Program Compatibility Approaches

Edmund Clarke[1], Natasha Sharygina[1,2], and Nishant Sinha[1]

[1] Carnegie Mellon University
[2] Universita della Svizzera Italiana

Abstract. This paper is a survey of several techniques that have proven useful in establishing compatibility among *behaviorally similar* programs (e.g., system upgrades, object sub- and supertypes, system components produced by different vendors, etc.). We give a comparative analysis of the techniques by evaluating their applicability to various aspects of the compatibility problem[1].

1 Introduction

Component-based development aims to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. The use of off-the-shelf components offers a great potential for: (1) significantly reducing cost and time-to-market of large-scale and complex software systems, (2) improving system maintainability and flexibility by allowing new components to replace old ones, and (3) enhancing system quality by allowing components to be developed by those who are specialized in the application area. Despite the advantages of the component-based approach, the use of commercial off-the-shelf software–especially when delivered as black-box components–has raised a number of technical issues. One of the fundamental problems relates to guaranteeing the safety of replacement of older components by their newer or upgraded counterparts. This problem is a particular instance of a more general task of checking compatibility between behaviorally similar program components. Among many approaches for component-based specification and design developed over the years (see an excellent overview in [25]), assessment of compatibility between different components remains a challenging task.

A limited answer to the component compatibility problem can be given by traditional type systems. It is well known [19], however, that type checking, while very useful, captures only a small part of what it means for a program to be correct. Instead it is necessary to establish a stronger requirement that ensures the behavioral correctness of components.

This paper provides a selective overview of several techniques that ensure the requirement of behavioral compatibility among components. The paper is organized as follows. Section 2 gives an overview of the interface automata formalism and describes the notions of compatiblity and substitutability as defined in this formalism. Section 3 presents a technique to check if upgrades to one or more components in a component

[1] The work described in section on substitutability check is based on a 2005 *Formal Methods* paper, *Dynamic Component Substitutability*, Lecture Notes in Computer Science 3582, 2005 by the same authors.

assembly are compatible with the other components in the assembly. Section 4 outlines ideas of behavioral subtyping which ensure that subtype objects preserve properties of their supertypes. Section 5 presents an automated and compositional procedure to solve the component substitutability problem in the context of evolving software systems. Finally, Section 6 provides a comparative evaluation of the presented techniques.

2 Interface Automata Compatibility

Interface automata [10] were proposed by Alfaro et al. for capturing the temporal input-output (I/O) behaviors of software component interfaces. Given a software component, these automata model both input assumptions about the temporal order of inputs and output guarantees about generation of outputs for the component. In contrast to similar formalisms like I/O automata [20], the interface automata approach handles both *composition* and *refinement* of automata differently. Two automata are said to be compatible if there exists *some* environment that can provide inputs so that the illegal states of the product automaton are avoided. Composition of two interface automata is defined only if they are mutually compatible. One interface automaton refines another if it has weaker input assumptions and stronger output guarantees. Both concepts are formally defined using game theory semantics. More specifically, they are defined in terms of a game between Input and Output players, which model the environment and the component interface automata, respectively.

The interface automata formalism relies on an optimistic approach to component composition. Composing two interface automata could lead to error states where one automaton generates an output that is an illegal input for the other automaton. However, the optimistic composition approach steers clear of the error states by checking if there exists a legal environment that does not lead the composed system to an error state. As opposed to the I/O automata approach [20] which allows an arbitrary input environment (which may lead the composed system to an illegal state), the interface automata assumes a helpful environment while computing the reachable states of the composed automaton. Algorithmically, such a composition is obtained by solving a game between the product automaton of the components (which attempts to get to an error state) and the environment (which attempts to avoid error states).

The following provides formal description of the interface automata formalism and the notions of the component composition and refinement.

An interface automaton is a tuple, $P = \langle V_P, V_P^{init}, A_P^I, A_P^O, A_P^H, T_P \rangle$, where :

- V_P is a set of states.
- $V_P^{init} \subseteq V_P$ is a set of initial states, having at most one state.
- A_P^I, A_P^O, A_P^H are mutually disjoint sets of input, output and internal actions. The set of all actions $A_P = A_P^I \cup A_P^O \cup A_P^H$.
- $T_P \subseteq V_P \times A_P \times V_P$ is a set of steps.

P is said to be *closed* if it has only internal actions, i.e., $A_P^I = A_P^O = \emptyset$, otherwise it is said to be *open*. If $(v, a, v') \in T_P$, then action a is said to be enabled at state v. The set $A_P^I(v)$ of enabled input actions specifies which inputs are accepted at the state v; the other inputs in $A_P^I \setminus A_P^I(v)$ are *illegal* inputs at that state.

2.1 Composition

Composition of two interface automata is defined only if their actions are disjoint, except that an input action of one component may coincide with the output action of another component, in which case it is called a shared action. Two automata synchronize on shared actions, and asynchronously interleave on all other actions.

Formally, two automata P and Q are composable if $A^I_P \cap A^I_Q = \emptyset$, $A^O_P \cap A^O_Q = \emptyset$, $A^H_Q \cap A_P = \emptyset$ and $A^H_P \cap A_P = \emptyset$. The shared actions $shared(P, Q)$ of the two automata are given by the expression $(A^I_P \cap A^O_Q) \cup (A^O_P \cap A^I_Q)$. If P and Q are composable interface automata, their product $P \otimes Q$ is the interface automaton defined by: $V_{P\otimes Q}$ = $V_P \times V_Q$, $V_{P\otimes Q}$ = $V^{init}_P \times V^{init}_Q$, $A^I_{P\otimes Q}$ = $(A^I_P \cup A^I_Q) \setminus shared(P, Q)$, $A^O_{P\otimes Q}$ = $(A^O_P \cup A^O_Q) \setminus shared(P, Q)$ and $A^H_{P\otimes Q}$ = $A^H_P \cup A^H_Q \cup shared(P, Q)$. The transitions $T_{P\otimes Q}$ are obtained by synchronizing P and Q on shared actions and asynchronously interleaving all other action steps. A state (v, u) in $P \otimes Q$ is said to be *illegal* if there exists a shared action $a \in P \otimes Q$, such that a is enabled in P at state v but is not enabled in Q at state u or vice-versa.

Compatibility. If the product $P \otimes Q$ is closed, then P and Q are compatible if no illegal state of $P \otimes Q$ is reachable from an initial state. When $P \otimes Q$ is open, then P and Q can be compatible if there exists a *legal* environment interface automaton E (composable with $P \otimes Q$) that can provide inputs to $P \otimes Q$ such that no illegal state is reachable in the composition $E \otimes (P \otimes Q)$. Alternatively, two interface automata P and Q are compatible iff (a) they are composable and (b) their composition is non-empty. If P and Q are compatible, then their composition can be computed by a polynomial time algorithm [10]. Composition of compatible interface automata is associative and hence can be computed in an iterative manner.

Intuitively, an interface automaton for a system component represents both assumptions about the environment and guarantees (or the observed outputs) of the specified component. Two assumptions are made about the environment: (i) each output step of the component must be accepted by the environment as an input and (ii) if an input action is not enabled at a state of a component, then the environment does not provide it as an input. The component guarantees consist of the behavior sequences and choices of input, output and internal actions at each state of the automaton. A drawback of this formalism is that one can construct trivial legal environments to show compatibility of components; an environment that generates no inputs for $P \otimes Q$ can trivially avoid the illegal states of $P \otimes Q$. In other words, the formalism can not express the fact that specific inputs must be present in the environment.

2.2 Refinement

The refinement relation formalizes the relation between abstract and concrete versions of the same component. The usual approaches to check refinement are based on the trace containment or simulation preorder relations. It is argued that the former notions are only suitable for input-enabled systems, where all input actions at each state are always enabled. In contrast, for non-input-enabled systems like interface automata, a refinement check based on checking *alternating simulation* [10] is proposed. The key idea here is that, in the non-input-enabled setting, the implementation must allow *more* legal inputs and exhibit *fewer* outputs, than the specification.

Intuitively, an interface automaton Q refines another interface automaton P (written $Q \preccurlyeq P$), if there exists an alternating simulation relation $\preccurlyeq$ between Q and P, i.e., all input steps of P can be simulated by Q and all output steps of Q can be simulated by P. Moreover, $A_P^I \subseteq A_Q^I$ and $A_Q^O \subseteq A_P^O$. Note that both P and Q may also have internal steps, which are independent from each other. Therefore, the definition of alternating simulation is extended to handle internal steps. Given a state v of an interface automaton P, $\epsilon - closure_P(v)$ is defined to be the set of states (including v) that are reachable from v in P via internal actions. Given a state u in Q, $\epsilon - closure_Q(u)$ is defined similarly. Let $I_P(v)$ and $I_Q(u)$ denote the set of input steps enabled at *all* states in $\epsilon - closure_P(v)$ and $\epsilon - closure_Q(u)$, respectively. Similarly, let $O(v)$ and $O(u)$ denote the set of output steps enabled at *some* state in $\epsilon - closure_P(v)$ and $\epsilon - closure_Q(u)$, respectively. Also let $S_P(v, a)$ denote the set of all successors v'' in P such that for some $v' \in \epsilon - closure_P(v)$, $(v', a, v'') \in T_P$. $S_Q(u, a)$ is defined similarly.

Now, a binary relation $\preccurlyeq \subseteq V_Q \times V_P$ is an alternating simulation from Q to P if for all states $v \in V_P$ and $u \in V_Q$ such that $u \preccurlyeq v$, the following conditions hold:

1. $I(v) \subseteq I(u)$ and $O(u) \subseteq O(v)$.
2. For all $a \in I(v) \cup O(u)$ and all states $u'' \in S_Q(u, a)$, there is a state $v'' \in S_P(v, a)$ such that $u'' \preccurlyeq v''$.

Finally, Q is said to refine P, if there exist states $v \in V_P^{init}$ and $u \in V_Q^{init}$ such that $u \preccurlyeq v$ and both $A_P^I \subseteq A_Q^I$ and $A_Q^O \subseteq A_P^O$ hold. Refinement between interface automata is a preorder, i.e., reflexive and transitive.

This notion of refinement has two useful properties:

– *Substitutivity.* If Q refines P and is connected to the environment by the same inputs, then we can always replace P by Q in a larger system. Note that Q must have no more inputs than P since incompatibilities may occur when environment presents those inputs to Q.
– *Compositionality.* In order to check if $Q \parallel Q' \preccurlyeq P \parallel P'$, it is sufficient to check $Q \preccurlyeq P$ and $Q' \preccurlyeq P'$, separately.

Interface automata, as defined above, execute asynchronously. The formalism has been extended to synchronous interfaces [6]. A general formalism relating components and their interface models has been developed [11] using the notion of interface automata.

3 Checking Compatibility of Upgrades

McCamant and Ernst present a technique [21] to check if upgrades to one or more components in a component assembly (also referred to as an *application*) are compatible with the other components in the assembly. More precisely, their work seeks to identify unanticipated interactions among software components as a result of an upgrade, before the older components are replaced by the upgraded ones. Their approach is based on computing a summary (a set of pre- and post-condition pairs on interface variables) of the observed behavior of the old and new components as well as a summary of the environment components that interact with the old component. An upgrade is permitted only if these summaries for the old and new components are compatible; otherwise, a warning is issued together with a witness that illustrates the incompatibility. The technique uses a large number of input test sequences and valid executions of components to compute the summary of the input/output behavior of a component and its environment. The compatibility check procedure crucially depends on the computed summaries and the authors have used it successfully to detect incompatibilities during upgrade of the Linux C library.

3.1 Single Component Upgrade

We now describe their approach for the case of a single component upgrade in an application. It is assumed that the application is observed to function properly with some version of the component and the verification task is to validate that the application will function correctly after upgrading the component to a new version. The upgrade compatibility check technique first computes an operational abstraction (summary) of the behaviors of the old component that are used by the rest of the system. The summary is computed using the tool Daikon [14] for automatically inferring program invariants from a representative set of program behaviors and consists of pre- and post-condition tuples for the component. The new component vendor also computes this observation summary for the new component, based on the test suite for the component and provides the summary to the compatibility check procedure. The procedure then compares the old and the new operational abstractions to test if the new abstraction is stronger than the older one. More precisely, the procedure checks if the new abstraction is able to accept as many inputs and produces no more outputs than the old abstraction. This check is performed using the Simplify theorem prover [12].

If the test succeeds, then the new component can be safely used in all situations where the old component was used. Otherwise, the check provides feedback about the incompatibility in terms of specific procedure pre- and post-conditions. In spite of an incompatibility being present, it may still be the case that the new component is acceptable. The final decision is made by examining the feedback manually.

3.2 Multiple Component Upgrades

Handling upgrades for components with persistent state (e.g., due to variables in object-oriented programs) or callbacks is more difficult. Similarly, handling simultaneous upgrades for multiple components requires more sophisticated analysis. For this analysis, an application is sub-divided into *modules* with the assumption that any upgrade affects one or more complete modules. Now, the upgrade compatibility check technique not only computes an observational summary of each upgraded module in the context of its environment modules, but also computes a summary of the behavior of environment modules, as observed by that module.

The observational summaries consist of three types of relations. (1) *Call and return relations* represent the procedure call dependencies between modules. (2) *Internal* *data-flow relations* represent how the output of a module depends on the module's inputs. This relation is stored as logical formula on input and output variables of a module and generalizes upon the observed behaviors of the module during testing. (3) *External summary relations* represent how each input to the module might depend on the behavior of the rest of the system and any previous outputs of the module. These relations may be considered to be dual of the internal data-flow relations and represent assumptions about how the module is used in the application.

In the event of an upgrade of multiple modules, it is assumed that each upgraded module is accompanied by sets of new data-flow and external summary relations. The compatibility check is then performed by checking if the new summary relations obey the old relations. This check is performed using the notion of a *feasible subgraph* [21] for a summary relation, which captures a subset of system executions over which the summary relation should hold. An upgrade is considered to be safe if it allows each summary relation to hold over every corresponding feasible subgraph. Several optimizations are presented aimed at reducing the number of feasible subgraphs to be re-validated for each summary relation.

As an example, consider an application with two modules, U and C, where C is supplied by a vendor and provides a procedure `f` and U calls `f`. The calls and returns to `f` are denoted by f and f', respectively. The summary relation associated with C consists of two parts: (i) the preconditions of `f` (assumptions on external components) represented by $\overline{C}(f)$ and (ii) a data-flow relation $C(f|f')$ describing the postconditions of `f`. Both these relations are based on the vendor's testing of C. Similarly, the relations associated with U are as follows: $U(f)$ consists of a set of preconditions describing how U calls `f` and $\overline{U}(f|f')$ describes the postconditions U expects from the call. In order to verify that the new component C may be safely substituted for the old one in the above application, the following checks are performed:

$$U(f) \implies \overline{C}(f) \quad (U(f) \wedge C(f'|f)) \implies \overline{U}(f'|f)$$

Here, the first check makes sure that all preconditions $\overline{C}(f)$ of `f` in C are met by postconditions $U(f)$ of U and the second check ensures that the return values from call to `f` satisfy the expectation of the module U. This procedure has been extended to handle upgrades for modules with internal state and callbacks and also for simultaneous upgrades of multiple modules.

3.3 Case Studies

The authors evaluate their technique on upgrades of the GNU C library distributed with Linux systems. The library provides the C standard library functions and wrappers around the low-level system calls and is used by a large number of commonplace applications on Linux. It contains multiple versions of some procedures in order to maintain backward compatibility and the linker is responsible for compiling the correct versions together. The authors subverted the version compatibility checks and checked if the procedures marked as incompatible can be used without error by the client applications and whether differences between procedures marked with the same version can cause errors. Binary versions of the applications and the C library were used together with a wrapper around the C library, which keeps track of the arguments to and from each function call in the library.

The authors compare two versions of the C library and used the set of common Linux applications as a ”test suite” for constructing the observational summaries. Of the selected 76 procedures in the library, the tool correctly warns of 10 behavioral differences and approves 57 upgrades as compatible. For the remaining 9 procedures, an (spurious) incompatibility is detected with respect to summaries computed with one or more test applications. A different experiment examined two incompatible upgrades of another set of procedures from the C library, where the tool was able to detect the incompatibilities.

4 Preservation of Program Properties by Behavioral Subtyping

The problem of behavioral consistency among various versions of programs is also addressed in the work of Liskov and Wing [19]. This work explores ideas of behavioral subtyping using invariants and constraints. It defines the subtype relation that ensures that subtype objects preserve properties of their supertypes. The properties are the formal specifications of the sub- and supertypes.

Liskov and Wing define an object's type not only by the object's legal values (as in traditional type checking) but also by its interface with environment (by means of the object's methods). Therefore, the properties of the subtype relations are both the supertype's values and its methods. Thus, altogether, the behavioral subtyping is defined to preserve the behavior of supertype methods and also all invariant properties of its supertype.

4.1 Subtype Relation

The formalization of the subtype relation is given in Figure 1. It uses the model of a type that is defined as a triple, $\langle O, V, M \rangle$, where O is a set of objects, V is a set of values for an object, and M is a set of methods that provide the means to manipulate the object. A *computation*, i.e., program execution, is a sequence of alternating states and transitions starting in some initial state. The formalization of types uses type invariants and constraints (for example, type σ can refer to its invariant I_σ and constraint, C_σ). Methods of each type are denoted by m (for example, methods of type τ are denoted as m_τ.

Subtyping is enforced by the invariant checking that is essentially established as an abstraction function. Additionally, a renaming map is defined. The rest of the section provides the formalization of this approach as presented in [19].

The subtype relation relates two types σ and τ. Each type's specifications preserve their invariants, I_σ and I_τ, and satisfy their constraints, C_σ and C_τ, respectively. In the rules, since x is an object of type σ, its value (x_{pre} or x_{post}) is a member of S (set of values of type σ) and therefore cannot be used directly in the predicates about objects of the supertype τ (which are in terms of values in T). Therefore, an abstraction function A is used to translate these values using the system predicates from subtype to supertype values. This approach requires that an abstraction function be defined for all legal values of the subtype (although it need not be defined for values that do not satisfy the subtype invariant). Moreover, it must map legal values of the subtype to legal values of the supertype.

The first clause (cf. Figure 1) addresses the need to relate inherited methods of the subtype to those of the supertype. The first two signature rules are the standard contra/covariance rules [4,3]. The exception rule says that m_σ may not throw more exceptions (the exceptions concept is taken from object-oriented programming) than m_τ, since a caller of a method on a supertype object should not expect to handle an unknown exception. The pre- and post-condition rules are the intuitive counterparts to the contravariant and covariant rules for method signatures. The pre-condition rule ensures the subtype's method can be called in any state required by the supertype. The post-condition rule says that the subtype method's post-condition can be stronger than the supertype method's post-condition; hence, any property that can be proved based on the supertype method's post-condition also follows from the subtype's method's post-condition.

The second clause addresses preserving program-independent properties. The invariant rule and the assumption that the type specification preserves the invariant suffices to argue that invariant properties of a supertype are preserved by the subtype. This approach does not include the invariant in the methods (or constraint) rule directly.

Definition of the subtype relation, $\preceq$: $\sigma = \langle O_\sigma, S, M \rangle$ is a *subtype* of $\tau = \langle O_\tau, T, N \rangle$ if $\exists A : S \rightarrow T$, and a renaming map, $R : M \rightarrow N$, such that:

1. Subtype methods preserve the supertype methods' behavior. If m_τ of τ is the corresponding renamed method m_σ of σ, the following rules must hold:

Signature Rule.

– *Contravariance of arguments.* m_τ and m_σ have the same number of arguments. If the list of argument types of m_τ is α and that of m_σ is β, then $\forall i. \alpha_i \preceq \beta_i$.
– *Covariance of result.* Either both m_τ and m_σ have a result or neither has. If there is a result, let m_τ's result type be α and and m_σ's be β. Then $\beta \preceq \alpha$.
– *Exception rule.* The exceptions thrown during execution of m_σ are contained in the set of exceptions thrown during execution of m_τ.

Methods Rule. For all $(x : \sigma)$, the following holds:

– *Pre-condition rule.* $m_\tau.pre[A(x_{pre})/x_{pre}] \Rightarrow m_\sigma.pre.$
– *Post-condition rule.* $m_\sigma.post \Rightarrow m_\tau.post[A(x_{pre})/x_{pre}, A(x_{post})/x_{post}]$

2. Subtypes preserve supertype properties. For all computations c, and all states ρ and ψ in c such that ρ precedes ψ, and for all $(x : \sigma)$, the following holds :

–*Invariant Rule.* Subtype invariants ensure supertype invariants. $I_\sigma \Rightarrow I_\tau[A(x_\rho)/x_\rho]$
–*Constraint Rule.* Subtype constraints ensure supertype constraints.
$C_\sigma \Rightarrow C_\tau[A(x_\rho)/x_\rho, [A(x_\psi)/x_\psi]$

Fig. 1. Definition of the subtype relation [19]

4.2 Pragmatics of the Subtype Relation Approach

The definition of the subtype relation by Liskov and Wing captures the intuition of programmers for designing type hierarchies in object-oriented languages. The major contribution is that it provides precise definitions to capture it. As a result systems become amenable to formal analysis of ensuring behavioral compatibility between super- and subtype objects. Liskov and Wing report a number of successful examples where the subtype relation was useful in validating several benchmarks.

5 Substitutability Check

Our own earlier work [5] gives an automated and compositional procedure to solve the substitutability problem in the context of evolving software systems. Checking substitutability is defined as verifying whether (i) any updated portion of software continues to provide all services provided by it earlier counterpart, and (ii) all previously established system correctness properties remain valid after the upgrades. A component is essentially a C program communicating with other components via blocking message passing. A component assembly consists of collection of such concurrent components. In the following, $\mathcal{I}$ denotes the set of indices of the upgraded components in the assembly.

The procedure consists of two phases, namely, containment and compatibility. The containment phase checks locally if any useful behavior has been lost during upgrade of a component in the assembly and relies on simultaneous use of over- and under-approximations of the evolved software component. The compatibility phase checks if the added behaviors of the upgraded component violate any global safety specifications. This phase uses a dynamic assume-guarantee reasoning algorithm, wherein previously generated assumptions before upgrades are reused efficiently to re-validate the new assembly. The framework uses iterative abstraction/refinement paradigm [2,8,17] for both containment and compatibility phases. This approach enabled extraction of relatively simple finite-state models from complex C code. State-event automata (finite automata with both state and edges labeled) are used to represent these abstractions. Moreover, simultaneous upgrade of multiple components are allowed in this framework.

5.1 Containment Check

The containment step verifies for each $i \in \mathcal{I}$, that $C_i \sqsubseteq C_i^{'}$, i.e., every behavior of C_i is also a behavior of $C_i^{'}$. If $C_i \not\sqsubseteq C_i^{'}$, we also generate a counterexample behavior in $Behv(C_i) \setminus Behv(C_i^{'})$ which will be subsequently provided as feedback. This containment check is performed iteratively and component-wise as depicted in Figure 2 (CE refers to the counterexample generated during the verification phase). For each $i \in \mathcal{I}$, the containment check proceeds as follows:

1. Abstraction. Construct finite models M and M' such that the following conditions **C1** and **C2** hold:

$$(\mathbf{C1})\ C_i \sqsubseteq M \qquad (\mathbf{C2})\ M' \sqsubseteq C_i^{'}$$

Here M is an *over-approximation* of C_i and can be constructed by standard predicate abstraction [15]. M' is constructed from $C_i^{'}$ via a modified predicate abstraction which produces an *under-approximation* of its input C component. We now describe the details of the abstraction steps.

Suppose that C_i comprises of a set of C statements $Stmt = \{st_1, \dots, st_k\}$. Let V be the set of variables in the C_i. A valuation of all the variables in a program corresponds to a concrete state of the given program. We denote it by $\bar{v}$.

Predicates are functions that map a concrete state $\bar{v} \in S$ into a Boolean value. Let $\mathcal{P} = \{\pi_1, \dots, \pi_k\}$ be the set of predicates over the given program. On evaluating the set of predicates in $\mathcal{P}$ in a particular concrete state $\bar{v}$, we obtain a vector of boolean values $\bar{b}$, where $b_i = \pi_i(\bar{v})$. The boolean vector $\bar{b}$ represents an abstract state and we denote this operation by an abstraction function α: $\bar{b} = \alpha(\bar{v})$.

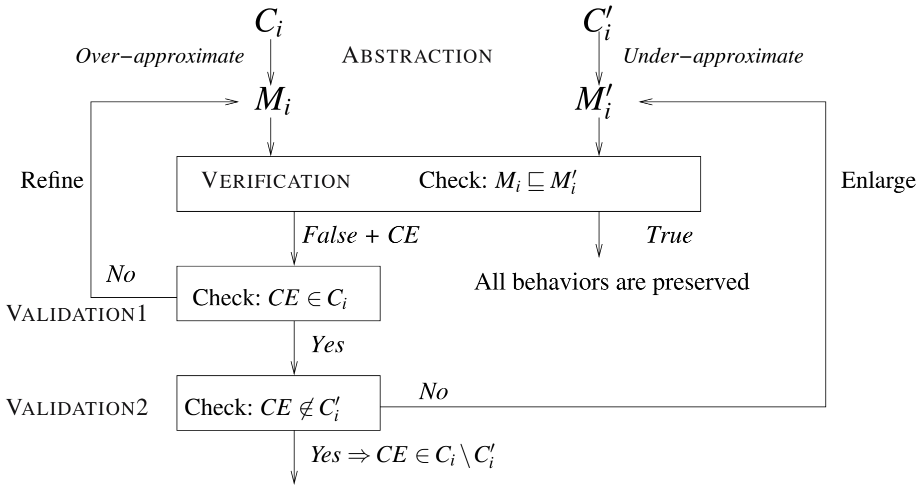

Fig. 2. The containment phase of the substitutability framework

May Predicate Abstraction: Over-approximation. This step corresponds to the standard predicate abstraction. Each statement (or basic block) St in C_i is associated with a transition relation $T(\bar{v}, \bar{v}')$. Here, $\bar{v}$ and $\bar{v}'$ represent a concrete state before and after execution of St, respectively. Given the set of predicates $\mathcal{P}$ and associated vector of Boolean variables $\bar{b}$ as before, we compute an abstract transition relation $\hat{T}(\bar{b}, \bar{b}')$ as follows:

$$\hat{T}(\bar{b}, \bar{b}') = \exists \bar{v}, \bar{v}' \; : \; T(\bar{v}, \bar{v}') \wedge \bar{b} = \alpha(\bar{v}) \wedge \bar{b}' = \alpha(\bar{v}') \tag{1}$$

$\hat{T}$ is an existential abstraction of T and is also referred to as its *may* abstraction $\hat{T}_{may}$ [24]. We compute this abstraction using the weakest precondition (WP) transformer [13,18] on predicates in $\mathcal{P}$ along with an automated theorem prover [15].

Must Predicate Abstraction: Under-approximation. The modified predicate abstraction constructs an under-approximation of the concrete system via universal or *must* abstraction. Given a statement St in the modified component $C_i^{'}$ and its associated transition relation $T(\bar{v}, \bar{v}')$ as before, we compute its must abstraction with respect to predicates $\mathcal{P}$ as follows:

$$\hat{T}(\bar{b}, \bar{b}') = \forall \bar{v}, \exists \bar{v}' \ : \ T(\bar{v}, \bar{v}') \wedge \bar{b} = \alpha(\bar{v}) \wedge \bar{b}' = \alpha(\bar{v}') \tag{2}$$

We use $\hat{T}_{must}$ to denote the above relation. Note that $\hat{T}_{must}$ contains a transition from an abstract state $\bar{b}$ to $\bar{b}'$ iff for every concrete state $\bar{v}$ corresponding to $\bar{b}$, there exists a concrete transition to a state $\bar{v}'$ corresponding to $\bar{b}'$ [24]. Further, it has been shown [24] that the concrete transition relation T simulates the abstract transition relation $\hat{T}_{must}$. Hence, $\hat{T}_{must}$ is an under-approximation of T. Again, we compute $\hat{T}_{must}$ using the WP transformer on the predicates together with a theorem prover. At the end of this phase, we obtain M as an over-approximation of C_i and M' as an under-approximation of $C_i^{'}$.

2. Verification. Verify if $M \sqsubseteq M'$ (or alternatively $M \setminus B \sqsubseteq M'$ if the upgrade involved some bug fix and the bug was defined as a SE automata B). If so then from **(C1)** and **(C2)** (cf. **Abstraction**) above we know that $C_i \sqsubseteq C_i^{'}$ and we terminate with success. Otherwise we obtain a counterexample CE.

3. Validation 1. Check if CE is a real behavior of C_i. To do this we first compute the set S of concrete states of C_i that are reachable by simulating CE on C_i. This is done via symbolic simulation and the result is a formula ϕ that represents S. Then CE is a real behavior of C_i iff $S \neq \emptyset$, i.e., iff ϕ is satisfiable. If CE is a real behavior of C_i, we proceed to the next step. Otherwise we refine model M (remove spurious CE) by constructing a new set of predicates $\mathcal{P}'$ and repeat from Step 2.

4. Validation 2. Check if CE is *not* a real behavior of $C_i^{'}$. To do this we first symbolically simulate CE on $C_i^{'}$ to compute the reachable set S' of concrete states of $C_i^{'}$. This is done as in the previous validation step and the result is again a formula ϕ that represents S'. Then CE is not a real behavior of $C_i^{'}$ iff $S' = \emptyset$, i.e., iff ϕ is unsatisfiable. If CE is not a real behavior of $C_i^{'}$, we know that $CE \in \mathit{Behv}(C_i) \setminus \mathit{Behv}(C_i^{'})$. We add CE to the feedback step and stop. Otherwise we enlarge M' (add CE) by constructing a new set of predicates $\mathcal{P}'$ and repeat from Step 2. This step is an antithesis of standard abstraction-refinement since it *adds* the valid behavior CE back to M'. However it is conceptually similar to standard abstraction-refinement and we omit its details in this article.

Figure 2 depicts the individual steps of this containment check. The check is either successful (all behaviors of C_i are verified to be present in $C_i^{'}$) or returns an actual diagnostic behavior CE as a feedback to the developers.

5.2 Compatibility Check

The compatibility check ensures that the upgraded system satisfies global safety specification. The check relies on an automated assume-guarantee reasoning procedure [9], where the key idea is to generate an environment assumption for a component automatically and then verify if the rest of the system satisfies the assumption. An algorithm for learning regular sets, L^* [1,23], is used to automatically generate these assumptions assisted by a modelchecker [7]. It is assumed that appropriate assumptions have been generated by performing automated A-G reasoning over the assembly before an upgrade occurs. Upon an upgrade, the compatibility check procedure reuses the previously generated assumptions and locally modifies them in order to re-validate the upgraded component assembly. Similar to the containment phase, this check is performed on finite-state state-event (SE) automaton abstractions from the C components.

Automated Assume-Guarantee Reasoning. Assume-guarantee (A-G) based reasoning [22] is a well-known compositional verification technique. The essential idea here is to model-check each component independently by making an assumption about its environment, and then discharge the assumption on the collection of the rest of the components. Given a set of component SE automata $M_1, \ldots, M_n$ obtained after abstraction and a specification SE automata φ, consider the following non-circular A-G rule (called AG-NC) for n components:

$$\frac{\begin{array}{c} M_1 \parallel A_1 \sqsubseteq \varphi \\ M_2 \parallel \cdots \parallel M_n \sqsubseteq A_1 \end{array}}{M_1 \parallel \cdots \parallel M_n \sqsubseteq \varphi}$$

In the above, A_1 is a deterministic SE automata representing the assumption about the environment under which M_1 is expected to operate correctly. The second premise is itself an instance of the top-level proof-obligation with $n-1$ component SE automata. Therefore, AG-NC can be recursively applied to the rest of the components so that every rule premise contains exactly one component automaton. The assumptions are generated using L^* together with a model checker for SE automata in an iterative fashion, in a manner similar to the technique proposed by Cobleigh et al. [9]. In order to show that a component satisfies a global property, the technique first iteratively learns an assumption automaton that must be satisfied by its environment components. However, this initial assumption may be too strong to hold on its environment. Therefore, the assumption is gradually weakened by model checking it alternately against the component and its environment, and using the counterexamples generated.

The compatibility check makes use of AG-NC in the above form to first generate $n-1$ assumptions and then perform re-validation of upgrades. This re-validation may involve modifying several previously generated assumptions. The compatibility check avoids re-generating all such assumptions from scratch by proposing a technique to effectively reuse the previous assumptions by re-validating them first. A dynamic L^* algorithm is proposed that first re-validates the previously stored set of samples with respect to the upgraded assembly and then continues to learn in the usual fashion. This gives rise to a dynamic procedure for A-G reasoning over component assemblies across upgrades, also called as dynamic A-G.

Compatibility check with Dynamic A-G. The central idea in the compatibility check algorithm is to use dynamic L^* for learning assumptions as opposed to the original L^* algorithm. This allows the check to fully reuse the previous verification results, and in particular, contributes to its locally efficient nature.

Suppose we have a component assembly $\mathcal{C}$ consisting of n components and a given index set $\mathcal{I}$, identifying the upgraded components. We assume that a set of $n-1$ assumptions are available from a compatibility check before the upgrade took place. Now, suppose that the component assembly goes through an upgrade and the behaviors of one or more SE automata M_i $(1 \leq i \leq n)$ change. Note that the previous compatibility check provides us with a set of assumptions A_j $(1 \leq j < n)$. The dynamic compatibility check procedure **DynamicCheck** learns new assumptions required for the verification of the upgraded assembly while reusing the previous set of assumptions A_j by first re-validating them, if necessary.

We present an overview of the algorithm **DynamicCheck** for two SE automata. The complete details of the generalization of the algorithm to an arbitrary collection of SE automata can be found in [5]. Suppose we have two old SE automata M_1, M_2 and a property SE automaton φ. We assume that we previously verified $M_1 \parallel M_2 \sqsubseteq \varphi$ using **DynamicCheck**. The algorithm **DynamicCheck** uses dynamic L^* to learn appropriate assumptions that can discharge the premises of AG-NC. In particular suppose that while trying to verify $M_1 \parallel M_2 \sqsubseteq \varphi$, **DynamicCheck** generated an assumption A, with an observation table $\mathcal{T}$.

Now suppose we have new versions M_1', M_2' for M_1, M_2 where at least one of the M_i is different from M_i'. **DynamicCheck** will now reuse $\mathcal{T}$ and invoke the dynamic L^* algorithm to automatically learn an assumption A' such that: (i) $M_1' \parallel A' \sqsubseteq \varphi$ and (ii) $M_2' \sqsubseteq A'$. More precisely, **DynamicCheck** proceeds iteratively as follows:

1. It checks if $M_1 = M_1'$. If this holds, then it follows from the definition of AG-NC that the corresponding assumption language remains the same. Therefore, the algorithm starts learning from the previous table $\mathcal{T}$ itself, i.e., it sets $\mathcal{T}' := \mathcal{T}$. Otherwise it re-validates $\mathcal{T}$ against M_1' to obtain a new table $\mathcal{T}'$.
2. The algorithm then derives a conjecture A' from $\mathcal{T}'$ and checks if $M_2' \sqsubseteq A'$. If this check passes, then the procedure terminates with TRUE and a new assumption A'. Otherwise, a counterexample CE is obtained.
3. The counterexample CE is analyzed to see if CE corresponds to a real counterexample to $M_1' \parallel M_2' \sqsubseteq \varphi$ (same as a membership query with M_1'). If so, the algorithm constructs such a counterexample and terminates with FALSE. Otherwise it updates $\mathcal{T}'$ using CE.
4. $\mathcal{T}'$ is closed by making membership queries and the algorithm repeats from Step 2.

5.3 Case Studies

The compatibility check phase for checking component substitutability was implemented in the COMFORT [16] framework. COMFORT extracts abstract component SE models from C programs using predicate abstraction and performs automated A-G reasoning on them. If the compatibility check returns a counterexample, the counterexample validation and abstraction-refinement modules of COMFORT are employed to check for spuriousness and perform refinement, if necessary. The evaluation benchmarks consist of an assembly having seven components, which implement an inter-process communication (IPC) protocol.

Both single and simultaneous upgrades of the *write-queue* and the *ipc-queue* components in the IPC assembly were used. The upgrades had both missing and extra behaviors as compared to the original system. A set of properties describing behaviors of the verified portion of the IPC protocol were used. It was observed that the compatibility check required much less time for re-validation (due to reuse of previously generated assumptions) as compared to time for compositional verification of the original system.

6 Comparative Analysis

We have presented four techniques each of which addresses a problem of behavioral consistency among programs. While the techniques address similar problems of the program compatibility, they differ greatly in the specification formalisms and algorithmic approaches. This makes it difficult to conduct comparative analysis among the techniques. To overcome this difficulty, we chose one of the techniques as a reference point against which we compared the other three approaches. Specifically, we compared the automata interface approach, the observation summary approach of McCamant and Ernst, and the behavioral subtyping technique to our own work on component substitutability.

6.1 Interface Automata Formalism

In the interface automata formalism, substitution check corresponds to a refinement check, which ensures that the newer component exhibits fewer outputs and accepts more inputs than the old component. Our approach, however, differentiates between the refinement and substitution checks. We believe that the refinement check is too strong to be used as a substitution check since it is not adequate to check substitution locally without taking into account the exact behaviors of the environment components.

Given two interface automata M and N, checking alternating refinement [10] between M and N ($N \preccurlyeq M$, cf. Section 2) is an effective way to *locally* check for substitution of M by N. However, this refinement check assumes that the environment components remain the same, i.e., they continue to stimulate all inputs of M and are capable of responding to no more than the present outputs of M. Note that in case of multiple component upgrades, it is possible that the new environment for the component is more *supportive*, i.e., on one hand, it does not stimulate all inputs of M and on the other it is able to respond to even more outputs from M. If the new environment is more supportive, then it is possible that $N \not\preccurlyeq M$ but is still substitutable. In other words, even though some inputs of N may be absent in M, M may still substitute N since the absent inputs are no longer stimulated by the new environment. Therefore a substitutability check must take account of the new environment precisely rather than identifying it on basis of input and output behaviors of the previous component M. These criteria becomes even more important if multiple components in an assembly are upgraded and as a consequence, the environment for several components changes simultaneously.

6.2 Observation-Based Compatibility of Upgrades

McCamant et al. [21] suggest a technique for checking compatibility of multi-component upgrades. They derive consistency criteria by focusing on input/output component behavior only and abstract away the temporal information. Even though they state that their abstractions are unsound in general, they report success in detecting important errors on GNU C library upgrades. In contrast, our work on component substitutability uses abstractions that preserve temporal information about component behavior and are always sound. Moreover, they need to recompute the external observational summaries for each upgrade from scratch while our compatibility check procedure is able to reuse the previous verification proofs to re-validate the upgraded system.

6.3 Behavioral Subtype Checking

Conceptually, the subtype relation-based approach is similar to our work not only in that it is based on establishing the behavioral consistency among system components, but also in that it handles changes among versions of programs. The subtype check approach handles mutable objects and allows subtypes to have more methods than their supertypes. The component substitutability approach allows removal and addition of behaviors to the upgraded component as compared to its earlier counterpart.

The subtype relation is established as an invariant check. It requires defining an abstraction function that is a restricted form of the simulation relation between the subtype and supertype objects. Our work, uses the language containment approach and thus is more expensive computationally. However, our framework allows checking general safety properties, while work of Liskov and Wing handles only a restricted set of safety properties.

References

1. Dana Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75(2), pages 87–106, November 1987.
2. T. Ball and S. Rajamani. Boolean programs: A model and process for software analysis. *TR-2000-14*, 2000.
3. A. Black, A. Hutchinson, N. Jul, E. Levy, and L. Carter. Distribution and abstract types in emerald. *IEEE TSE*, 13(1):65–76, 1987.
4. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
5. Sagar Chaki, Edmund Clarke Natasha Sharygina, and Nishant Sinha. Dynamic component substitutability analysis. In *Proc. of Conf. on Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 512–528. Springer Verlag, 2005.
6. Arindam Chakrabarti, Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Synchronous and bidirectional component interfaces. In *CAV*, pages 414–427, 2002.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.
8. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, July 2000.

9. J. M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, April 2003.
10. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *FSE*, 2001.
11. Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *EMSOFT*, pages 148–165, 2001.
12. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
13. Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
14. M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering (ICSE'99)*, pages 213–224, 1999.
15. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.
16. James Ivers and Natasha Sharygina. Overview of ComFoRT: A model checking reasoning framework. *CMU/SEI-2004-TN-018*, 2004.
17. Robert Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.
18. K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
19. B. Liskov and J. Wing. Behavioral subtyping using invariants and constraints. *Formal Methods for Distributed Processing, an Object Oriented Approach*, pages 254–280, 2001.
20. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. 1987.
21. Stephen McCamant and Michael D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP Conference*, Olso, Norway, 2004.
22. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*. Springer-Verlag New York, Inc., 1985.
23. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Information and Computation*, volume 103(2), pages 299–347, 1993.
24. Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for CTL. In *TACAS*, pages 546–560, 2004.
25. Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, ACM Press, 2002.