, every possible situation in which software might be used. As one example, consider a software system that successfully uses a component. A supposedly compatible software upgrade may cause system failure or misbehavior if the system uses the new component in a manner for which it was not designed or tested. Even if the component developer conscientiously tests the component in many situations, the new component may not have been tested in an environment like the user's, or the developer may have inadvertently changed the behavior in the user's environment.

This paper builds on previous research [15] that seeks to identify unanticipated interactions among software components, before the components are actually integrated with one another. The 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, for the way that the component is used in an application. The 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 the method are formally capturing observed behaviors and a test that compares those behaviors via logical implication. The observed behavior is captured via dynamic detection of likely program invariants [9], 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 verify, and when available they may fail to capture all of the properties on which program correctness depends.

The previous technique has a number of positive qualities. It is application-specific, so it can indicate that an upgrade is safe for one client but unsafe for a different client. It operates before integrating the new component into the system (perhaps even before deciding whether to purchase the new component!) or running system tests. It is automated and does not require writing or proving formal specifications. It issues warnings for errors made by either the component vendor or the component client, and it does not require the vendor to have any knowledge of client behavior. It does not require an oracle indicating correct behavior, and it does not require access to source code.

However, the previous technique suffers a number of shortcomings, which we address in this paper. Most seriously, the upgrade model is overly simplistic: it is more applicable to functional than to object-oriented programs. It assumes that the upgrade involves a single module being upgraded, that the system interacts with the module only by calling it, and that each such call is independent. No accommodation is made for simultaneous upgrades of multiple components, or for components that keep internal state or make callbacks, as is common in object-oriented frameworks. Furthermore, in applying the technique to a real-world component, we discovered four circumstances that arise in practice and that require extensions to the technique. The previous technique is too permissive (it issues too few warnings) when a component's behavior depends on non-local state. The previous technique is too restrictive (it issues too many warnings) when procedure results are non-deterministic (or depend on unavailable facts), when test suites are insufficient, and when pre-existing apparent incompatibilities are present that did not prevent correct system behavior in the past. This paper addresses all of these issues, so its technique covers the essence of objects. The paper also describes a case study in which we upgraded the Linux C library and observed the predicted and actual effects on 48 Unix programs.

The remainder of this paper is organized as follows. Section 2 outlines the technique for detecting incompatibilities, in the simplest case of upgrading a single component that is called by the rest of the system. Section 3 extends the framework to accommodate more sophisticated interactions between upgraded components and the system. Section 4 gives examples of the more sophisticated interactions and shows how our implementation handles them. Section 5 further extends the technique to handle non-local state, lack of test suites, nondeterminism, and pre-existing incompatibilities. Section 6 describes our case studies with the C library and 48 Unix programs. Section 7 discusses related work, and Section 8 concludes.

2 Overview: Upgrading a Component

This section describes our upgrade-checking technique in outline. For simplicity of exposition, we describe the case of a single component. Suppose that there is a complete software system, the *application*, that includes a separately developed module, the *component*. The component may be a library of procedures, a collection of classes, or a formally packaged component in the sense of COM or CORBA; we assume only that it is used according to some procedure call interface. The application is observed to function properly with some version of the component, and we ask whether it will still function correctly if that component is replaced by a different version.

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. Our implementation computes this abstraction using the Daikon tool [9]. 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 on both the implementation of the component and 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 supplies the new component's operational abstraction (with respect to its test suite) to the customer.

(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. The test determines whether 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. The specific test is described formally in Sect. 3. Our implementation performs this checking using the Simplify automatic theorem prover [5]. (Additional imple-

mentation details appear in a technical report [16].) 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 changed behavior 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 to indicate that the upgrade would perform compatibly.

3 Upgrades Involving Many Modules

The comparison technique described in [15] is appropriate for upgrades of a single component, containing a single procedure that is called from the rest of a system. It can easily be generalized to a component with several independent procedures (by checking the safety of an upgrade to each procedure 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 that arise in object-oriented systems, such as components with state, 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 the 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.

Call and return relations. Roughly speaking, each module is modeled as a black box, with certain inputs and outputs. 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' and f_r.

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. In a module M, $M(v|u_1, \ldots, u_k)$ is the data-flow relation from inputs u_1 through u_k to an output v. As a degenerate case, an *independent output* $M(v)$ is one whose value is not affected by any input to the module. A constant-valued output would be independent. 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.

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. Because each variable might have a large or infinite domain, 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

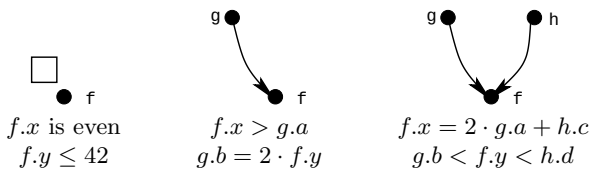

Fig. 1. Examples of data-flow relations over one, two, and three program points.

(observed to be) always true over the input and output variables. The values that satisfy these formulas are a 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. (Object-oriented dispatch is an example of such a situation; also see Sect. 4.3.) To keep track of when u might be followed by v, our technique computes a property ϕ 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 ϕ as the antecedent.

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. In a module M, we refer to the summary relation from outputs u_1 through u_k to an input v as $\overline{M}(v|u_1, \ldots, u_k)$. As a degenerate case, an *independent input* $\overline{M}(v)$ is one not affected by any outputs. 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.

Graphical representation. In explaining which conditions must be checked to assess the safety of an upgrade, it is helpful to represent the relational description of modules as a directed graph, in which nodes correspond to program points (module inputs and outputs). 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. Figure 1 shows our representation of relations. For examples of this graphical model, see Figs. 2, 6, and 10.

3.2 Considering an Upgrade

So far, we have described a model of the behavior of a modular system. For each module, its 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 the 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 [7, 10] or symbolic evaluation [19]. 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 [26]; our use of a functional representation that combines control flow with data dependence is reminiscent of the slicing algorithm of [8].

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, 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 [22], 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

$$\bigwedge_{\substack{\text{feasible } G \\ \text{for } \overline{M}_0(v_0|u_1,\ldots,u_k)}} \left[\left(\bigwedge_{M_i(v_i|\ldots)\in G} M_i(v_i|\ldots) \right) \Rightarrow \overline{M}_0(v_0|u_1,\ldots,u_k) \right] \quad (1)$$

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 in the previous paragraph, would be potentially inefficient, performing unnecessary logical comparisons. In the worst case, there may 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. 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 an important optimization 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.

3.3 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 [15] is a special case of the more general framework described in this section.

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.

Our technique claims that the new component may be safely substituted for the old one in the application if and only if

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

Our original reasons for choosing this formula, and its relation to alternative formulas, are explained in [15]; it is also the result given by the algorithm of Sect. 3.2. Chen and Cheng [4] prove formally, using a relational semantics, that this condition is the weakest (most general) condition on a component that is guaranteed to preserve application behavior. This formula is similar to, but more general than, the classic substitutability condition of behavioral subtyping [1, 14], used to check that objects of a subtype are a safe replacement for supertype objects. In our notation, the condition that behavioral subtyping imposes on a single functional method is that

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

4 Examples of Upgrades

The framework for upgrade safety checks described in Sect. 3 generalizes that of [15] to be applicable to more complex software systems, including object-oriented systems. 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

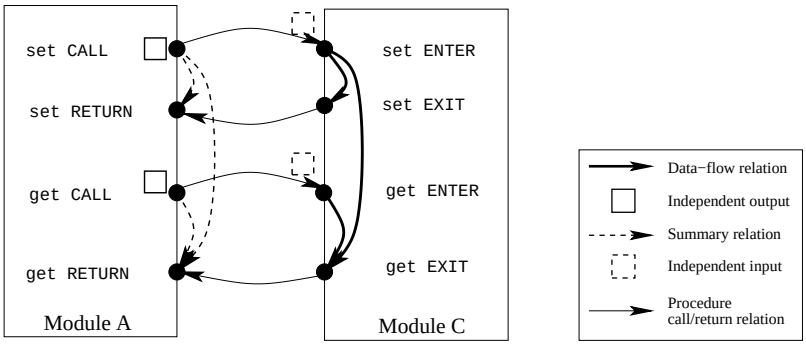

Fig. 2. A system with a module C whose procedures share state.

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

Fig. 3. Consistency conditions, derived from equation 1 of Sect. 3.2, for the system shown in Fig. 2; s and g represent the `set` and `get` procedures.

```
public class C {
  private int private_x;

  int set(int x) {
    private_x = x;
    return 0; // success
  }

  int get() {
    return private_x + 1;
  }
}
```

Fig. 4. Source code for a module with the structure of C from Fig. 2.

$A(s_c)$: $s.x$ 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	

Fig. 5. Operational abstractions for A and C as in Fig. 2. 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`.

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

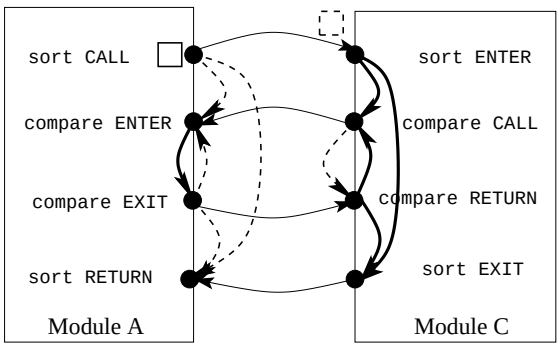

Fig. 6. A system with a module C that calls back to the using module A.

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

Fig. 7. Consistency conditions, derived from equation 1 of Sect. 3.2, for the system shown in Fig. 6; 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;
    }
  }
  ...
    obj.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;
        }
  }
}
```

Fig. 8. Source code for a module C and part of a module A with the structure shown in Fig. 6.

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

Fig. 9. Operational abstractions for A and C as in Fig. 6.

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 with Internal State

Figures 2 and 4 show a system in which one module, C, provides two procedures whose behavior is interdependent: the result of `get` depends on the previous call to `set`. Such dependencies often arise when methods share state in an object instance, but our approach is 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 Fig. 3 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 5 describe this situation, and it can be seen that the conditions in Fig. 3 do hold. For instance, consider the last condition: if $s.x$ is even, and $g'.return = s.x + 1$, then $g'.return$ will be odd.

4.2 Modules with Callbacks

Figure 8 shows source code from a system in which A calls C's `sort` procedure, which calls back to the `compare` procedure defined in A. Figure 6 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 Fig. 7 hold. For instance, the left-hand column of Fig. 9 gives an operational abstraction for A, which sorts only 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 Fig. 6 were observed: for instance, calls to `compare` were not inter-dependent. Again, we can easily see that conditions of Fig. 7 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.

4.3 More Than Two Modules

Figure 12 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 Fig. 10. 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 Fig. 11 (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.

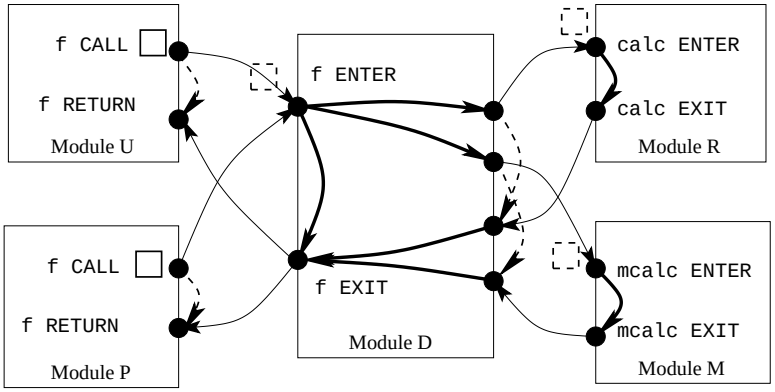

Fig. 10. A system consisting of five modules.

By checking the conditions of Fig. 11 using the operational abstractions shown in Fig. 13 (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 Sect. 3: observe that the corresponding properties in Fig. 13 are implications whose antecedents are properties over a variable of f.

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

Fig. 11. Consistency conditions, derived from equation 1 of Sect. 3.2, for the system shown in Fig. 10. f, c, and m represent the `f`, `calc`, and `mcalc` procedures respectively.

```
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 * 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;
  }
}
```

Fig. 12. Java-like pseudocode for modules D, R, and M as in Fig. 10.

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

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

$U_2(f_c)$: $f.o \in \{\mathtt{d}, \mathtt{i}, \mathtt{r}\}$
$R_2(c'|c)$: $c.o = \mathtt{r} \Rightarrow c'.return \equiv 0 \pmod{10}$
$c.o = \mathtt{r} \Rightarrow c'.return \leq c.i$
$c.o = \mathtt{i} \Rightarrow c'.return = c.i + 1$
$c.i \geq 0 \Rightarrow c'.return \geq 0$

$\overline{R}_2(c)$: $c.o \in \{\mathtt{i}, \mathtt{r}\}$
$\overline{U}_2(f_r|f_c)$: $f.o = \mathtt{d} \Rightarrow f'.return = 2 \cdot f.i$
$f.o = \mathtt{i} \Rightarrow f'.return = f.i + 1$
$f.o = \mathtt{r} \Rightarrow f'.return \equiv 0 \pmod{10}$
$f.o = \mathtt{r} \Rightarrow f'.return \leq f.i$

Fig. 13. Operational abstractions for modules in Fig. 10. 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 potential upgrades to the U and R modules respectively.

5 Enhancements to the Upgrade Technique

We have developed several additional enhancements that make the upgrade technique more effective in validating upgrades to complex software systems. These techniques are general solutions to specific problems that we encountered while running our tools to evaluate upgrades of the C library (Sect. 6). This section describes four improvements: a change to make more information about a program's behavior available to our system, which improves its accuracy; two techniques that indicate which detected behavioral differences are most relevant to upgrade safety; and a technique to avoid the need for a large test suite.

5.1 Including Non-local State Information

In order to conclude 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, the inputs that determine a subroutine's behavior are not all supplied as parameters or as object fields. For instance, in the Unix system-call interface, functions like `open` and `close` create and destroy stateful 'file descriptor' objects that are actually small integer indices into a table that exists only in the kernel.

This sort of extra information can be thought of as residing in virtual fields. The program's own (pure) accessor methods are one source of contents for such fields. Additionally, we used annotations to indicate values that should be virtual fields, for instance associating the file-descriptor pseudo-datatype with `fstat`, a function that returns a variety of information about a file.

5.2 Distinguishing Non-deterministic Differences

Section 5.1 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. For example, such non-determinism is often associated with errors.

Suppose that a return value, thrown exception, or side effect representing an error occurred during testing but never in an application's use. Then our technique would reject an upgrade, unless it could demonstrate that the application could never induce the erroneous behavior (as the application never had while using the old component). It is reasonable to establish this for a divide-by-zero error — say, if the application never passes in a zero value. However, other faults are effectively non-deterministic. It is not reasonable to predict a 'disk full' error by considering the hard disk's previous state as an input to every 'write file' operation. Failures that result from a physical fault like a broken cable or dust on a floppy disk are completely unpredictable.

For such effectively non-deterministic failures, we assume that if they never occurred with the old component, they will never occur with the new component either. This is unsound, but effective in practice. Our technique determines what results represent such failures by examining language features such as exceptions and error codes, possibly augmented by annotations.

5.3 Highlighting Cross-version Incompatibilities

When our technique issues a warning, the warning might be an indication of a behavioral difference (or use of undefined functionality) between the two versions of the component. On the other hand, the upgrade may be a valid, behavior-preserving upgrade, but the warning results from insufficient testing, an inadequate grammar of the operational abstraction, or a theorem proving weakness.

This section proposes a post-processing technique that aims 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. Classifying warnings permits users or tools to focus on those that are most likely to result from a behavioral difference.

The postprocessing step first considers a *self-upgrade* from the old module to itself. Such an 'upgrade' is always behavior-preserving, but our technique might still fail to verify the upgrade's safety. Any warning is a false alarm, and is likely to also be issued for the real upgrade. By contrast, a warning that occurs only with the new module, but not the old, certainly represents a behavioral difference, a *cross-version* incompatibility. Our technique highlights these cross-version warnings for the user's immediate consideration. (An earlier version of this idea was mentioned in [15], under the name 'meta-comparison', but not developed, implemented, or evaluated.)

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.

5.4 Using Other Applications as a Test Suite

Extensive testing is an important part of software engineering, but not all software has a large formal test suite, nor are operational abstractions from those formal test suites necessarily available to users considering an upgrade. When an organized test suite is unavailable, the role of the 'test suite' in our technique can instead be played by other applications. For each application, we use as the 'test suite' all uses of the new component by all of the other available applications. Analogous to the 'late adopter' practice of letting one's colleagues use a new software version first, this is effective if the other uses of the component are both sufficiently extensive and sufficiently similar to the uses of the application in question. 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 Sect. 5.3 should also be used, to reduce the number of warnings that indicate only insufficient testing.

6 Case Studies

In order to test our techniques, we performed case studies of upgrading a large software component, the Linux C library. On Unix systems a single library, traditionally named `libc`, provides the C standard library functions, wrappers around the low-level system calls, and miscellaneous utility functions. Most Linux systems use version 2 of the GNU C library [11], 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. This mechanism assists with maintaining compatibility and avoiding incompatibility, but it is insufficient. We subverted this 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 in some cases 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 (approximately 1000 in all), and records the arguments to and results from each invocation.

6.1 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 programs from version 7.3 of the standard Red Hat Linux distribution.

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. 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 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 suite, the role of the 'test suite' in our case studies is instead played by the other applications, as described in Sect. 5.4. The subject programs called 199 instrumented library procedures. Because our technique

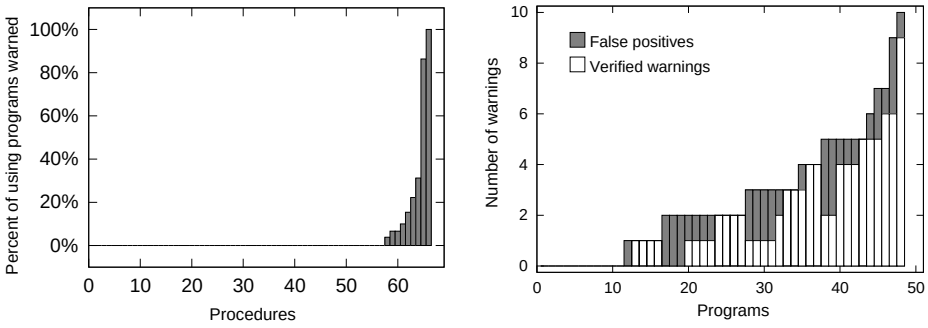

Fig. 14. Reported incompatibilities between C library versions 2.1.3 and 2.3.2. On the left, for 66 procedures whose behavior did not change, the percentage of the programs that used that procedure for which a behavioral difference warning was reported (false positives). On the right, the number of warnings produced per program. Unshaded bars show incompatibilities that we have verified by hand. Shaded bars show warnings that are probably false positives.

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. Our tool also reports a number of behavioral differences arising from 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 our technique finds are not relevant to programs that correctly treat the structure as opaque.

The 9 false positive incompatibility warnings are summarized in on the left in Fig. 14. The two tallest bars correspond to `tcgetattr` and `select`. When `tcgetattr` 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. For `select`, two expected properties fail to hold: one bounding a return value indicating the number of microseconds left to wait when the procedure returns, and one concerning a field that our tools 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 on the right in Fig. 14. As would be expected, larger applications have more potential for incompatibility: the two programs with the most warnings were Netscape Communicator and GNU Emacs.

6.2 C Library Incompatibilities

We 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 Sect. 6.1 because they were used by too few of those programs, though one of the differences exists between the versions considered there.

The `mktime` procedure The `mktime` procedure converts date and time values specified with separate components (year through seconds) into a single value of type `time_t`, which is traditionally a signed integer counting seconds since the 1970 'epoch'. If the time cannot be so represented, `mktime` returns -1. 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 [25] (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` procedure in the version of the C library on a Red Hat 7.3 workstation (Red Hat version 2.2.5-43), 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 Sect. 5.4.

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`. Specifically, when `mktime` completes successfully, it updates several fields of the supplied time structure, but when it returns an error, these fields remain uninitialized. The new version of `mktime` gives these uninitialized values for pre-1969 dates when the old version did not. 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 underlying cause as one discovered in a previous case study of Perl modules [15]; however, this manifestation is completely different and its effects were discovered in a different way and in different programs.

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 two integers counting seconds 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. During the summer of 2003, this time conversion 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, the change was reverted in response to user complaints.

Our technique recognized this change. We compared the behavior of the system 2.2.5 version of the C library 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`. 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 Sect. 5.1.

7 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 it is applicable in more situations.

7.1 Subtyping and Behavioral Subtyping

Strongly typed object-oriented programming languages, such as Java, use subtyping to indicate when component replacement is permitted [23, 2, 3]. 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 [1, 14]: 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'.

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. 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.

7.2 Specification Matching

Zaremski and Wing generalize behavioral subtyping to consider several varieties of matching between specifications [28]. 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 [21, 24].

Though they considered a large number of possible comparison formulas, Zaremski and Wing omitted the one that we adopted for our single-component upgrade [15]. Formulas equivalent to the single-component formula have been used for reuse (sometimes called the "satisfies" match [21]) and in work building on behavioral subtyping [6]. Also, in the VDM tradition [12], 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 [28]. Schumann and Fischer use an automated theorem prover with some specialized preprocessing [24]. 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.

7.3 Other Component-Based Techniques

The 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 [13]. A more common approach has been to abstract component behavior with finite state representations such as regular languages [20] or labeled transition systems [17]. 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. Incorporating temporal properties into our framework might be a fruitful direction for future work.

7.4 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. 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 [27]. For instance, techniques based on symbolic evaluation can verify the correctness of changes made by an optimizing compiler, such as common subexpression elimination [18]. If a program change was subtle enough to require human expertise in its application, it is probably too subtle to be proved sound automatically.

8 Conclusion

We have provided a technique for predicting problems resulting from behavioral differences among purportedly compatible versions of software components. The technique runs before the new versions are integrated or system tests are run. It logically compares two subsets of behavior: tested behavior and behavior used by an application. The technique is based on a rich component model, capturing many situations common in object-oriented frameworks, such as multiple simultaneous upgrades, shared state, callbacks, and indirect communication through

the system. The logical test generalizes that used by previous work, subsuming the previous test as a special case. We also extended the technique to situations that arise in real-world code: non-local state, apparent non-determinism, innocuous pre-existing incompatibilities, and lack of 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.

References

1. America, P., van der Linden, F.: 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), Ottawa, Canada (1990) 161–168
2. Black, A., Hutchinson, N., Jul, E., Levy, H., Carter, L.: Distributed and abstract types in Emerald. IEEE Transactions on Software Engineering **13** (1987) 65–76
3. Cardelli, L.: A semantics of multiple inheritance. Information and Computation **76** (1988) 138–164
4. Chen, Y., Cheng, B.H.C.: A semantic foundation for specification matching. In: Foundations of Component-Based Systems. Cambridge University Press, New York, NY (2000) 91–109
5. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA (2003)
6. Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, IEEE Computer Society Press (1996) 258–267
7. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18** (1975) 453–457
8. Ernst, M.D.: Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA (1994)
9. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27** (2001) 1–25 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.
10. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK (2001) 193–205
11. Free Software Foundation: GNU C library (2003) `http://www.gnu.org/software/libc/libc.html`.
12. Jones, C.B.: Systematic Software Development using VDM. Second edn. Prentice Hall (1990)
13. Leavens, G.T., Dhara, K.K.: Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In: Foundations of Component-Based Systems. Cambridge University Press, New York, NY (2000) 113–135
14. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems **16** (1994) 1811–1841

15. McCamant, S., Ernst, M.D.: Predicting problems caused by component upgrades. In: Proceedings of the 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Helsinki, Finland (2003) 287–296
16. McCamant, S., Ernst, M.D.: Predicting problems caused by component upgrades. Technical Report 941, MIT Laboratory for Computer Science, Cambridge, MA (2004) Revision of first author's Master's thesis.
17. Moisan, S., Ressouche, A., Rigault, J.P.: 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)
18. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, Vancouver, BC, Canada (2000) 83–94
19. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation, Montreal, Canada (1998) 333–344
20. Nierstrasz, O.: Regular types for active objects. In: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press (1993) 1–15
21. Penix, J., Alexander, P.: 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)
22. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA (1995) 49–61
23. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., Wilpolt, C.: An introduction to Trellis/Owl. In: Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR, USA (1986) 9–16
24. Schumann, J., Fischer, B.: NORA/HAMMR: Making deduction-based software component retrieval practical. In: Proceedings of the 1997 International Conference on Automated Software Engineering (ASE '97), Lake Tahoe, California. (1997) 246–254
25. The Open Group, ed.: The Single UNIX Specification, Version 3. The Open Group (2003) `http://www.unix.org/version3/`.
26. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3** (1995) 121–189
27. Yang, W., Horwitz, S., Reps, T.: A program integration algorithm that accommodates semantics-preserving transformations. ACM Transactions on Software Engineering and Methodology **1** (1992) 310–354
28. Zaremski, A.M., Wing, J.M.: Specification matching of software components. ACM Transactions on Software Engineering and Methodology **6** (1997) 333–369