# Dynamic Component Substitutability Analysis ⋆

Sagar Chaki Edmund Clarke Natasha Sharygina Nishant Sinha
chaki|nys@sei.cmu.edu emc|natalie|nishants@cs.cmu.edu

Carnegie Mellon University

**Abstract.** This paper presents an *automated* and *compositional* procedure to solve the substitutability problem in the context of evolving software systems. Our solution contributes two techniques for checking correctness of software upgrades: 1) a technique based on simultaneous use of over and under approximations obtained via existential and universal abstractions; 2) a *dynamic* assume-guarantee reasoning algorithm – previously generated component assumptions are reused and altered on-the-fly to prove or disprove the global safety properties on the updated system. When upgrades are found to be non-substitutable our solution generates constructive feedback to developers showing how to improve the components. The substitutability approach has been implemented and validated in the ComFoRT model checking tool set and we report encouraging results on an industrial benchmark.

**Keywords:** Software Model Checking, Verification of Evolving Software, Learning Regular Sets, Assume/Guarantee Reasoning

## 1 Introduction

Software systems evolve throughout the product life-cycle. For example, any software module (or component) is inevitably transformed as designs take shape, requirements change, and bugs are discovered and fixed. In general such evolution results in the removal of previous behaviors from the component and addition of new ones. Since the behavior of the updated software component has no direct correlation to that of its older counterpart, substituting it directly can lead to two kinds of problems. First, the removal of behavior can lead to unavailability of previously provided services. Second, the addition of new behavior can lead to violation of global correctness properties that were previously being respected.

In this context, the *substitutability* problem has been defined [7] as the verification of the following two criteria: (i) any *updated portion* of a software system

⋆ This research was sponsored by the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547, the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and was conducted as part of the Predictable Assembly from Certifiable Components (PACC) project at the Software Engineering Institute (SEI). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, ONR, NRL, the U.S. Government or any other entity.

must continue to provide all *services* offered by its earlier counterpart, and (ii) previously established system *correctness properties* must remain valid for the new version of the software system.

Model checking can be used at each stage of a system's evolution to solve both the above problems. However, conventionally model checking is applied to the entire system after every update irrespective of the degree of modification involved. The amount of time and effort required to verify an entire system can be prohibitive and repeating the exercise after each (even minor) system update is therefore impractical. In this article we present an *automated* framework that *localizes* the necessary verification to only modified system components, and thereby reduces dramatically the effort to check substitutability after every system update. Note that our framework is general enough to handle changes in the environment since the environment can also be modeled as a component.

In our framework a component is essentially a C program communicating with other components via blocking message passing. An assembly is a collection of such concurrently executing and mutually interacting components. We will define the notion of a component's behavior precisely later but for now let us denote the set of behaviors of a component $C$ by $Behv(C)$. Given two components $C$ and $C'$ we will write $C \sqsubseteq C'$ to mean $Behv(C) \subseteq Behv(C')$.

Suppose we are given an assembly of components: $\mathcal{C} = \{C_1, \ldots, C_n\}$, and a safety property $\varphi$. Now suppose that *multiple* components in $\mathcal{C}$ are upgraded. In other words, consider an index set $\mathcal{I} \subseteq \{1, \ldots, n\}$ such that for each $i \in \mathcal{I}$ there is a *new* component $C_i^{'}$ to be used in place of its *old* version $C_i$. Our goal is to check the substitutability of $C_i^{'}$ for $C_i$ in $\mathcal{C}$ for every $i \in \mathcal{I}$ with respect to the property $\varphi$. This paper presents a framework that satisfies this goal by establishing the following two tasks:

**Containment.** Verify, for each $i \in \mathcal{I}$, that every behavior of $C_i$ is also a behavior of $C_i^{'}$, i.e., $C_i \sqsubseteq C_i^{'}$. If $C_i \not\sqsubseteq C_i^{'}$, we also construct a set $\mathcal{F}_i$ of behaviors in $Behv(C_i) \setminus Behv(C_i^{'})$ which will be subsequently used for feedback generation. Note that the upgrade may involve the removal of behaviors designated as errant, say $B$. In this case, we check $C_i \setminus B \sqsubseteq C_i^{'}$ since behaviors of $B$ will clearly be absent in $C_i^{'}$.

**Compatibility.** Let us denote by $\mathcal{C}'$ the assembly obtained from $\mathcal{C}$ by replacing the old component $C_i$ with its new version $C_i^{'}$ for each $i \in \mathcal{I}$. Since in general it is not the case that for each $i \in \mathcal{I}$, $C_i^{'} \sqsubseteq C_i$. Therefore, the new assembly $\mathcal{C}'$ may have more behaviors than the old assembly $\mathcal{C}$. Hence $\mathcal{C}'$ might violate $\varphi$ even though $\mathcal{C}$ did not. Thus, our second task is to verify that $\mathcal{C}'$ satisfies the safety property $\varphi$ (which would imply that the new components can be safely integrated).

Note that checking compatibility is non-trivial because it requires the verification of a concurrent system where multiple components might have been modified. Moreover, this task is complicated by the fact that our goal is to focus on the components that have been modified.

The component substitutability framework is defined by the following new algorithms: 1) a technique based on simultaneous use of over and under approximations obtained via existential and universal abstractions for the containment check of the substitutable components; 2) a *dynamic* assume-guarantee algorithm developed for the compatibility check. The algorithm is based on automata-theoretic learning for regular sets. It is dynamic in the sense that it learns appropriate environment assumptions for the new components by *reusing* the environment assumptions for their older versions.

The framework uses an iterative abstraction/refinement paradigm for both the containment and compatibility check procedures. The abstraction-based approach is essential since it not only enables the extraction of finite-state models from software programs but also reduces the complexity of software verification. Details of the abstraction procedure and the abstraction/refinement process are beyond the scope of this article and can be found in [4]. In summary, the developed component substitutability framework has several advantageous features:

- It allows *multiple* components to be upgraded simultaneously. This is crucial since modifications in different components often interact non-trivially to maintain overall system safety and integrity. Hence such modifications must be analyzed jointly.
- It identifies features of an old component which are absent in its updated version. It subsequently generates feedback to localize the modifications required to add the missing features back.
- It is completely automated and uses *dynamic* assume-guarantee style reasoning to scale to large software systems.
- It allows new components to have more behaviors than their old counterparts in order be replaceable. The *extra* behaviors are critical since they provide vendors with flexibility to implement new features into the product upgrades. Our framework verifies if these new behaviors do not violate previously established global specifications of a component assembly[1].

We employ state/event-based modeling techniques [5] to model and reason about both the data and communication aspects of software. In particular we use the state/event computational structures, called Doubly Labeled Automata (DLA) to model, as well as to specify, software systems. We have implemented the substitutability framework as part of the ComFoRT [6] reasoning framework, which is based on the C model checker MAGIC [4, 15]. We experimented with an industrial benchmark and report encouraging results in Section 7.

## 2 Related Work

Related projects often impose the restriction that every behavior of the new component must also be a behavior of the old component. In such a case the new component is said to refine the old component. For instance, de Alfaro et al. [11, 8] define a notion of interface automaton for modeling component interfaces and

[1] Verification of these new features remains a responsibility of designers of the upgraded systems.

show compatibility between components via refinement and consistency between interfaces. However, automated techniques for constructing interface automata from component implementations are not presented. In contrast, our approach automatically extracts conservative DLA models (which are similar to finite state interface automata) from component implementations. Moreover, we do not require refinement among the old components and their new versions.

Ernst et al. [16] 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. In contrast, our abstractions preserve temporal information about component behavior and are always sound. They also use a refinement-based notion on the generated consistency criteria for showing compatibility.

The application of learning is extremely useful from a pragmatic point of view since it is amenable to complete automation, and is gaining rapid popularity [14] in formal verification. The use of learning for automated assume-guarantee reasoning was proposed originally by Cobleigh et al. [10]. The use of learning along with predicate abstraction has also been applied in the context of interface synthesis [1] and various types of assume-guarantee proof rules for automated software verification [3].

This work is related to our earlier project [7] that solves the component substitutability problem in the context of verifying *individual* component upgrades. A major improvement of the current work is that it is aimed at verifying the component substitutability in the presence of *simultaneous upgrades of multiple components*. Another distinction of this work is that it provides an innovative *dynamic* assume-guarantee reasoning framework for the compatibility check. The dynamic nature of the compatibility check allows reusing previously computed assumptions to prove or disprove the global properties of the updated system.

Additionally, this paper gives a new solution to the containment check problem presented in [7]. In our earlier work, the containment step is solved using learning techniques for regular sets and handles finite-state systems only. In contrast, the new approach is extended to handle infinite-state C programs. Moreover, this paper defines a new technique based on simultaneous use of over and under approximations obtained via existential and universal abstractions.

## 3 Background and Notation

Let $\bullet$ denote the concatenation operator over sequences and $X^*$ denote zero or more applications of $\bullet$ over $X$ as usual. For any two sets $X$ and $Y$ we will denote the set $\{x \bullet y \mid x \in X \wedge y \in Y\}$ by $X \bullet Y$.

**Definition 1 (Words and Traces).** *Given an alphabet $\Sigma$ and a set of atomic propositions $AP$ we often say that $(\Sigma, AP)$ is a state/event (SE) alphabet. For an SE alphabet $\widehat{\Sigma} = (\Sigma, AP)$, the set of words over $\widehat{\Sigma}$ is denoted by $Word(\widehat{\Sigma})$*

*and defined as* $Word(\widehat{\Sigma}) = (\Sigma \bullet 2^{AP})^*$. *The set of traces over* $\widehat{\Sigma}$ *is denoted by* $Trace(\widehat{\Sigma})$ *and defined as* $Trace(\widehat{\Sigma}) = 2^{AP} \bullet Word(\widehat{\Sigma})$.

Thus a word or a trace is an alternating sequence of subsets of $AP$ and elements of $\Sigma$. However a word always begins with an action and ends with a set of propositions and can be empty. In contrast, a trace begins and ends with a set of propositions and cannot be empty.

**Definition 2 (Doubly Labeled Automaton).** *A doubly labeled automaton (DLA) is a 7-tuple* $(S, Init, AP, \mathcal{L}, \Sigma, \delta, F)$ *such that: (i)* $S$ *is a finite set of* states, *(ii)* $Init \subseteq S$ *is a set of initial states, (iii)* $AP$ *a finite set of (atomic) state propositions, (iv)* $\mathcal{L} : S \to 2^{AP}$ *a state-labeling function, (v)* $\Sigma$ *a finite set of events or actions (alphabet), (vi)* $\delta \subseteq S \times \Sigma \times S$ *a transition relation, and (vii)* $F \subseteq S$ *is a set of final or accepting states.*

For any DLA with transition relation $\delta$ we write $q \xrightarrow{\alpha} q'$ to mean $q' \in \delta(q, \alpha)$. A DLA is said to be deterministic (DDLA) iff for any $q \in S$, $\alpha \in \Sigma$ and $p \subseteq AP$ there is at most one $q' \in S$ such that $q \xrightarrow{\alpha} q'$ and $\mathcal{L}(q') = p$. DLAs are not more expressive than standard finite automata since propositional labelings can always be rewritten in terms of actions [9]. However, we choose to use the DLA formalism for the sake of simplicity since it captures the essence of the state/event-based notation.

**Definition 3 (Language).** *Let* $M = (S, Init, AP, \mathcal{L}, \Sigma, \delta, F)$ *be a DLA and* $\widehat{\Sigma} = (\Sigma, AP)$. *A trace* $t \in Trace(\widehat{\Sigma})$ *is accepted by* $M$ *iff* $t = p_1, \alpha_1, p_2, \ldots, \alpha_{n-1}, p_n$ *and there exists a sequence* $s_1, s_2, \ldots, s_n$ *of states of* $M$ *such that: (i)* $s_1 \in Init$, *(ii)* $s_n \in F$, *(iii) for* $1 \leq i \leq n$, $\mathcal{L}(s_i) = p_i$, *and (iii) for* $1 \leq i < n$, $s_i \xrightarrow{\alpha_i} s_{i+1}$. *The language of* $M$ *is denoted by* $\mathbb{L}(M)$ *and defined as the set of all traces accepted by* $M$.

A language is said to be regular iff it is accepted by some DLA. The set of regular languages is closed under union, intersection and complementation. DDLAs are equivalent to DLAs as far as language acceptance is concerned. In other words for any regular language $L$ there is a DDLA $M$ such that $\mathbb{L}(M) = L$. Also every regular language $L$ is accepted by a unique (up to isomorphism) minimal DDLA.

**Definition 4 (Abstraction).** *Given two DLAs* $M_1$ *and* $M_2$ *we say that* $M_2$ *is an abstraction of* $M_1$, *denoted by* $M_1 \sqsubseteq M_2$, *iff* $\mathbb{L}(M_1) \subseteq \mathbb{L}(M_2)$.

**Definition 5 (Parallel Composition).** *Let* $M_1 = (S_1, Init_1, AP_1, \mathcal{L}_1, \Sigma_1, \delta_1, F_1)$ *and* $M_2 = (S_2, Init_2, AP_2, \mathcal{L}_2, \Sigma_2, \delta_2, F_2)$ *be two DLAs. The parallel composition of* $M_1$ *and* $M_2$, *denoted by* $M_1 \parallel M_2$, *is the DLA* $(S_1 \times S_2, Init_1 \times Init_2, AP_1 \cup AP_2, \mathcal{L}, \Sigma_1 \cup \Sigma_2, \delta, F_1 \times F_2)$, *where: (i)* $\mathcal{L}(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$, *and (ii)* $\delta$ *is such that* $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ *iff:*

$$\forall i \in \{1, 2\} \centerdot (\alpha \notin \Sigma_i \wedge s_i = s_i') \bigvee (\alpha \in \Sigma_i \wedge s_i \xrightarrow{\alpha} s_i')$$

In other words, DLAs must synchronize on shared actions and proceed independently on local actions. This notion of parallel composition is derived from CSP [19].

**Definition 6 (Weakest Assumption).** *For any DLA $M$, and any safety property, expressed as a DLA $\varphi$, there exists a weakest (w.r.t. the $\sqsubseteq$ preorder) DLA $WA$ with the following property: for any DLA $E$, $M \parallel E \sqsubseteq \varphi$ iff $E \sqsubseteq WA$ [12]. In fact it can be shown that $WA$ is a DLA accepting the language $\mathbb{L}(\overline{M \parallel \overline{\varphi}})$.*

## 4 Containment

Recall that in the containment step we verify 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 construct a set $\mathcal{F}_i$ of behaviors in $Behv(C_i) \setminus Behv(C_i^{'})$ which will be subsequently used for feedback generation. This containment check is performed iteratively and component-wise as depicted in Figure 1 ($CE$ refers to the counterexample generated during the verification phase). For each $i \in \mathcal{I}$, the containment check proceeds as follows:

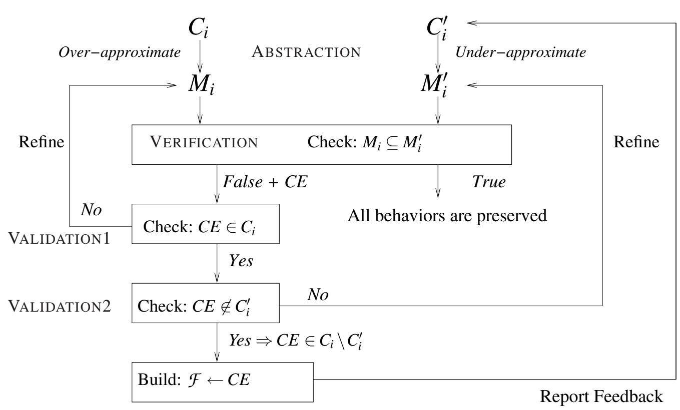


**Fig. 1.** The containment phase of the substitutability framework.

**1. Abstraction.** Construct finite models $M$ and $M'$ such that **(C1)** $C_i \sqsubseteq M$ and **(C2)** $M' \sqsubseteq C_i^{'}$. Note that $M$ is an *over-approximation* of $C_i$ and can be constructed by standard predicate abstraction [13]. However $M'$ is constructed from $C_i^{'}$ via a modified predicate abstraction which produces an *under-approximation* of its input C component. We give an overview of predicate abstraction and then the modified predicate abstraction. Complete details of our predicate abstraction procedure can be found elsewhere [4].

**Predicates and Valuations.** Suppose we are given a set of predicates (pure C expressions) $\mathcal{P}$. Each valuation $\mathcal{V}$ of $\mathcal{P}$ is simply a mapping from $\mathcal{P}$ to $\{0,1\}$. Thus if $\mathcal{P} = \{x < 1, y \geq 0\}$ then the set of valuations of $\mathcal{P}$ is $\{(0,0),(0,1),(1,0),(1,1)\}$. Let $\mathcal{P} = \{p_1,\ldots,p_n\}$ and $\mathcal{V}$ be a valuation of $\mathcal{P}$. Then the concretization of $\mathcal{V}$ is denoted by $\gamma(\mathcal{V})$ and defined as: $\gamma(\mathcal{V}) \equiv \bigwedge_{i=1}^{n} X_i$ where $X_i = p_i$ iff $\mathcal{V}(p_i) = 1$ and $\neg p_i$ otherwise. For example consider $\mathcal{P} = \{x < 1, y \geq 0\}$ and $\mathcal{V} = (0,1)$. Then $\gamma(\mathcal{V}) = \neg(x < 1) \wedge (y \geq 0)$.

**Predicate Abstraction.** Suppose that $C_i$ comprises of a set of C statements $\mathit{Stmt} = \{st_1,\ldots,st_k\}$. Without loss of generality we assume that each statement of $C_i$ is either an assignment, an `if-then-else` or a `goto`. Also we are given a set of predicates $\mathcal{P}$ with set of valuations $\mathit{Val}$. The general idea behind predicate abstraction is to represent a set of concrete states symbolically using a formula. Thus the predicate abstraction $C_i$ w.r.t. $\mathcal{P}$ is an DLA $M$ whose set of states $= \mathit{Stmt} \times \mathit{Val}$. Intuitively each state $s = (st, \mathcal{V})$ of $M$ represents the set of all concrete execution states $c$ of $C_i$ such that $st$ is the next statement to be executed at $c$ and the expression $\gamma(\mathcal{V})$ is satisfied by the memory configuration at $c$. In such a case we often say $c \in s$ to highlight the fact that each state of $M$ can be thought of as a set of concrete execution states of $C_i$.

The transitions of $M$ are defined such that $M$ is an over-approximation of $C_i$ via *existential* abstraction. For example, let $s_1 = (st_1, \mathcal{V}_1)$ and $s_2 = (st_2, \mathcal{V}_2)$ be two states of $M$ such that $st_1$ is an assignment. Then $M$ contains a transition from $s_1$ to $s_2$ if there is a transition from *some* concrete state $c_1 \in s_1$ to some concrete state $c_2 \in s_2$. It turns out that this is equivalent to: (i) $st_2$ being the next statement to be executed after $st_1$, and (ii) the formula $WP\{\gamma(\mathcal{V}_2)\}[st_1] \wedge \gamma(\mathcal{V}_1)$ being satisfiable where $WP\{\gamma(\mathcal{V}_2)\}[st_1]$ denotes the weakest precondition of $\gamma(\mathcal{V}_2)$ w.r.t. $st_1$. Other kinds of statements are handled analogously.

**Modified Predicate Abstraction.** In contrast our modified predicate abstraction constructs an under-approximation of the concrete system via *universal* abstraction. More precisely suppose $C_i^{'}$ comprises of a set of C statements $\mathit{Stmt}'$ and we are given a set of predicates $\mathcal{P}'$ with set of valuations $\mathit{Val}'$. Then the modified predicate abstraction of $C_i^{'}$ w.r.t. $\mathcal{P}'$ is an DLA $M'$ whose set of states $= \mathit{Stmt}' \times \mathit{Val}'$. The correspondence between the states of $M'$ and the execution states of $C_i^{'}$ is exactly as in the case of predicate abstraction. The difference is in the way the transitions of $M'$ are defined. More precisely, let $s_1 = (st_1, \mathcal{V}_1)$ and $s_2 = (st_2, \mathcal{V}_2)$ be two states of $M'$ such that $st_1$ is an assignment. Then $M'$ contains a transition from $s_1$ to $s_2$ if there is a transition from *every* concrete state $c_1 \in s_1$ to some concrete state $c_2 \in s_2$. This is equivalent to: (i) $st_2$ being the next statement to be executed after $st_1$, and (ii) the formula $\gamma(\mathcal{V}_1) \implies WP\{\gamma(\mathcal{V}_2)\}[st_1]$ being valid. Other kinds of statements are handled analogously. The satisfiability and validity of formulas are checked using an automated theorem prover.

**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 DLA $B$). If so then from

**(C1)** and **(C2)** 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 can simulate $CE$. This is done via symbolic simulation and the result is a formula $\phi$ that represents $S$. Then $CE$ is a real behavior of $C_i$ iff $S \neq \emptyset$, i.e., iff $\phi$ is satisfiable. If $CE$ is a real behavior of $C_i$, we proceed to the next step. Otherwise we refine model $M$ by constructing a new set of predicates $\mathcal{P}$ and repeat from Step 2. The refinement step is done according to the procedure implemented in the MAGIC [4] tool.

**4. Validation 2.** Check if $CE$ is *not* a real behavior of $C_i^{'}$. To do this we first compute the set $S'$ of concrete states of $C_i^{'}$ that can simulate $CE$. This is done as above and the result is again a formula $\phi$ that represents $S'$. Then $CE$ is not a real behavior of $C_i^{'}$ iff $S' = \emptyset$, i.e., iff $\phi$ is unsatisfiable. If $CE$ is not a real behavior of $C_i^{'}$, we know that $CE \in Behv(C_i) \setminus Behv(C_i^{'})$. We add $CE$ to $\mathcal{F}_i$ and stop. Otherwise we refine $M'$ by constructing a new set of predicates $\mathcal{P}'$ and repeat from Step 2. This refinement 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.

Note that the above process terminates as soon as it adds a single behavior to $\mathcal{F}_i$. However it can be extended to generate a set of behaviors in $\mathcal{F}_i$ as follows. First a set of counterexamples $\widehat{CE}$ is constructed in Step 2. Then each element of $\widehat{CE}$ is processed via Steps 3 and 4 and every counterexample which belongs to $C_i$ but not to $C_i^{'}$ is added to $\mathcal{F}_i$. The use of $\mathcal{F}_i$ to provide feedback to developers showing how to correct the updated components is discussed in Section 6.

## 5 Compatibility

Recall that the compatibility check is aimed at ensuring that the upgraded system satisfies global safety specifications. Our compatibility check procedure involves two key paradigms - *dynamic* regular set learning and assume guarantee reasoning. We first present these two techniques and then describe their use in our overall compatibility algorithm.

### 5.1 Dynamic Regular Set Learning

Central to our compatibility check procedure is a new *dynamic* algorithm to learn regular languages. Our algorithm is based on the $L^*$ algorithm developed by Angluin [2]. The compatibility check uses a state/event version of the $L^*$ that is a straight forward extension of the original algorithm (for simplicity we will refer to both as $L^*$). The detailed description of the state/event $L^*$ algorithm and the proof of its correctness and complexity analysis can be found in [20]. We will first present the state/event learning algorithm and then describe a *dynamic* version of it that we actually use for checking compatibility. We will denote the symmetric difference of two sets $X$ and $Y$ by $X \oplus Y$, i.e, $\rho \in X \oplus Y$ iff $\rho \in X \setminus Y$ or $\rho \in Y \setminus X$.

**The $L^*$ Algorithm.** Let $U$ be an unknown regular language over some SE alphabet $\widehat{\Sigma} = (\Sigma, AP)$. In order to learn $U$, $L^*$ interacts with a *minimally adequate teacher* $MAT$ for $U$, which can provide Boolean answers the following two kinds of queries:

1. *Membership*. Given a $\rho \in \mathit{Trace}(\widehat{\Sigma})$, $MAT$ returns TRUE iff $\rho \in U$.
2. *Candidate*. Given a DDLA $D$, $MAT$ returns TRUE iff $\mathbb{L}(D) = U$. If $MAT$ returns FALSE, it also returns a counterexample trace $w \in \mathbb{L}(D) \oplus U$.

Given an unknown regular language $U \subseteq \mathit{Trace}(\widehat{\Sigma})$ and a $MAT$ for $U$, the $L^*$ algorithm *iteratively* constructs a minimal DDLA $D$ such that $L(D) = U$. It maintains an observation table $(S, E, T)$ where: (i) $S$ is a prefix-closed set over $\mathit{Trace}(\widehat{\Sigma})$ labeling the rows of the table, (ii) $E$ a suffix-closed set over $\mathit{Word}(\widehat{\Sigma})$ labeling the columns of the table, and (iii) $T : (S \cup S \bullet \widehat{\Sigma}) \times E \to \{0, 1\}$ is the valuation of the table entries such that:

$$\forall s \in S \cup S \bullet \widehat{\Sigma} \centerdot \forall e \in E \centerdot T[s, e] = 1 \iff s \bullet e \in U$$

Additionally, for any $s \in S \cup S \bullet \widehat{\Sigma}$, let us define a function $r_s$ as follows:

$$\forall e \in E \centerdot r_s(e) = T[s, e]$$

Given a trace $t \in \mathit{Trace}(\widehat{\Sigma})$ we write $\mathit{Last}(t)$ to mean the last set of propositions in $t$. $L^*$ always ensures that the following invariant holds on the table: for any two distinct $s_1, s_2 \in S$ either $r_{s_1} \neq r_{s_2}$ or $\mathit{Last}(s_1) \neq \mathit{Last}(s_2)$. The table is said to be *closed* if for every $t \in S \bullet \widehat{\Sigma}$, there exist an $s \in S$ such that $r_s = r_t$ and $\mathit{Last}(s) = \mathit{Last}(t)$.

Let us denote the empty word by $\lambda$. Then $L^*$ starts with a table $(S, E, T)$ such that $S = 2^{AP}$, $E = \{\lambda\}$ and in each iteration proceeds as follows. It first updates the table using membership queries till it is closed. Next $L^*$ builds a candidate DDLA $D$ from the table and makes a candidate query with $D$. If the $MAT$ returns TRUE to the candidate query, $L^*$ returns $D$ and stops. Otherwise, $L^*$ updates $E$ with a single word (constructed from the $CE$ returned by the candidate query) and proceeds with the next iteration. The complexity of $L^*$ is expressed by the following theorem [2, 20].

**Theorem 1.** *If $n$ is the number of states of the minimum DDLA accepting $U$ and $m$ is the upper bound on the length of any counterexample provided by the $MAT$, then the total running time of $L^*$ is bounded by a polynomial in $m$ and $n$. Moreover, the observation table is of size $O(m^2n^2 + mn^3)$.*

**Dynamic $L^*$.** Normally $L^*$ initializes with: $S = 2^{AP}$ and $E = \{\lambda\}$. This can be a drawback in cases where a previously learned candidate (and hence a table) exists and we wish to restart learning using information from the previous table. In the following, we show (Theorem 2) that if $L^*$ begins with any non-empty *valid* table then it must terminate with the correct result. In particular, this allows us to perform our compatibility check dynamically by restarting $L^*$ with any previously computed table by *re-validating* it instead of starting from an empty table[2].

**Definition 7 (Agreement).** *An observation table $(S, E, T)$ is said to agree with a regular language $U$ iff: $\forall(s, e) \in (S \cup S \bullet \widehat{\Sigma}) \times E$, $T(s, e) = 1$ iff $s \bullet e \in U$. Also, $(S, E, T)$ agrees with a candidate DDLA $D$ if it agrees with $\mathbb{L}(D)$.*

**Definition 8 (Validity).** *An observation table $\mathcal{T} = (S, E, T)$ is said to be valid for a language $U$ iff $(S, E, T)$ agrees with $U$. We say that a candidate derived from a closed table $\mathcal{T}$ is valid if $\mathcal{T}$ is valid.*

**Theorem 2.** *$L^*$ terminates with a correct result for any unknown language $U$ starting from any valid table for $U$.*

*Proof.* Let $n$ be the number of states in the minimal DDLA $M_U$ such that $\mathbb{L}(M_U) = U$. Note that both Theorem 1 and Lemma 5 from Angluin's correctness proof for $L^*$ [2] hold for valid and closed tables and candidates consistent with them. It follows from Theorem 1 and Lemma 5 that $L^*$ can always make a valid table closed and hence is able to construct a candidate, say $D$, with at most $n$ states. We now show that every subsequent candidate must have at least one more state than $D$.

A candidate query with $D$ either returns TRUE or a counterexample $CE \in \mathbb{L}(D) \oplus U$. Note that the table must agree with $D$ since $D$ is consistent with it. Also since the table is valid, it must agree with $U$. Therefore, $CE \notin (S \cup S \bullet \widehat{\Sigma}) \bullet E$ and will be added to $S$. Again, a valid and closed table $(S', E', T')$ must be obtained eventually after adding $CE$. Let $D'$ be the corresponding candidate.

Now, $D'$ is consistent with $T$ since $T'$ extends $T$. Also $D'$ agrees with $M_U$ as far accepting $CE$ is concerned while $D$ does not. Hence $D'$ is inequivalent to $D$ and by Theorem 1 in Angluin's proof, must have at least one more state than $D$. Hence, starting from $D$, $L^*$ can make at most $n - 1$ incorrect candidates, since the number of states is initially at least one, always increases monotonically and may not exceed $n - 1$. Since $L^*$ must keep making new candidates as long as it is running, it must terminate with a correct candidate $M_U$. □

Suppose we have a table $\mathcal{T}$ which is valid for an unknown language $U$ and we have a new unknown language $U'$ different from $U$. Suppose we want to learn $U'$ by starting $L^*$ with table $\mathcal{T}$. Note that in general $\mathcal{T}$ will not be valid for $U'$ and hence starting from $\mathcal{T}$ will not be appropriate. However, we can first *validate* $\mathcal{T}$ against $U'$ and then start $L^*$ from the validated $\mathcal{T}$. Theorem 2 provides the key insight behind the correctness of this procedure. As we shall see, this idea forms the backbone of our dynamic compatibility check procedure (cf. Section 5.3).

---

[2] A similar idea was also proposed in the context of *adaptive* model checking [14].

### 5.2 Assume-Guarantee Reasoning

Along with dynamic $L^*$, we also use assume-guarantee style compositional reasoning to check compatibility. Given a set of component DLAs $M_1, \ldots, M_n$ and a specification DLA $\varphi$, the following non-circular rule **AG** [17] can be used to verify $M_1 \parallel \cdots \parallel M_n \sqsubseteq \varphi$:

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

In the above, $A_1$ is an DLA representing the assumption about the environment under which $M_1$ is expected to operate correctly. As also observed by Cobleigh et al. [10], the second premise is itself an instance of the top-level proof-obligation with $n-1$ component DLAs. Hence, **AG** can be applied to decompose it further.

### 5.3 Compatibility Check for C Components

The procedure for checking compatibility of new components in the context of the original component assembly is presented in Figure 2. Given an old component assembly $\mathcal{C} = \{C_1, \ldots, C_n\}$, and a set of new components $\mathcal{C}' = \{C'_i \mid i \in \mathcal{I}\}$ (where $\mathcal{I} \subseteq \{1, \ldots, n\}$), it checks if a safety property $\varphi$ holds in the new assembly. We first present an overview of the compatibility procedure and then discuss its implementation in detail. The procedure uses a **DynamicCheck** algorithm, and is done in an iterative abstraction refinement style as follows:

1. Use predicate abstraction to obtain finite DLA models $M_i$, where $M_i$ is constructed from $C_i$ if $i \notin \mathcal{I}$ and from $C'_i$ if $i \in \mathcal{I}$. The abstraction is carried out component-wise. Let $\mathcal{M} = \{M_1, \ldots, M_n\}$.
2. Apply **DynamicCheck** on $\mathcal{M}$. If the result is TRUE the compatibility check terminates successfully. Otherwise we obtain a counterexample *CE*.
3. Check if *CE* is a valid counterexample. Once again this is done component-wise. If *CE* is valid, the compatibility check terminates unsuccessfully with *CE* as counterexample. Otherwise we go to the next step.
4. Refine a specific model, say $M_k$, such that the spurious *CE* is eliminated. Repeat from Step 2.

**Overview of DynamicCheck.** We first present an overview of the algorithm for two DLAs and then generalize it to an arbitrary collection of DLAs. Suppose we have two old DLAs $M_1, M_2$ and a property DLA $\varphi$. We assume that we previously tried to verify $M_1 \parallel M_2 \sqsubseteq \varphi$ using **DynamicCheck**. The algorithm **DynamicCheck** uses dynamic $L^*$ to learn appropriate assumptions that can discharge the premises of **AG**. In particular suppose that while trying to verify $M_1 \parallel M_2 \sqsubseteq \varphi$, **DynamicCheck** had constructed an observation table $\mathcal{T}$.

Now suppose we have new versions $M'_1, M'_2$ for $M_1, M_2$. Note than in general it could be that either $M'_1$ or $M'_2$ is identical to its old version. **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:

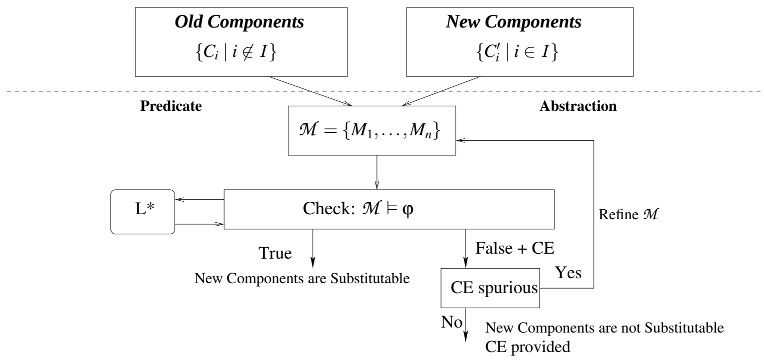


**Fig. 2.** The compatibility phase of the substitutability framework.

1. It checks if $M_1 = M'_1$. If so, it starts learning from the previous table $\mathcal{T}$, 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. It derives a conjecture $A'$ from $\mathcal{T}'$ and checks if $M'_2 \sqsubseteq A'$. If this check passes it terminates with TRUE and the new assumption $A'$. Otherwise it obtains a counterexample $CE$.
3. It analyzes $CE$ to see if $CE$ corresponds to a real counterexample to $M'_1 \parallel M'_2 \sqsubseteq \varphi$. If so, it constructs such a counterexample and terminates with FALSE. Otherwise it updates $\mathcal{T}'$ using $CE$.
4. It makes $\mathcal{T}'$ closed by making membership queries and repeats from Step 2.

**Generalized DynamicCheck.** We first describe the key ideas that enable us to reuse the previous assumptions and then present the complete **DynamicCheck** algorithm for multiple DLAs. Note that due to its dynamic nature, the algorithm will be able to *locally* identify the set of assumptions that need to be modified to re-validate the system.

**Incremental Changes between Successive Assumptions.** Recall that the $L^*$ algorithm maintains an observation table $(S, E, T)$ corresponding to an assumption $A$ for every component $M$. During an initial compatibility check, this table stores the information about membership of the current set of traces in an unknown language $U$ (i.e., the language of the *weakest assumption* for $M$). Upgrading the component $M$ modifies this unknown language for the corresponding assumption from $U$ to say, $U'$. Therefore, checking compatibility after an upgrade requires that the learner must compute a new assumption $A'$ corresponding to $U'$. In most cases, the languages $L(A)$ and $L(A')$ may *differ only slightly* and hence the information about behaviors of $A$ is *reused* in computing $A'$.

**Table Re-validation.** The original $L^*$ algorithm computes $A'$ starting from an empty table. However, as mentioned before, a more efficient algorithm would intend to reuse the previously inferred set of elements of $S$ and $E$ to learn $A'$. The result in Section 5.1 (Theorem 2) precisely enables the $L^*$ algorithm to achieve this goal. In particular, since $L^*$ terminates starting from any *valid* table, the assumption learner first obtains a valid table by reusing words in $S$ and $E$: update $T$ by asking membership queries w.r.t. $U'$ for each $\rho \in (S \cup S \bullet \widehat{\Sigma}) \bullet E$. The valid table $(S, E, T')$ hence obtained is subsequently made closed and then learning proceeds in the normal fashion. This allows the compatibility check to restart from any previous set of assumptions by *re-validating* them. The **GenerateAssumption** module implements this feature.

**Overall DynamicCheck Procedure.** The **DynamicCheck** procedure instantiates the **AG** rule for $n$ components and enables checking multiple upgrades simultaneously by reusing previous assumptions and verification results. In the description, we denote the previous and the new versions of a component DLA by $M$ and $M'$ and the previous and the new versions of a component assemblies by $\mathcal{M}$ and $\mathcal{M}'$ respectively. For ease of description, we always use a property, $\varphi$, to denote the right hand side of the top-level proof obligation of the compositional rule. We denote the modified property[3] at each recursion level of the algorithm by $\varphi'$. The old and new assumptions are denoted by $A$ and $A'$ respectively.

Figure 3 presents the pseudo-code of the algorithm **DynamicCheck** to perform the compatibility check. Lines (1-4) describe the case when $\mathcal{M}$ contains only one component. In Line 5, an assumption $A'$ corresponding to $M'$ and $\varphi'$ is generated using dynamic $L^*$ such that $M' \parallel A' \sqsubseteq \varphi'$. Lines (6-8) describe recursive invocation of **DynamicCheck** on $\mathcal{M} \setminus M$ against property $A'$. Finally, lines (9-15) show how the algorithm detects a counterexample *CE* and updates $A'$ with it or terminates with a TRUE/FALSE result. The salient features of this algorithm are the following:

- **GenerateAssumption** (line 5) does not generate new assumptions every time **DynamicCheck** is invoked. Instead, it reuses (by re-validating if necessary) the assumption $A$ computed in the previous compatibility check. When *CE* is used to update $A$, **GenerateAssumption** (line 12) does not need to re-validate $A$ since it must be validated previously.
- Verification checks are repeated on a component $M'$ (or a collection of components $\mathcal{M}' \setminus M'$) only if it is (they are) found to be different from the previous version $M$ ($\mathcal{M} \setminus M$) or if the corresponding property $\varphi$ has changed

[3] Note that under the recursive application of the compatibility check procedure the updated property $\varphi'$ corresponds to an assumption from the previous recursion level

**DynamicCheck** ($\mathcal{M}'$, $\varphi'$) **returns** counterexample or TRUE

1: **let** $M'$ = *first* element of $\mathcal{M}'$;
2: **if** ($\mathcal{M}' = \{M'\}$)
3:     **if** ($M \neq M'$ or $\varphi \neq \varphi'$) **return** ($M' \sqsubseteq \varphi'$);
4:     **else return** $M \sqsubseteq \varphi$;
5: $A'$ := **GenerateAssumption**($M'$, $\varphi'$);
6: **if** ($A \neq A'$ or $\mathcal{M} \setminus M \neq \mathcal{M}' \setminus M'$ )
7:     $CE$ := **DynamicCheck**($\mathcal{M}' \setminus M'$, $A'$);
8: **else** $CE$ := **DynamicCheck**($\mathcal{M} \setminus M$, $A$);
9: **while**($CE$ is non-empty)
10:     **if** ($M' \parallel CE \sqsubseteq \varphi'$)
11:         $A'$ := **UpdateAssumption** ($A'$,$CE$);
12:         $A'$ := **GenerateAssumption** ($M'$, $\varphi'$);
13:         $CE$ = **DynamicCheck** ($\mathcal{M}' \setminus M'$, $A'$);
14:     **else return** a witness counterexample $CE$ to $M' \parallel CE \not\sqsubseteq \varphi'$;
15: **return** TRUE;

**Fig. 3.** Pseudo-code for efficient compatibility check

(lines 3,7,12). Otherwise, the previously computed result is re-used (lines 4,8).

The correctness of **DynamicCheck** follows from the following theorem.

**Theorem 3.** *Given modified $\mathcal{M}'$ and $\varphi'$,* **DynamicCheck** *algorithm always terminates with either* TRUE *or a counterexample $CE$ to $\mathcal{M}' \sqsubseteq \varphi'$.*

We use the notion of weakest assumptions in proving the correctness of **DynamicCheck**. We know that for any DLA $M$, there must exist a weakest environment assumption DLA $WA$ such that $M \parallel E\varphi$ iff $E \sqsubseteq WA$. Suppose, we have a system of components $M_1, \ldots, M_n$ and a global property $\varphi$. Consider rules of form $M_i \parallel A_i \sqsubseteq A_{i-1}(1 \leq i \leq n-1, A_0 = \varphi)$ and $M_n \sqsubseteq A_{n-1}$ as used in our recursive procedure to show that $M_1 \parallel .. \parallel M_n \sqsubseteq \varphi$. It is clear that a weakest assumption $WA_1$ exists such that $M_1 \parallel WA_1 \sqsubseteq \varphi$. Given $WA_1$, it follows that $WA_2$ must exist so that $M_2 \parallel WA_2 \sqsubseteq WA_1$. Therefore, by induction on $i$, there must exist weakest assumptions $WA_i$ for $1 \leq i \leq n-1$, such that $M_i \parallel WA_i \sqsubseteq WA_{i-1}(1 \leq i \leq n-1, WA_0 = \varphi)$ and $M_n \sqsubseteq A_{n-1}$. Also, by Theorem 2, **UpdateAssumption**($A$, $CE$) must terminate starting from any valid assumption $A'$ with respect to $U'$ and a counterexample $CE \in L(A') \oplus U'$.

*Proof.* Suppose, without loss of generality, that component DLA $M'$, is upgraded. Note that after an upgrade, a weakest assumption $WA'$ (possibly different from $WA$) must exist for every $M' \in \mathcal{M}'$. We proceed by induction over the size $k$ of $\mathcal{M}'$. In the base case, it is clear that we need to model check $M'$ against $\varphi'$ only if either $M$ or $\varphi$ changed (line 3). This either returns a counterexample to $M' \sqsubseteq \varphi'$ or the previous $M \sqsubseteq \varphi$ (line 4) result holds.

Assume for the inductive case that **DynamicCheck**($\mathcal{M}' \setminus M'$, $A'$) terminates with either TRUE or a counterexample $CE$. It is clear from its definition that $A'$

computed by **GenerateAssumption** (line 5) is valid. If line 6 holds, i.e, $A' \neq A$ or $\mathcal{M} \setminus M \neq \mathcal{M}' \setminus M'$ then by inductive hypothesis, execution of line 7 terminates with either a TRUE result or a counterexample $CE$. Otherwise, the previously computed $CE$ result is used (line 8). It remains to be shown that lines (9-15) compute the correct return value based on this result.

If this result is TRUE then it follows from the soundness of the assume-guarantee rule that $\mathcal{M}' \sqsubseteq \varphi'$ and **DynamicCheck** returns TRUE (line 15). If $M' \parallel CE \not\sqsubseteq \varphi'$ (line 10), then by set-theoretic arguments based on the definitions of $A'$ and $CE$, we know that $\mathcal{M}' \not\sqsubseteq P'$ and a suitable witness $CE'$ (line 14) is returned by the algorithm. Otherwise, since $A'$ is valid, both **UpdateAssumption** (line 11) and **GenerateAssumption** (line 12) must terminate by learning a new assumption, say $A''$, such that $M' \parallel A'' \sqsubseteq \varphi'$. It follows from the proof of correctness of $L^*$ that $|A'| < |A''|$ and from the definition of weakest assumptions that $|A''| \leq |WA'|$. Also, by inductive hypothesis, line 13 must terminate with the correct $CE$ result. Hence, lines 9-13 of the **while** loop may be executed only a finite number of times until $|A''| = |WA'|$, when (by set-theoretic arguments) either the result is TRUE (line 15) or a witness counterexample $CE'$ (line 14) for $\mathcal{M}' \not\sqsubseteq P'$ is returned.

□

**Further optimizations.** Recall that our procedure reuses assumptions generated during previous compatibility checks. We further optimize it by identifying a subset of assumptions that have to be re-validated at the initialization of the next check. This optimization is enabled by the following lemma whose proof follows directly from Theorem 3 and definition of weakest assumptions.

**Lemma 1.** *Let $\mathcal{M} = \{M_1, \ldots, M_n\}$ be an assembly of components, $A = \{A_1, \ldots, A_{n-1}\}$ be a set of previously computed assumptions and $\mathcal{I} \subseteq \{1, \ldots, n\}$ be an index set. Also, let $\{M_i' \mid i \in \mathcal{I}\}$ be the set of new components. If $k$ is the minimum index of $\mathcal{I}$, then it is sufficient for* **DynamicCheck** *to re-validate only the assumptions in the set $\{A_j \mid j \geq k \wedge j \leq n\}$.*

## 6 Feedback

Recall that for some $i \in \mathcal{I}$, if our containment check detects that $C_i \not\sqsubseteq C_i^{'}$, it also computes a set $\mathcal{F}_i$. Intuitively each element of $\mathcal{F}_i$ represents a behavior of $C_i$ which is not a behavior of $C_i^{'}$. We now present our process of generating feedback from $\mathcal{F}_i$. In the rest of this section we will write $C$, $C^{'}$ and $\mathcal{F}$ to mean $C_i$, $C_i^{'}$ and $\mathcal{F}_i$ respectively.

Consider any behavior $\pi$ in $\mathcal{F}$. Recall that $\pi$ is a trace of a DLA $M$ obtained by predicate abstraction of $C$. By simulating $\pi$ on $M$, we construct an alternating sequence $Rep(\pi) = \langle s_1, \alpha_1, \ldots, s_n \rangle$ of states and actions of $M$ corresponding to $\pi$. Recall from our earlier discussion of predicate abstraction (cf. Section 4) that each $s_i$ is of the form $(st_i, \mathcal{V}_i)$ where $st_i$ is a statement of $C$ and $\mathcal{V}_i$ is a predicate valuation. Thus, $Rep(\pi) = \langle (st_1, \mathcal{V}_1), \alpha_1, \ldots, (st_n, \mathcal{V}_n) \rangle$.

We also know that $\pi$ represents an actual behavior of $C$ but not an actual behavior of $C^{'}$. Thus, there is a prefix $\mathit{Pref}(\pi)$ of $\pi$ such that $\mathit{Pref}(\pi)$ represents a behavior of $C^{'}$. However any extension of $\mathit{Pref}(\pi)$ is no longer a valid behavior of $C^{'}$. Note that $\mathit{Pref}(\pi)$ can be constructed by simulating $\pi$ on $C^{'}$. Let us denote the suffix of $\pi$ after $\mathit{Pref}(\pi)$ by $\mathit{Suff}(\pi)$. Since $\mathit{Pref}(\pi)$ is an actual behavior of $C^{'}$ we can also construct a representation for $\mathit{Pref}(\pi)$ in terms of the statements and predicate valuations of $C^{'}$. Let us denote this representation by $\mathit{Rep}'(\mathit{Pref}(\pi))$.

As our feedback we output, for each $\pi \in \mathcal{F}$, the following representations: $\mathit{Rep}(\mathit{Pref}(\pi))$, $\mathit{Rep}(\mathit{Suff}(\pi))$ and $\mathit{Rep}'(\mathit{Pref}(\pi))$. Note that such feedback allows us to identify the exact *divergence* point of $\pi$ beyond which it ceases to correspond to any concrete behavior of $C^{'}$. Since the feedback refers to program statement, it allows us to understand at the source code level why $C$ is able match $\pi$ completely but $C^{'}$ is forced to diverge from $\pi$ beyond $\mathit{Pref}(\pi)$. This makes it easier to modify $C^{'}$ so as to add back to it the missing behavior $\pi$.

## 7 Implementation and Experimental Evaluation

We implemented and evaluated the compatibility check phase for checking component substitutability in the ComFoRT framework. ComFoRT extracts abstract component DLA models from C programs using predicate abstraction. It also serves as a $MAT$ (cf. Section 5.1) for learning assumptions in the compatibility check. If the compatibility check returns a counterexample, the counterexample validation and abstraction-refinement modules of ComFoRT are employed to check for spuriousness and do refinement, if necessary.

We validated the component substitutability framework while verifying upgrades of a benchmark provided to us by our industrial partner, ABB Inc. (`http://www.abb.com`). The benchmarks consist of seven components which together implement an interprocess communication (IPC) protocol. The combined state-space is over $10^6$.

| Upgrade#(Prop.) | # Mem. Queries | $T_{orig}$ (msec) | $T_{ug}$ (msec) |
|---|---|---|---|
| $ipc_1(P_1)$ | 279 | 2260 | 13 |
| $ipc_1(P_2)$ | 308 | 1694 | 14 |
| $ipc_2(P_1)$ | 358 | 3286 | 17 |
| $ipc_2(P_2)$ | 232 | 805 | 10 |
| $ipc_3(P_1)$ | 363 | 3624 | 17 |
| $ipc_3(P_2)$ | 258 | 1649 | 14 |
| $ipc_4(P_1)$ | 355 | 1102 | 24 |

**Table 1.** Comparison of times required for original verification ($T_{orig}$) and verification on upgrade ($T_{ug}$) by **DynamicCheck**. *#Mem. Queries* denotes the total number of membership queries made during verification of the original assembly.

We used a set of properties describing functionality of the verified portion of the IPC protocol. We used upgrades of the *write-queue* ($ipc_1$) and the *ipc-queue* ($ipc_2$ and $ipc_3$) components. The upgrades had both missing and extra behaviors compared to their original versions. We verified two properties ($P_1$ and $P_2$) before and after the upgrades. We also verified the properties on a simultaneous upgrade ($ipc_4$) of both the components. $P_1$ specifies that a process may write data into the *ipc-queue* only after it obtains a lock for the corresponding critical section. $P_2$ specifies an order in which data may be written into the ipc-queue. Table 1 shows the comparison between time required for initial verification of the IPC system with the time taken by **DynamicCheck** for verification of upgrades. We observed that the previously generated assumptions in all the cases were sufficient to prove the properties on the upgraded system also. Hence, the compatibility check succeeded in a *small fraction of time* ($T_{ug}$) as compared to the time for compositional verification ($T_{orig}$) of the original system.

## 8 Conclusions and Future Work

We proposed a solution to the critical and vital problem of component substitutability consisting of two phases: *containment* and *compatibility*. The compatibility check performs compositional reasoning with help of a *dynamic* regular language inference algorithm and a model checker. Our experiments confirm that the dynamic approach is more effective than complete re-validation of the system after an upgrade. The containment check detects behaviors which were present in each component before but not after the upgrade. These behaviors are used to construct useful feedback to the developers. We observed that the order of components used to discharge the assume-guarantee rules has a significant impact on the algorithm run times and hence needs investigation. We would further like to investigate a modification of it based on a more efficient $L^*$ algorithm by Rivest et al. [18] in order to improve the performance of **DynamicCheck**.

**Acknowledgement.** We thank the FM 2005 referees for their invaluable comments and suggestions. The first author is also grateful to Corina S. Păsăreanu and Dimitra Giannakopoulou for insightful discussions on automated assume-guarantee and learning.

## References

1. R. Alur, P. Cerny, G. Gupta, P. Madhusudan, W. Nam, and A. Srivastava. Synthesis of interface specifications for Java classes. In *Symp. on Principles Of Programming Languages (POPL)*, 2005.
2. D. Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75(2), pages 87–106, November 1987.
3. S. Chaki, E. Clarke, D. Giannakopoulou, and C. S. Pasareanu. Abstraction and assume-guarantee reasoning for automated software verification. Technical Report 05.02, Research Institute for Advanced Computer Science (RIACS), August 2004.

4. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2–3), 2004.
5. S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Integrated Formal Methods*, volume 2999, pages 128–147. LNCS, 2004.
6. S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau. The ComFoRT reasoning framework. In *Proceedings of Computer Aided Verification (CAV)*, 2005.
7. S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software. In *3rd Workshop on Spec. and Ver. of Component-based Systems, ESEC/FSE*, 2004.
8. A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F. Y. C. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, pages pp. 428–441. LNCS 2404, Springer-Verlag, 2002.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 2619. Springer-Verlag, 2003.
11. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001.
12. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the ASE*, 2002.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of Computer Aided Verification*, 1997.
14. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 357–370. Springer-Verlag, 2002.
15. MAGIC. `http://www.cs.cmu.edu/~chaki/magic`.
16. S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, Oslo, Norway, 2004.
17. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, pages 123–144, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
18. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Information and Computation*, volume 103(2), pages 299–347, 1993.
19. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Int., 1997.
20. Learning for software. `http://www.cs.cmu.edu/~chaki/fme-05`.