Formalizing Lightweight Verification of Software Component Composition

Stephen McCamant Michael D. Ernst
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139 USA
smcc@csail.mit.edu, mernst@csail.mit.edu

ABSTRACT

Software errors often occur at the interfaces between separately developed components. Incompatibilities are an especially acute problem when upgrading software components, as new versions may be accidentally incompatible with old ones. As an inexpensive mechanism to detect many such problems, previous work proposed a technique that adapts methods from formal verification to use component abstractions that can be automatically generated from implementations. 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. The technique is based on a rich model of components that support internal state, callbacks, and simultaneous upgrades of multiple components, and component abstractions may contain arbitrary logical properties including unbounded-state ones.

This paper motivates this (somewhat non-standard) approach to component verification. The paper also refines the formal model of components, provides a formal model of software system safety, gives an algorithm for constructing a consistency condition, proves that the algorithm's result guarantees system safety in the case of a single-component upgrade, and gives a proof outline of the algorithm's correctness in the case of an arbitrary upgrade.

1. INTRODUCTION

Previous work [12, 13] introduced a technique that seeks to identify unanticipated interactions among software components, before the components are actually integrated with one another. The technique compares the observed behavior of an old component to the observed behavior of a new component; it permits the upgrade only if the behaviors are compatible, for the way that the component is used in an application. The technique 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 technique constructs *operational abstractions*, mathematical statements syntactically similar to specifications that describe a component's behavior and its expectations about the behavior of other components. For a given system of components, the technique constructs a consistency condition that relates the expectations of one module to how they might be satisfied by the behaviors of others. This combination of the abstractions according to the consistency condition is then passed to an automatic theorem prover (our prototype uses Simplify [4]), and the upgrade is approved only if the consistency condition is verified to hold. We have used our implementation to find behavioral inconsistencies in large software systems — for instance, differences between versions in the behavior of the Linux C library, as used by desktop applications.

In the case of an upgrade to a single, purely functional module, the consistency condition that our technique checks is similar to the classic condition of behavioral subtyping relating procedure pre- and postconditions [1, 3]. For upgrades to more complex systems with arbitrary numbers of components, bidirectional interactions among them, and components with internal state, the consistency condition is more complicated. This work refines the multi-component system model from [13] and gives an improved algorithm for constructing a consistency condition.

In order to decide whether our changes to the algorithm are really improvements, we need a standard by which to judge our technique. The major new work described here is a formalization of the consistency checking problem. We can use this formalization to verify that consistency checks have desirable logical properties. Specifically, we wish to verify that the consistency condition is sound relative to the abstractions that describe the behavior of individual components. If an upgrade is approved by virtue of satisfying a consistency condition, and the behavioral abstractions that are related via that condition are safe approximations of the components' actual behavior, then the actual upgrade in question is *safe* — that is, the upgraded system satisfies specific properties that the original one did. The algorithm improvements we describe eliminate unsound aspects of our previous algorithm, and using them we describe a strategy for a relative soundness proof (though we have not yet completed the proof in all details).

The remainder of this paper is organized as follows. Section 2 compares our technique with other approaches to component-based verification. Section 3 describes a model of the structure of a multi-component systems. Section 4 formalizes a simplified version of the upgrade safety problem. Section 5 proves that the consistency condition we use in the simplest case of upgrading a single component is indeed sound. Section 6 proposes a general algorithm for constructing consistency conditions and gives a proof outline of the algorithm's correctness. Section 7 concludes.

2. COMPARISON WITH OTHER WORK

Many other researchers share our goal of making component-based development safer and more efficient, as well as the general approach of verifying that components will interact correctly based on an abstraction of their behavior. However, our approach takes as its starting point a somewhat atypical combination of four theses. We argue that the abstractions describing components:

- should be stated in an expressive language at the same abstraction level as concrete interfaces
- should describe concrete implementations and the way they are exercised by real test suites

- should be compiled and compared automatically
- need not be sound over arbitrary executions

The following subsections discuss these points in turn.

2.1 An expressive language of abstractions

The operational abstractions that our technique uses to represent a component's behavior are expressed as statements in a first order logic whose atomic statements can refer to the same concrete values that program statements can, and include the same primitive operators as the language itself. The statements can express the same sorts of properties that a programmer might consider important, for instance as might be checked in a conditional or assertion statement. Matching the tool's understanding to the developer's has two benefits. First, it helps the tool find properties that might be important for correctness. Second, when user interaction is required, such as after a potential incompatibility has been flagged, it makes it easier for the developer to understand the problem.

Much of the most important early work on specifications and their combination, such as behavioral subtyping [11] and the Vienna Development Method [10] used expressive specification languages similar to our operational abstractions. More recent work has seen a trend toward less expressive representations, especially finite state ones such as regular languages [16] or labeled transition systems [15]. Besides being more amenable to automatic checking, such representations also focus verification effort on a more limited set of properties, such as those related to global temporal ordering. Such approaches necessarily neglect aspects of correctness in a program's local behavior, and cannot even express non-finite-state properties involving integers or unbounded data structures.

2.2 Using real implementations

The operational abstractions used in our technique are different from formal specifications in that they describe software as it has actually been implemented, rather than as it is intended to perform. While formal specifications can be useful as part of the design of a system, or to assign responsibility for deviations from an interface, they are unavailable for most real systems. In particular, some of the most productive uses of formal specifications take advantage of the ability to describe a component at a high level of abstraction, so as to capture only the aspects of its behavior most important to a global architecture. While it is possible to describe the complete correctness conditions of a component at the level of concrete inputs and outputs in a formal specification, doing so is prohibitively expensive for any but the most critical systems.

By contrast, our technique's use of operational abstractions is intended to leverage the investments that developers already make in implementation and testing, and to discover potential problems that would affect actual system executions. We presume that the developers of individual system modules have checked locally to a module, informally and/or via unit testing, that those modules behave as intended. The job of our technique is to propagate the characterization of behavior embodied in such local checks and cross-check it for consistency with the expectations held by other separately developed modules in a large system. Unlike dynamic contract-checking [14, 7], our technique is intended to be used before system integration, rather than during execution, and discovers inconsistencies without assigning blame to one component or another.

2.3 Automatic generation and comparison

Our tool automatically derives operational abstractions from a component's implementation, as it is exercised by representative uses such as a test suite. This approach takes advantage of developer effort already expended in development and in choosing which aspects of behavior to test. Such derivation is of course not applicable to a specification-first methodology, but dynamically inferred properties are increasingly used for verification; in addition to the axiomatic-semantics style properties we use [6], other researchers have applied similar techniques to infer algebraic specifications [8] and temporal properties [2, 19].

Many component-based verification techniques use behavioral abstractions that can be compared automatically, at least for finite scopes, by conceptually simple techniques such as model checking or approaches based on finite automata. In theory, the unbounded-state properties that make up our operational abstractions are more difficult to operate on, with many operations in fact undecidable. However, we have not found our use of an automated theorem prover to be a major bottleneck in our technique, for two reasons. First, because propositional logic is a standard abstraction, we can treat the theorem prover as a black box, and ignore its internal complexities. Second, the properties we wish to check tend to be straightforward deductions from a general statement to a more specific one, involving only simple arithmetic and relations between variables. In previous work, a more common approach has been to combine human direction and a proof assistant tool [20]; this can increase assurance relative to a completely manual proof, but does not necessarily reduce the effort required. Schumann and Fischer use an automated theorem prover with some specialized preprocessing [18] to compare specifications for a procedure reuse application, but the space of examples they consider is quite small.

2.4 Soundness and precision

Operational abstractions describe a component's behavior in specific contexts, namely those in which the component was tested. Our approach does not require the operational abstractions to be sound as statements describing a component's execution in any context. Even if our technique had access to a sound description of a component's general behavior, we would still want it to separately record the contexts in which a component was used, to be able to verify that a system uses only tested behavior. The real limitation of our approach is that we cannot necessarily make this distinction between properties that are true in general and those that hold only in a restricted context.

At the same time we give up soundness, however, we gain a dual benefit of precision [5]. By virtue of their construction, every property (from a particular grammar) that fails to appear in an operational abstraction can be traced back to at least one concrete execution in which it was false, and any property that held over each observed execution will appear in the abstraction. In exchange for restricting our attention to specific system executions, we enjoy accurate information about them, avoiding the over-approximation that can come with approaches that are sound.

3. A MULTI-COMPONENT MODEL

This section describes a model of software systems that handles 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. This model differs from that of [13] in separating control flow from data flow in some situations. This separation allows a more precise determination of what parts of a system influence others (Section 6.1) and a sound treatment of data flow through non-local state (Section 6.2.4).

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 consistency checking is modular, but not simply compositional; it also summarizes each module's observations of the rest of a working system, and uses them as a basis for comparison with a proposed upgrade.

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 flow relations* represent the behavior of individual modules, in context: that is, the way in which each output of the module potentially depends on the module's inputs. *External summary relations* represent a module's observations of the behavior of the rest of the system: how each input to the module might depend on the behavior of the rest of the system and any previous outputs of the module.

3.1.1 Call and return relations

Roughly speaking, each module is modeled as a black box, with certain inputs and outputs. 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.

3.1.2 Internal flow relations

Internal 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 flow relation from inputs u_1 through u_k to an output v. In some cases, it is also helpful to decompose a flow relation into a number of *flow edges*, one connecting each input to the output. An *independent output* $M(v)$ is one whose value is not affected by any input to the module.

Conceptually, the flow 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 (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.

Flow relations capture the behavior of a module primarily in terms of relations over data in variables. However, a limited characterization of a system's control flow is required to correctly combine facts from different relations. To this end, we further model flow edges as being of two types: those that represent control flow as well as data flow information, or *control-flow edges* for short, and *data-flow edges* that represent the flow of data not mediated by control (say, communication via a shared variable). To represent conditional control flow, control-flow edges include an additional fact called a *guarding condition*. 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. A guarding condition ϕ_g is a property 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. If the control flow is unconditional, ϕ_g is simply "true."

In order to facilitate analysis of a model, we impose the restriction that the subgraph consisting of control-flow, call and return edges has no cycles. This restriction forbids mutual recursion between procedures when the procedures appear in different modules, but not the use of recursive procedures in the implementation of a module. Note that procedure calls in both directions between a pair of modules are not restricted as long as there can be no cycle of procedure invocations in different modules; for instance, callbacks are allowed. See Section 6.2.3 for further discussion of why we impose this restriction.

3.1.3 External summary relations

External summary relations are in many ways dual to internal 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.

4. FORMALIZING THE MODEL

The key properties of our technique depend on the relationship between the abstract model and the concrete behavior of real components, but reasoning about the full complexities of languages such as Java, Perl or C would be difficult. Instead, this section presents a very simple module-structured language, and describes the meaning of the system model for that language, as well as giving a precise notion of soundness for systems in that language. In the context of this formalization it is then possible to unambiguously discuss whether a consistency checking technique is sound.

To concentrate on the most important aspects of the consistency checking problem, the formalization also differs in two key ways from our actual technique. First, our real technique has a broad goal of preserving the correct behavior of a system after an upgrade, as well as ensuring that an upgraded system relies only on tested behavior. To unify these notions and make them precise, the formalized language includes assertion statements, and we say that an upgraded system is safe if no assertions fail. Second, our real technique characterizes behavior by generalizing from facts that were observed to be true over finitely many executions. In the formalization, we imagine that these generalizations are always sound, so that descriptions of a component's behavior will be true for any inputs, and descriptions of the conditions under which behavior is safe in fact guarantee safety for any inputs. Because our actual implementation lacks this soundness property, soundness results about the formalization correspond only to relative soundness properties of the real system: guarantees about safety are only as reliable as the operational abstractions on which they are based.

4.1 Language

The statements of our simplified programming language have the following grammar. (C stands for code, and D stands for dynamic program point, which is discussed in Section 4.2.)

$$\begin{array}{lll} C & ::= & C; C \mid v := E \mid \texttt{if}\ P\ \texttt{then}\ C\ \texttt{else}\ C \\ & & \mid v := M.f(v_1, ..., v_k) \mid D \\ D & ::= & M.f.\mathrm{DPP}(\texttt{enter}\ M.f) \\ & & \mid M.f.\mathrm{DPP}(\texttt{exit}\ M.f) \\ & & \mid M.f.\mathrm{DPP}(\texttt{call to}\ M.f\ \texttt{\#}n) \\ & & \mid M.f.\mathrm{DPP}(\texttt{return from}\ M.f\ \texttt{\#}n) \end{array}$$

Predicates P and side-effect-free terms E include variable references and an arbitrary set of function and predicate symbols, such as the integer operations $+$, $\times$, and $<$. M and f range over the names of modules and procedures respectively.

Procedure definitions have the form $M.f(v_1, \ldots, v_k) : C$. Their semantics are defined by a substitution that transforms a program into one without procedure calls. A call $v_r = M.f(\alpha_1, \ldots, \alpha_k)$ originally appearing in a procedure $M_2.f_2$ rewrites to

$$\begin{array}{l} M_2.f_2.\mathrm{DPP}(\texttt{call to}\ M.f\ \texttt{\#}i); \\ v'_1 := \alpha_1; \\ \ldots \\ v'_k := \alpha_k \\ M.f.\mathrm{DPP}(\texttt{enter}\ M.f); \\ C[v'_1/v_1] \cdots [v'_k/v_k][r'/return]; \\ M.f.\mathrm{DPP}(\texttt{exit}\ M.f); \\ v_r := r'; \\ M_2.f_2.\mathrm{DPP}(\texttt{return from}\ M.f\ \texttt{\#}i) \end{array}$$

where i and the primed variables are fresh. v_1 through v_k are called the parameter variables, and α_1 through α_k are the argument variables. The return value of a procedure is signified by a distinguished variable *return*. Recursion is prohibited, as are calls between procedures in the same module (which can be simulated with ahead-of-time inlining). Note that this restriction on recursion is stronger than the one imposed in the real implementation, which forbids only recursion between modules. Our formalized language has no iteration constructs, and is far from Turing-complete, but we believe that the complications of loop verification and nontermination are orthogonal to the questions we wish to address with the formalization, so we have chosen to omit them.

The parameters to a procedure, and *return*, are local to it, and cannot be mentioned outside the procedure. Additional locals can be obtained by declaring parameters and ignoring their values. All other variables are associated with a particular module, and can only be mentioned there. Each module has a special variable *fail* which is initially zero, but set to 1 if any assertion fails. It can be mentioned only via a special syntactic sugar `assert`(P) which is otherwise equivalent to `if` P `then` *fail* := *fail* `else` *fail* := 1. (For brevity in examples, we will sometimes abbreviate *return* as r and combine expressions and procedure calls in single statements.)

Modules may refer to other modules by name. A system is a collection of modules with a distinguished `main` procedure in one of the modules, such that all named references to other modules in a module can be satisfied by other modules in the system. An execution of the system is an execution of the main procedure, including the expansions of all called procedures (transitively), in which the initial values of all the variables are arbitrary, except that all the special *fail* variables are initially zero.

The semantics of the language are the usual ones, given by the following small-step relation $\mapsto$, which maps code and a store to either new code and a new store, or just a new store to signify termination.

$$\langle v := E, s\rangle \mapsto s[v := s(E)] \qquad \text{[assign]}$$

$$\langle D, s\rangle \mapsto s \qquad \text{[ppt]}$$

$$\frac{\langle C_1, s\rangle \mapsto \langle C'_1, s'\rangle}{\langle C_1; C_2, s\rangle \mapsto \langle C'_1; C_2, s'\rangle} \qquad \text{[seqProgress]}$$

$$\frac{\langle C_1, s\rangle \mapsto s'}{\langle C_1; C_2, s\rangle \mapsto \langle C_2, s'\rangle} \qquad \text{[seqElim]}$$

$$\frac{s(P)}{\langle \texttt{if}\ P\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \mapsto \langle C_1, s\rangle} \qquad \text{[ifTrue]}$$

$$\frac{\neg s(P)}{\langle \texttt{if}\ P\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \mapsto \langle C_2, s\rangle} \qquad \text{[ifFalse]}$$

A system execution is safe for a module M if at the end of execution, the *fail* variable of module M is still zero. A module is safe in a system if every execution is safe for the module, and a system is safe if every module in it is safe.

4.2 Program points and relations

The execution of a dynamic program point D marks a moment of execution; it has no other runtime effect, and may not appear in the original program. The name of a dynamic program point gives an event (in parentheses) and the procedure where the event occurs (before the 'DPP'). The values at a dynamic program point are the current values of all in-scope variables when the point expression evaluates.

Static program points S abstract over dynamic program points; for any procedure g in module M_1:

Static	Dynamic
$M{:}\,f$	$M.f.\mathrm{DPP}(\texttt{enter}\ M.f)$
$M{:}\,f'$	$M.f.\mathrm{DPP}(\texttt{exit}\ M.f)$
$M_1{:}\,M_2.f_c$	$M_1.g.\mathrm{DPP}(\texttt{call to}\ M_2.f)$
$M_1{:}\,M_2.f_r$	$M_1.g.\mathrm{DPP}(\texttt{return from}\ M_2.f)$

Static program points $M{:}\,f$ and $M_1{:}\,M_2.f_r$ are called input program points, and $M{:}\,f'$ and $M_1{:}\,M_2.f_c$ are output program points.

A flow relation $M(S^{\mathrm{o}}|S^{\mathrm{i}}_1, \ldots, S^{\mathrm{i}}_k)$ consists of an output program point S^{o} and zero or more input program points S^{i}_j, all belonging to a module M, along with a formula ψ over all the variables of the given program points. In addition, one or more edges from inputs S^{i}_j to the output may be control-flow relations, with associated guarding conditions ϕ_j. ψ and each ϕ_j are together required to be sound in the following sense:

For any system containing M and other modules, a dynamic instance of the flow relation is a dynamic program point corresponding to the output point, along with a dynamic program point corresponding to each static input point, such that no later dynamic point for the same input occurs before the dynamic output point. The flow relation holds over a dynamic instance if ψ holds over the values of its variables at the dynamic points. A flow relation is required to hold over any dynamic instance in any system containing the module, for any system inputs. In addition, for each control-flow edge, it must be the case that every instance of the input program point S^{i}_j at which the guarding condition ϕ_j holds is followed later in the execution order, without any intervening instances of the output program point, by an instance of the output program point such that S^{i}_j and the output point are part of an instance of the relation.

A summary relation $\overline{M}(S^{\mathrm{i}}|S^{\mathrm{o}}_1, \ldots, S^{\mathrm{o}}_k)$ consists of an input program point S^{i} and zero or more output program points S^{o}_j, all belonging to a module M, along with a formula ψ over the variables at all of the given program points. When the name of a summary relation appears in a logical formula, it stands for the formula ψ. ψ is required to soundly assure safety in the following sense:

For any system containing M and other modules, a dynamic instance of the summary relation is a dynamic program point corresponding to the input point, along with a dynamic program point corresponding to each static output point, such that no later dynamic point for the same output occurs before the dynamic input point. The summary relation holds over a dynamic instance if ψ holds over the values of its variables at the dynamic points. The summary relations of a module are required to have the property that if in any system, they all hold on each of their dynamic instances for any system input, then that execution must be safe for that module.

The variables in the formulas of flow and summary relations are named so that every name is qualified by the static program point it corresponds to; thus relations can refer separately to the value of a program variable at different program points.

A call relation consists of a procedure call program point, a procedure entrance point for the called procedure, and a formula that states that each formal parameter variable is equal to the corresponding actual argument variable. When the name of a call relation appears in a formula, it stands for this conjunction of equalities. Similarly a return relation consists of a procedure exit program point, a procedure return point for the procedure that the exit is the exit from, and a formula stating that the value of the return in the procedure returned to is equal to the value returned by the exiting procedure. When the name of a return relation appears in a formula, it stands for this equality.

5. SAFETY FOR A SINGLE COMPONENT UPGRADE

To illustrate the use of the formalism developed in the previous section, consider the simple case of an upgrade to a module that provides a single procedure without visible side effects. We will give our technique's consistency condition for such a system, and prove that it is sound. This result is analogous to Example 5 of [3], which proves that a similar condition between specifications is "reuse-preserving," based on a relational semantics of specifications. Our proof is more involved, because it addresses more explicit details such as the passing of procedure arguments. Explicitly formalizing these notions becomes important for more complicated systems (for instance, if a procedure might have multiple callers).

Consider a system consisting of two modules U and L. Library L contains a single procedure f, and U contains a single procedure m, which makes one or more calls to f. Furthermore, assume that f makes no use of any module-wide variables (except of course for uses of *fail* in assertions). We call this the single-component upgrade case.

We can model the system with two flow relations, $U(f_c)$ and $L(f'|f)$, and two dual summary relations, $\overline{L}(f)$ and $\overline{U}(f_r|f_c)$. Since every call to the procedure returns, the flow edge from f to f' is a control-flow edge with guarding condition "true." As a concrete example, one might imagine that the procedure f increments its argument, and that U happens to call f only with even integers; then $U(f_c)$ might be "f_c is even", $L(f'|f)$ might be "$f'.r = f.x + 1$", $\overline{L}(f)$ might be"$f.x$ is an integer", and $\overline{U}(f_r|f_c)$ might be "$f_r.r = f_c.x + 1 \wedge f_r.r$ is odd", while C and R would be simply $f.x = f_c.x$ and $f_r.r = f'.r$.

PROPOSITION 1. *If*

$$(U(f_c) \wedge C) \Rightarrow \overline{L}(f)$$

and

$$(U(f_c) \wedge C \wedge L(f'|f) \wedge R) \Rightarrow \overline{U}(f_r|f_c),$$

where C and R are the call and return relations for the call to and return from f, then the system of U and L is safe.

PROOF. First, we will check that the system is safe for L. Since L has only one summary relation, $\overline{L}(f)$, it suffices to check that $\overline{L}(f)$, which is a formula over the parameters to f, holds at each dynamic entrance to f (recall that f uses no module-wide variables, so the parameters to f are the only relevant variables for the execution of f). Now, each dynamic occurrence of $L\colon f$ is immediately preceded, except for intervening assignments of arguments to parameters, by a dynamic occurrence of $U\colon L.f_c$, by the definition of procedure expansion. Because $U(f_c)$ is a flow relation for U, on any execution, $U(f_c)$ will hold at each dynamic execution of $U\colon L.f_c$. Furthermore, the assignments of f's arguments to its parameters assure that C will hold at the occurrence of $L\colon f$, and since the parameters are disjoint from the arguments, the formula of $U(f_c)$ will continue to hold at that point. Thus, the formula $U(f_c) \wedge C$ will hold at each instance of $L\colon f$. Thus by assumption, $\overline{L}(f)$ will hold at each instance of $L\colon f$, completing the proof that L is safe.

Next, we'll check that the system is safe for U. Since U has only one summary relation, $\overline{U}(f_r|f_c)$, it suffices to check that $\overline{U}(f_r|f_c)$, which is a formula over the value returned by f given the arguments to f, perhaps along with other variables in m, holds for the actual arguments, the return value, and those other variables, for each dynamic execution of the return. Consider any particular dynamic instance of $\overline{U}(f_r|f_c)$, consisting of a call $U\colon L.f_c$ and a return $U\colon L.f_r$. By the structure of the procedure expansion, the instance of $U\colon L.f_c$ must be followed by an assignment of arguments to parameters, and an instance of $L\colon f$. Similarly, the instance of $U\colon L.f_r$ must be immediately preceded by a return assignment, and before that an instance of $L\colon f'$. Since, by the definition of a summary relation, the instance of $U\colon L.f_c$ was the most recent prior to the instance of $U\colon L.f_r$, and because only U calls f, it must also be that the instance of $L\colon f$ is the most recent prior to the instance of $L\colon f'$; thus, the instances of $L\colon f$ and $L\colon f'$ are related by the flow relation $L(f'|f)$. In other words, we know that $L(f'|f)$ holds over the parameters to and the return value from this dynamic invocation of f. Furthermore, $U\colon L.f_c$ and $L\colon f$ are separated only by the assignment of arguments to parameters, so C holds as a relation between the arguments and the parameters, and similarly R holds as a relation between the copies of the return value at $L\colon f'$ and at $U\colon L.f_r$. Finally, $U(f_c)$ is a flow relation for U, so it must hold at the same point $U\colon L.f_c$. In summary, we see that $U(f_c) \wedge C \wedge L(f'|f) \wedge R$ holds over the parameters, arguments, and return value of f, so by the assumed safety condition, $\overline{U}(f_r|f_c)$ also holds over that dynamic execution. Since we picked an arbitrary execution, $\overline{U}(f_r|f_c)$ holds for each dynamic invocation on any input, completing the proof that U is safe. □

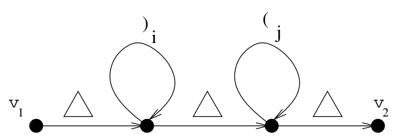
6. A MORE GENERAL CONDITION

The previous section described the consistency condition used by our technique when considering the simplest sort of component upgrade, and showed that it gave sound determinations of upgrade safety when used with sound abstractions of individual components. This section describes the algorithm for computing such conditions in arbitrary multi-component systems, and discusses the features that allow it to have the same soundness property. If desired, the algorithm can be performed separately for each summary relation in the model of a system, but we will describe it as checking all the relations together, in a series of three phases. First, for each summary relation it selects a subset of the model that is relevant to the expectations summarized in that relation; this process is analogous to the technique of slicing in program analysis. Second, it transforms the flow relations in the model so that they can be soundly combined to describe the behavior of modules working together. Finally, it combines the transformed flow relations in the subset of the model to construct a logical condition connecting the abstractions describing the behavior of various components to the expectations of the summary relation, so that if the condition holds, the summary expectations will be satisfied. The second and third phases are analogous to the construction of a verification condition in program verification, except that they operate using much larger atomic units of program behavior. The following subsections describe these three phases in turn.

In previous work [13] we described a simpler algorithm with the same purpose as the one described here, which combined the aforementioned three phases into one. However, the previous algorithm contained an ambiguity in its description, relating to the order in which the system graph was traversed, and the consistency conditions it produced could potentially lead to both false positive results (rejected safe upgrades) and false negatives (approved unsafe upgrades). In the present algorithm we focus on eliminating false negatives, to achieve soundness.

6.1 Selecting relevant relations

The set of data-flow relations that are relevant to a given summary relation can be determined using context-free language reachability [17] on the graph representing the model. Suppose that the edges of the model graph, excluding summary edges, are labelled as follows. Control-flow edges are labeled with $\triangle$. Procedure calls and returns are labelled with $(_i$ and $)_i$ respectively, where the indices i are chosen to be unique except that the call from and return to any particular site have the same index. Data-flow edges are replaced with sequences of three edges labeled with $\triangle$, connecting the original ends via two fresh vertices:



The fresh vertices are each adorned with a number of self-edges: one for each closing parenthesis $)_i$ on the first vertex, and one for each opening parenthesis $(_j$ on the second vertex. Now, let v be the input of a summary relation in such a graph. We say that a path from a node u to v is relevant if it is labelled by a word in the following context-free language:

$$\begin{array}{lcl} S & \rightarrow & R\,L \\ L & \rightarrow & Z \mid (_i\, L \mid B\,L \\ R & \rightarrow & Z \mid R\,)_i \mid R\,B \\ B & \rightarrow & Z \mid (_i\, B\,)_i \mid B\,B \\ Z & \rightarrow & \varepsilon \mid \triangle \end{array}$$

where productions with parentheses are repeated for all i. In other words, relevant paths include no mismatched parentheses; a $\triangle$ may appear anywhere. The set of all nodes u that start relevant paths can be determined by a dynamic-programming approach that enumerates all the triples consisting of two nodes and a nonterminal such that the path between the nodes can be labeled by the nonterminal. We say that an edge is relevant if it occurs on any relevant path. A relation is relevant if it contains any relevant edge (though in fact, either all or none of the edges in a relation will be relevant), or for an independent output if its node is on a relevant path.

The intent of this selection algorithm is to conservatively choose a subset of the system model whose behavior might affect the validity of a summary relation. The basic approach is similar to traditional interprocedural slicing: any path is considered feasible unless it violates the correct matching of procedure calls and returns. The treatment of data-flow edges, however, is non-traditional. Data-flow edges represent non-local dependencies between parts of a program, which need not obey the proper nesting of calls and returns: internal state might be set at one point in execution and read much later in an unrelated context. Thus, reachability across data-flow edges is granted an exemption from the usual matching of calls with returns, achieved via the data-flow edge rewriting shown above. Our treatment of state may be contrasted with the usual treatment of global variables in CFL-reachability-based program slicing [9], in which globals are threaded through procedures as extra parameters. While the traditional approach is potentially more precise, it relies on more detailed information about procedure implementations than is available in our framework.

Note that abstractly, this first phase of our algorithm is superfluous: one could obtain the correct results for each summary relation by using a consistency condition covering the entire system. However, including the first phase as described above has a number of practical benefits. First, as a matter of efficiency, existing automatic theorem provers are often unable to avoid consideration of supplied premises that are irrelevant to the statement being verified, especially when those premises include quantification. We would expect it to be more efficient to exclude irrelevant facts before passing them on to the theorem proving stage. Second, this slicing of the system model helps users of a tool track down and fix potential incompatibility warnings when they are generated. After a potential incompatibility is flagged, it is up to a user to decide which components must be modified or re-tested to allow the system to operate correctly. Knowing which components might be responsible for a failure reduces the scope of this search.

6.2 Transformations for sound composition

We aim to construct a consistency condition by conjoining formulas representing each relation in the relevant subgraph of the system model. We already assume that relations are sound with respect to the single module where they are found, but they must be changed to be sound over the larger domain of a combined system. Conjoining the formulas for each relation unmodified would lead to a consistency condition that might be satisfied even by an unsafe system. Recall that a flow relation from (input) program points $u_1, u_2, \ldots, u_k$ to an (output) program point v is a formula that holds at each dynamic instance of v, in terms of the values at the most recent preceding instances of each point u_i. In order to soundly combine relation formulas by conjunction, we must ensure that the variables in those formulas always consistently refer to the same set of values. In the following subsections, we explain where variable reference inconsistencies arise, and how we transform the relations to achieve sound combination.

6.2.1 Splitting relations into edges

Flow relations connect any number of inputs to a single output using formulas over the relevant variables at each program point. When combining information about multiple modules, though, it is more convenient to divide the relations into edges matching the graph structure of the model. Consider a relation from $u_1, \ldots, u_k$ to v. Introduce a fresh set of variables corresponding to all the variables at each of the points u_i, and rewrite any formula involving at least one of the u_i variables and a v variable to use the fresh copies of the variables from each u_i. Next, add formulas setting each original variable from u_i equal to its corresponding copy. Now, just these equations can be associated with each edge from a point u_i to v. The new set of equations represents the same relation between the variables at the points u_i and the variables at v that the original one did.

6.2.2 Guarding conditional control flow

Consider a control-flow edge from a point u to a point v. The 'meaning' of the edge is described with two formulas (as explained in Section 3.1.2). First, a relation ψ over the variables of both u and v describes data flow, holding for each occurrence of v over the variable values of that occurrence of v and the values at the most recent previous occurrence of u. Second, a guarding condition ϕ_g is a relation only over the variables of u, and holds on only on occurrences of u that are followed by an occurrence of v (without an intervening u). To construct an edge that can be used to soundly 'predict' the values at v given those at u, we combine these formulas into a new condition $\phi_g \Rightarrow \psi$. This new formula holds over every occurrence of u and the next following v: for any u, if u is not followed by v (without an intervening u), then ϕ_g will be false, making the implication true. On the other hand, if some u is followed by an occurrence of v, then that u is the most recent occurrence before v, so ψ holds over the values, and the implication is again true.

6.2.3 Duplicating based on calling context

Consider a procedure that is called from more than one module (say two); let e be its exit program point, and r_1 and r_2 the return program points for the two callers. The return relation between e and r_1 says that the return value seen by the caller is equal to the value returned, and similarly for the return relation between e and r_2. However, conjoining these relations would be unsound, because they are effectively referring to two different sets of values at e: one the values that will be returned to r_1, for the first caller, and the other only those to be returned to r_2. Given a property of the values returned to r_1 that distinguished them from the values returned to r_2, one might imagine using a technique similar to guarding to resolve this inconsistency, but that is not a feasible approach. Such a property may not even exist (if there are some values that could be returned to either caller), and even it did it exist it couldn't be determined in our modular approach, because it would require knowledge of all the calling modules.

Instead, the mismatch can be corrected by duplicating the program point e to create two points, e_1 connected to r_1 and e_2 connected to r_2, so that the variables at e_1 refer only to values on calls that return to r_1, and similarly for e_2. After this transformation, the return relations that equate e_1 with r_1, and e_2 with r_2, will be sound for any invocations of the restricted e_1 or e_2. Of course, if the program point e is split, we must also describe how the other edges ending at e are transformed. In the case of control-flow edges, we can repeatedly apply the same splitting technique to predecessor points until reaching the calls corresponding to the procedure returns; the net effect is to duplicate the representation of the procedure between the various call sites. For data-flow edges, see Section 6.2.4.

In practice, the duplication need not be performed step by step as it was just introduced. We can construct the right number of duplicates for every program point by simply traversing the system graph, maintaining a stack corresponding to the call stack of a system execution, and constructing one copy of a node for each unique stack contents (calling context) with which it might be reached. It is somewhat unfortunate that for soundness, our technique currently requires this extensive duplication of procedures called from multiple contexts. Because our model of the tested behavior of each module is collected without knowledge of the particular contexts where the module will be used, the context sensitivity provided by duplication should not be expected to significantly increase the precision of the technique's results. Rather, duplication seems necessary as a matter of soundness, for cases when two uses of a module are considered together, to avoid effectively assuming that a procedure's return value always took a single value, as would happen if it were represented by a single logical variable. Figure 1 shows an example where duplication appears necessary. Without duplication, one could conclude that the values returned by c and d are equal, and thus that the return value of b must be even, when in fact it must be odd. We are still looking for less extensive modifications that might achieve soundness while remaining essentially context-insensitive. The overhead incurred by duplication, though exponential in the worst case, should be modest in practice, because the number of modules comprising a system will be relatively small.

$$
\begin{array}{rl}
A.m(x): & x := x \cdot x + 1;\ r := B.b(x);\ \texttt{assert}\,(r > 4 \cdot x) \\
B.b(y): & r := C.c(2 \cdot y) + D.d(2 \cdot y + 1) \\
C.c(v): & r := E.i(v) \\
D.d(v): & r := E.i(v) \\
E.i(x): & r := x + 1
\end{array}
$$

$$
\begin{array}{l}
(A(b_c) \wedge \mathrm{Call}(B.b|A.b_c) \wedge B(c_c|b) \wedge \mathrm{Call}(C.c|B.c_c) \\
\wedge\, C(i_c|c) \wedge \mathrm{Call}((E.i)_c|C.i_c) \wedge (E(i'|i))_c \\
\wedge\, \mathrm{Ret}(C.i_r|(E.i')_c) \wedge C(c'|i_r) \wedge \mathrm{Ret}(B.c_r|C.c') \\
\wedge\, B(d_c|b) \wedge \mathrm{Call}(D.d|B.d_c) \wedge D(i_c|d) \\
\wedge\, \mathrm{Call}((E.i)_d|D.i_c) \wedge (E(i'|i))_d \wedge \mathrm{Ret}(D.i_r|(E.i')_d) \\
\wedge\, D(d'|i_r) \wedge \mathrm{Ret}(B.d_r|D.d') \wedge B(b'|c_r, d_r) \\
\wedge\, \mathrm{Ret}(A.b_r|B.b')) \qquad\qquad \Rightarrow \overline{A}(b_r|b_c)
\end{array}
$$

$$
\begin{array}{l}
(A.b_c.y > 0 \wedge B.b.y = A.b_c.y \wedge B.c_c.v = 2 \cdot B.b.y \wedge C.c.v = B.c_c.v \\
\wedge\, C.i_c.x = C.c.v \wedge (E.i.x)_c = C.i_c.x \wedge (E.i'.r)_c = (E.i.x)_c + 1 \\
\wedge\, C.i_r.r = (E.i'.r)_c \wedge C.c'.r = C.i_r.r \wedge B.c_r.r = C.c'.r \\
\wedge\, B.d_c.v = 2 \cdot B.b.y + 1 \wedge D.d.v = B.d_c.v \wedge D.i_c.x = D.d.v \\
\wedge\, (E.i.x)_d = D.i_c.x \wedge (E.i'.r)_d = (E.i.x)_d + 1 \wedge D.i_r.r = (E.i'.r)_d \\
\wedge\, D.d'.r = D.i_r.r \wedge B.d_r.r = D.d'.r \wedge B.b'.r = B.c_r.r + B.d_r.r \\
\wedge\, A.b_r.r = B.B'.r) \qquad\qquad \Rightarrow A.b_r.r > 4 \cdot A.b_c.y
\end{array}
$$

Figure 1: A small example of a consistency condition derived by the algorithm of Section 6. The three sections show code for a system, the form of the consistency condition as computed by the algorithm of Section 6, and the actual condition as passed to a theorem prover. The increment routine i of module E is called in different contexts by modules C (with an even argument) and D (with an odd argument). Duplication of the logical variables for procedure i is indicated by the notations $(\cdot)_c$ and $(\cdot)_d$.

6.2.4 Mixing data-flow edges

Data-flow edges must also be changed when the nodes they connect are duplicated, but they cannot be duplicated along with the nodes they connect in the way that control-flow edges are, because a data-flow edge does not correspond to any particular execution context. In fact, a data-flow edge might connect two nodes that are duplicated a different number of times. Our model is too abstract to determine which flows between copies of the source program point and copies of the target program point might occur (for instance, because it does not provide a total ordering of calls), so we instead conservatively assume that any flow might be possible. If an original source program point u is duplicated n times and a target point v is duplicated m times, we create nm copies of the data-flow edge originally connecting them, one connecting each duplicate of u to each duplicate of v. However, at each duplicate of v, the formulas for the edges are disjoined (rather than conjoined as formulas are otherwise). The effect is to express that each destination receives values flowing from at least one source.

6.3 Assembling a consistency condition

Once the relational model has been transformed as described in the previous subsection, assembling the consistency condition is straightforward. For a given summary relation, take the set of flow relations relevant to the summary input, as computed by CFL reachability. Let S be the set of all those relation formulas, as rewritten or duplicated according to the transformations above. Then the consistency condition states that the conjunction of all the formulas in S implies the summary relation formula.

In outline, the proof of the soundness of this technique is as follows, for a single summary relation. Suppose that the consistency condition holds; it is an implication of the form $(\bigwedge_i \phi_i) \Rightarrow \sigma$, where each ϕ_i is a transformed flow relation formula and σ is a summary relation formula. Each ϕ_i holds individually, due to the assumption that the original flow relations are sound and the fact that their transformations were soundness-preserving. Furthermore, the formulas ϕ_i use common variables consistently, so they can be legitimately conjoined, and their conjunction is true. Thus, by our assumption of the implication, σ must hold. By this argument, each summary relation σ in a system can be seen to hold, which guarantees, by the definition of a summary relation, that the system will be safe.

7. CONCLUSION

This research takes steps toward proving the soundness of a technique for verifying properties about a software system made up of components. One use of the technique is to verify that after some components of a software system have been upgraded, the system as a whole continues to behave as desired.

This paper makes five key contributions. First, we provided an abstract model of the behavior of software components. The model differs from previous models by distinguishing between control flow and data flow. Second, we gave a formalized version of the problem of checking consistency for a modular system in a simple, imperative language. Safe component composition is modeled by the success of arbitrary assertion statements. Third, we defined an algorithm that constructs a consistency condition. The consistency condition is a logical formula such that if the components satisfy its parts (which express expectations about component behavior), then the system as a whole behaves safely. This algorithm differs from previous work by applying to the new, more detailed model and by fixing an error in previous formulations. Fourth, we proved the correctness of the algorithm in the special case of a single-component upgrade: the condition that the algorithm generates suffices to ensure that an upgrade is safe. Fifth, we gave a proof outline of the correctness of the algorithm in the general case. Completing this proof is future work.

We made two types of changes to previous work. Some changes were motivated by the desire to prove correctness; for example, the proof required formalization of the safety condition. Other changes correct errors in previous work that were not apparent until revealed by our formalization and proof attempts. The result is a more detailed and correct technique for verifying the composition of software components.

Though our technique does not require its users to understand the complexities behind its operation, this research demonstrates that techniques from formal specification and verification can make possible a practical and lightweight tool to help software development.

Acknowledgment: This work was funded in part by NSF grant CCR-0133580, DARPA contract FA8750-04-2-0254, the Oxygen project, and an NDSEG fellowship.

8. REFERENCES

[1] P. America. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP*, pages 234–242, June 1987.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, Jan. 2002.

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

[4] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, Palo Alto, CA, July 23, 2003.

[5] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):1–25, Feb. 2001.

[7] R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE*, pages 229–236, Sept. 2001.

[8] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP*, pages 431–456, July 2003.

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, Jan. 1990.

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

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

[12] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC/FSE*, pages 287–296, Sept. 2003.

[13] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, June 2004.

[14] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[15] S. Moisan, A. Ressouche, and J.-P. Rigault. Behavioral substitutability in component frameworks: A formal approach. In *SAVCBS*, pages 22–28, Sept. 2003.

[16] O. Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, Sept./Oct. 1993.

[17] T. W. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11–12):701–726, Nov./Dec. 1998.

[18] J. Schumann and B. Fischer. NORA/HAMMR: Making deduction-based software component retrieval practical. In *ASE*, pages 246–254, Nov. 1997.

[19] J. Yang and D. Evans. Dynamically inferring temporal properties. In *PASTE*, pages 23–28, June 2004.

[20] A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM TOSEM*, 6(4):333–369, Oct. 1997.