Verification of Evolving Software*

Sagar Chaki Natasha Sharygina Nishant Sinha
chaki|natalie|nishants@cs.cmu.edu

ABSTRACT

We define the *substitutability* problem in the context of evolving software systems as the verification of the following two criteria: (i) previously established system *correctness properties* must remain valid for the new version of a system, and (ii) the *updated portion* of the system must continue to provide all (and possibly more) *services* offered by its earlier counterpart. We present a completely *automated procedure* based on learning techniques for regular sets to solve the substitutability problem for component based software. We have implemented and validated our approach in the context of the ComFoRT reasoning framework and report encouraging preliminary results on an industrial benchmark.

1. INTRODUCTION

Model checking [7] is a formal verification approach for detecting behavioral anomalies (including safety, reliability and security problems) in hardware and software systems. While model checking produces extremely valuable results, often uncovering defects that otherwise go undetected, there are several barriers to its successful integration into sofware development processes. In particular, model checking is hamstrung by scalability issues and is difficult for software engineers to use directly.

Most current research on model checking focuses on improving its scalability, and innovative techniques such as automated predicate abstraction and assume-guarantee reasoning have greatly improved the applicability of model checking to industrial-scale systems. However, there has been less progress on its transition from an academic to a practically viable discipline.

For instance, any software system inevitably evolves as designs take shape, requirements change, and bugs are discovered and fixed. While model checking is useful at each of these stages, it is usually applied to the entire system at every point irrespective of the amount of modification the system has actually undergone. 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 discourages its use by practitioners.

In this article we present a framework that, while not affecting the initial model checking effort, is aimed at reducing dramatically the effort to keep analysis results up-to-date with evolving systems. More specifically, we make the following two contributions. First, we define the *substitutability* problem as the verification of the following two criteria: (i) previously established system *correctness properties* must remain valid for the new version of an evolving software, and (ii) the *updated portion* of the system must continue to provide all (and possibly more) *services* offered by its earlier counterpart. Second, we present a completely *automated procedure* to solve the substitutability problem in the context of component based software.

We will define our notion of components and their behaviors more precisely later. Intuitively, a behavior of a component involves a sequence of observable message-passing interactions with other components. We denote the set of behaviors of a component C by $\mathcal{B}(C)$. Also given two components C and C' we will write $C \preccurlyeq C'$ to mean that $\mathcal{B}(C) \subseteq \mathcal{B}(C')$. Suppose we are given an assembly of components: $\mathcal{A} = \{C_1, \ldots, C_n\}$, a safety property φ, and a new component, C_i^S, to be used in place of C_i. We assume that φ was proven to hold on the original assembly $\mathcal{A}$. We wish to check for the *substitutability* of C_i^S for C_i in $\mathcal{A}$ with respect to the property φ. More specifically, our aim is to develop a procedure that achieves the following goals:

1. **Containment.** Verify that $C_i \preccurlyeq C_i^S$, i.e., every behavior of C_i is also a behavior of C_i^S. To this end, we will construct a component C_i^F such that $\mathcal{B}(C_i^F) = \mathcal{B}(C_i) \cup \mathcal{B}(C_i^S)$. In particular, if $C_i \preccurlyeq C_i^S$, then C_i^F will be the same as C_i^S. If the check fails, we provide the developers with feedback regarding the differences between C_i and C_i^S. Assuming that the missing behaviors would be added to C_i^S subsequently, we proceed with C_i^F as a safe abstraction of the new component in the next phase.

2. **Compatibility.** Verify that the new assembly $\mathcal{A}' = \{C_1, \ldots, C_i^F, \ldots, C_n\}$ satisfies the safety property φ. Note that in general $\mathcal{B}(C_i^F) \supset \mathcal{B}(C_i)$ owing to additional behaviors from C_i^S. Hence $\mathcal{A}'$ might violate φ even though φ was satisfied by $\mathcal{A}$. Therefore, in our framework, compatibility must be explicitly verified.

*This work was done as part of the Predictable Assembly from Certifiable Components initiative at the Software Engineering Institute.

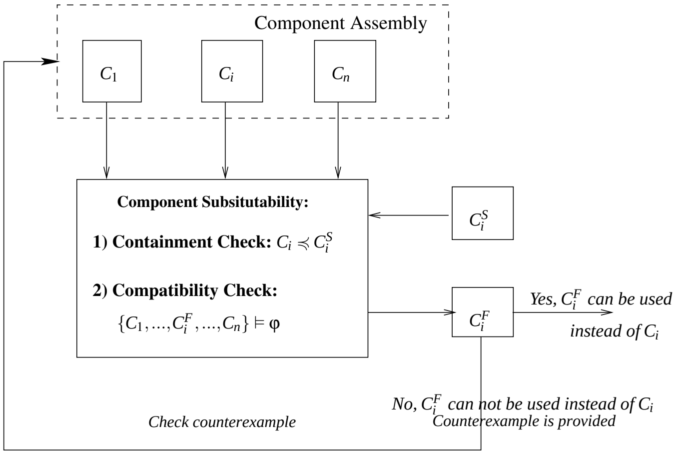

Figure 1: Overview of Component Substitutability.

We believe that our methodology is uniquely distinguished by the two phases it involves. **Containment** ensures that the substituted component satisfies the following criterion (CONT): it provides all services rendered by the original component C_i. If the new component C_i^S does not satisfy CONT, we generate a component which does, viz., C_i^F.

The new component C_i^S will usually be the result of design changes, bug fixes and other updates by a varied group of software professionals. Thus, it is unrealistic to expect C_i^S to always bear a specific relationship with the original component C_i. For instance, C_i^S will seldom refine C_i in the sense that all behaviors of the C_i^S are already there in C_i. We believe that in order to be viable, any approach to the substitutability problem must allow additional behaviors in C_i^S, and yet ensure that all old features of C_i continue to be supported. This is precisely the purpose of the containment phase which culminates in the construction of C_i^F. To the best of our knowledge, ours is the first framework to address this issue explicitly.

Compatibility guarantees that C_i^F can be safely integrated with the other components in the assembly. Recall that this must be checked explicitly since in general $\mathcal{B}(C_i^F) \supset \mathcal{B}(C_i)$. The compatibility check results in either a substitutable component C_i^F or produces a counterexample showing why the substitution of C_i^S is not feasible. The component C_i^F is such that: (i) it renders every service of C_i and yet (ii) the new assembly $\mathcal{A}' = \{C_1, \ldots, C_i^F, \ldots, C_n\}$ satisfies the safety property φ. The complete substitutability check procedure is outlined in Figure 1.

In addition to the computation of C_i^F, a major focus of the containment phase is to compute a set of behaviors in $\mathcal{B}(C_i) \setminus \mathcal{B}(C_i^S)$. Since these behaviors express features of C_i that are absent in C_i^S, they can be used to generate feedback for the developers. Such feedback can be of critical help by *localizing* the changes required to add the missing features back to C_i^S. We discuss this issue further in Section 5.4. We use automata-theoretic learning techniques for both the containment and compatibility phases of our approach. Specifically, we use techniques based on a learning algorithm for regular sets proposed by Angluin [2]. As we shall see later, our use of learning will aid in efficient feedback generation.

Finally, we employ state/event-based modeling techniques [4] in order to be able to model and reason about both the data and communication aspects of software. We use labeled Kripke structures (LKSs) to model, as well as to specify, software systems. This is important for our approach to be practically applicable to real-life component-based systems.

We implemented and validated our approach in the context of the Component Formal Reasoning Technology (ComFoRT) [10] reasoning framework being developed as part of the Predictable Assembly from Certifiable Components (PACC) [14] initiative at the Software Engineering Institute (SEI), Carnegie Mellon University (CMU). Specfically we implemented our substitutability framework as part of the model checking engine of ComFoRT, which is based on the C model checker MAGIC [5, 11] developed at CMU. In the rest of this article we will use the ComFoRT model checker and MAGIC synonymously.

The ComFoRT model checker employs *automated predicate abstracion* to extract finite models from concurrent C programs. Since abstract models often contain unrealistic behaviors, any counterexample obtained from an abstract model must be validated against the concrete system. If the counterexample is found to be spurious, the model must be refined and verification repeated. This iterative procedure is known as counterexample guided abstraction refinement (CEGAR) and implemented by MAGIC in a completely automated form. Furthermore, in the context of concurrent systems, MAGIC conducts both counterexample validation and abstraction refinement steps in a component-wise manner. Both predicate abstraction and automated CEGAR were critical for applying our technique to industrial component-based systems.

In summary, we believe that the presented component substitutability procedure has several advantages:

- Unlike conventional approaches, our methodology does not subscribe to the idea of *trace-theoretic refinement* while checking for substitutability. We believe that it is unduly restrictive to require a new component to directly refine its old counterpart in order be replaceable, and instead allow new components to have more behaviours. The *extra* behaviors are critical since they provide vendors with flexibility to implement new features into the product upgrades[1].

- Our technique identifies features of the old component C_i which are missing in the new component C_i^S. It also generates feedback to localize the modifications required in C_i^S to add the missing features back.

- Our method uses techniques based on learning algorithms for regular sets for accomplishing both phases of the substitutability check. This unified approach enables automatic verification of evolving software.

- Our technique supports component-wise abstraction, counterexample validation and abstraction refinement steps of the verification procedure and is thus expected to scale to large software designs.

This article is organized as follows. In Section 2 we discuss related work. Preliminary definitions and notations are

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

presented in Section 3 followed by a description of the L^* learning algorithm in Section 4. Details of our core component substitutability framework are presented in Section 5. Finally we present experimental results in Section 6 and conclude in Section 7.

2. RELATED WORK

This work relates to multiple projects targeting verification of component-based systems. In general, in contrast to our work, other 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.

Alfaro et. al. [9, 6] define a notion of interface automaton for modeling component interfaces and show compatibility between components via refinement and consistency between interfaces. However, automated techniques for constructing interface automata from a component implementations are not presented. Labeled Kripke structures (LKSs) coupled with the interface alphabet as they are used in this work for constructing component abstractions are similar to interface automata. In contrast, this work is based on sound predicate abstraction techniques that automatically extract LKSs from component implementations. Also our work is not limited to showing refinement between the old component and the new one and therefore it suits more to realistic systems.

Ernst et. al. [13] suggest a technique for checking compatibility of multi-component upgrades. However, they restrict themselves to input/output specifications of components by abstracting away temporal information about the sequence of actions. They acknowledge that even though the abstraction is not sound, their approach is useful in detecting important problems. In contrast to that, since our work employs existential abstraction, our framework is sound. Another drawback of this related work is that due to the nature of the input/output abstraction that eliminates sequencing of actions, component specifications are not complete. This is not a problem in our work since predicate abstraction is over-approximation and preserves all possible behaviors of the original components. Another difference with our project is that [13] component consistency criteria imply that all behaviors of component upgrades are also behaviors of their old counterparts.

The compatibility check in the current work is automated following ideas of Cobleigh et. al. [8] of using learning for regular sets techniques. While [8] focuses on automating assume-guarantee reasoning, our work solves a more general problem of the component substitutability. We use compositional reasoning to discharge new behaviors of the component upgrades in new assemblies. Our approach differs from theirs in a number of ways. Firstly, we take care of state labeling information of LKSs by including both state and transition labels in the language definition of LKSs. Also in our case, the compatibility check is embedded in the abstraction-refinement framework for C programs, which makes verification tractable.

3. BACKGROUND AND NOTATION

Definition 1 (Finite Automata). *A non-deterministic finite automaton (NFA) is a 5-tuple $(S, S_0, \Sigma, \Delta, F)$ with S a finite set of states, $S_0 \subseteq S$ a set of initial states, Σ a finite alphabet, $\Delta \subseteq S \times \Sigma \times S$ a transition relation, and $F \subseteq S$ a set of final (accepting) states. The language of an NFA M is denoted by $L(M)$ and defined in the usual way. A deterministic finite automaton (DFA) is a NFA such that S_0 has exactly one element and Δ is a function from $S \times \Sigma$ to S.*

Definition 2 (Labeled Kripke Structure). *A labeled Kripke structure (LKS for short) is a 6-tuple $(S, Init, AP, \mathcal{L}, \Sigma, \delta)$ with S a finite set of* states, *$Init \subseteq S$ a set of initial states, AP a finite set of (atomic) state propositions, $\mathcal{L} : S \rightarrow 2^{AP}$ a state-labeling function, Σ a finite set of events or actions (alphabet), and $\delta \subseteq S \times \Sigma \times S$ a transition relation.*

For any NFA, DFA (or LKS) with transition relation Δ (or δ), we write $q \xrightarrow{\alpha} q'$ to mean $(q, \alpha, q') \in \Delta$ (or $(q, \alpha, q') \in \delta$). We wish to define the language of an LKS in terms of that of an equivalent NFA. However since the states of an NFA are not labeled, we will have to transform the state labeling of the LKS to events in accordance with some scheme. Moreover, we would like to vary the alphabet of the resulting NFA by focusing on different sets of propositions. This idea is captured by an induced NFA.

Definition 3 (Induced NFA). *The NFA induced by an LKS $M = (S, Init, AP, \mathcal{L}_M, \Sigma_M, \delta)$ is denoted by $NFA(M)$ and defined as: $(S \cup \{s_i\}, \{s_i\}, \Sigma_N, \Delta, S \cup \{s_i\})$ where $s_i \notin S$ is a new state, $\Sigma_N = (\Sigma_M \cup \{\tau\}) \times 2^{AP}$, and Δ is defined as follows:*

$$\forall s \in Init \centerdot s_i \stackrel{<\tau, \mathcal{L}_M(s)>}{\longrightarrow} s \in \Delta$$

$$\forall s \xrightarrow{\alpha} s' \in \delta \centerdot s \stackrel{<\alpha, \mathcal{L}_M(s')>}{\longrightarrow} s' \in \Delta$$

Definition 4 (LKS Language). *The language of an LKS M is denoted by $L(M)$ and defined as the language of the induced $NFA(M)$. Note that $L(M)$ is prefix-closed:*

$$\forall w \centerdot w \in L(M) \implies \forall w' \in prefix(w) \centerdot w' \in L(M)$$

Definition 5 (Abstraction). *Given two LKSs M_1 and M_2 we say that M_2 is an abstraction of M_1, denoted by $M_1 \preccurlyeq M_2$, iff $L(M_1) \subseteq L(M_2)$. Note that this concretizes our intuitive notion of abstraction being a form of behavioral containment since the set of behaviors of an LKS is captured by its language.*

Definition 6 (Parallel Composition). *Let $M_1 = (S_1, Init_1, AP_1, \mathcal{L}_1, \Sigma_1, \delta_1)$ and $M_2 = (S_2, Init_2, AP_2, \mathcal{L}_2, \Sigma_2, \delta_2)$ be two LKSs. The parallel composition of M_1 and M_2, denoted by $M_1 \parallel M_2$, is the LKS $(S_1 \times S_2, Init_1 \times Init_2, AP_1 \cup AP_2, \mathcal{L}, \Sigma_1 \cup \Sigma_2, \delta)$, where $\mathcal{L}(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$, and δ is such that $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ iff one of the following holds:*

1. *$\alpha \in (\Sigma_1 \setminus \Sigma_2) \cup \{\tau\}$ and $s_1 \xrightarrow{\alpha} s_1'$ and $s_2' = s_2$*
2. *$\alpha \in (\Sigma_2 \setminus \Sigma_1) \cup \{\tau\}$ and $s_2 \xrightarrow{\alpha} s_2'$ and $s_1' = s_1$*
3. *$\alpha \in (\Sigma_1 \cap \Sigma_2) \setminus \{\tau\}$ and $s_1 \xrightarrow{\alpha} s_1'$ and $s_2 \xrightarrow{\alpha} s_2'$*

In other words, LKSs must synchronize on shared actions (except τ) and proceed independently on local actions (and τ). This notion of parallel composition is derived from CSP [16].

Definition 7 (Components and Models). *In our framework a component is essentially a C program communicating with other components via blocking message passing. Since C programs are in general infinite state systems we will extract finite LKS models from components via predicate abstraction [5], and perform further analysis on these models. The data and message-passing aspects of a component C will be transformed conservatively into predicates and actions of its model M. Consequently, M is guaranteed to be a sound abstraction of C.*

Definition 8 (Assembly and Environment). *A component assembly $\mathcal{A}$ is a collection of components $\{C_1, \ldots, C_k\}$. For $1 \leq i \leq k$, let M_i be a model of C_i. Then the collection of models $\{M_1, \ldots, M_k\}$ is called a model assembly corresponding to $\mathcal{A}$ and denoted by $M_{\mathcal{A}}$. The environment of a component C_i with respect to $\mathcal{A}$ is a set $Env(C_i) \subseteq \mathcal{A}$ such that each component in $Env(C_i)$ communicates with C_i via message-passing. Similarly, the environment of a model M_i with respect to $M_{\mathcal{A}}$ is the model assembly corresponding to $Env(C_i)$.*

4. LEARNING REGULAR SETS

Central to our substitutability check procedure is the L^* inference algorithm for regular languages developed by Angluin [2] and later improved by Rivest et. al. [15]. In the rest of this article we will only concern ourselves with the original algorithm of Angluin. Let U be an unknown regular language over some alphabet Σ. In order to learn U, L^* needs to interact with a *minimally adequate teacher* MAT for U, which can answer two kinds of queries.

1. *Membership*. Given a word $\rho \in \Sigma^*$, MAT returns $true$ if $\rho \in U$ and $false$ otherwise.
2. *Candidate*. Given a DFA D, MAT returns $true$ if $L(D) = U$ and $false$ otherwise. If MAT returns $false$, it also returns a counterexample word w in the symmetric difference of $L(D)$ and U.

Given an unknown regular language U and a MAT for U, the L^* algorithm *iteratively* constructs a minimal DFA D such that $L(D) = U$. It maintains an observational table T where it records information about elements and non-elements of U. The rows of T are labeled by the elements of $S \cup S \cdot \Sigma$ where S is a prefix-closed set over Σ^*. The columns of T are labeled by the elements of a suffix-closed set E over Σ^*. Let us denote the set $S \cup S \cdot \Sigma$ by *Row*. Then the following condition always holds for T:

$$\forall s \in Row \centerdot \forall e \in E \centerdot T[s,e] = true \iff s \cdot e \in U$$

Additionally, for any $s \in Row$, let us define a function r_s as follows:

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

Then T is said to be *closed* and *consistent* if the following two conditions hold respectively:

$$\forall t \in S \cdot \Sigma \centerdot \exists s \in S \centerdot r_s = r_t$$

$$\forall s_1, s_2 \in S \centerdot r_{s_1} = r_{s_2} \implies \forall a \in \Sigma \centerdot r_{s_1 \cdot a} = r_{s_2 \cdot a}$$

L^* starts with a table T such that $S = E = \emptyset$ and in each iteration proceeds as follows. It first updates T using membership queries (starting with words of length at most one) till T is closed and consistent. Next L^* builds a candidate DFA $D(T)$ from T and makes a candidate query with $D(T)$. If the MAT returns $true$ to the candidate query, L^* returns $D(T)$ and stops. Otherwise, L^* updates T with all prefixes of the counterexample returned by MAT and proceeds with the next iteration. The complexity of L^* is expressed by the following theorem.

Theorem 1. *[2] If n is the number of states of the minimum DFA 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 T is of size $O(m^2n^2 + mn^3)$.*

Optimizations. Note that in our case, the unknown language U is always prefix-closed since, by definition, the language of LKSs are prefix-closed. This allows us to augment L^* with some optimizations similar to those proposed by Berg et. al. [3]. A prefix-closed language L is characterized by the property that for a trace $\rho \in L$, all prefixes of ρ are in L. Conversely, for a trace $\rho \notin L$, no extension of ρ is in L.

Therefore, whenever L^* makes a membership query with ρ, we first look up all of ρ's prefixes in a *query cache*. The cache returns $false$ if any of the prefixes are present and marked $false$. Otherwise if ρ itself is present and marked $true$, the cache returns $true$. If none of the above cases hold, the query is passed on to the MAT. These optimizations yielded up to 20% speedup during our experiments (cf. Section 6).

5. COMPONENT SUBSTITUTABILITY

Recall that the substitutability problem involves two major phases: containment and compatibility. Suppose we are given an assembly of components: $\mathcal{A} = \{C_1, \ldots, C_n\}$ and an LKS φ such that $\mathcal{A} \preccurlyeq \varphi$. Also, we are given a new component C_i^S to be used in place of C_i. Our goal is to check for the substitutability of C_i^S for C_i in $\mathcal{A}$ while preserving all previous services as well as the validity of φ. Figure 2 shows the schematic diagram of our substitutability framework.

The complete substitutablility procedure occurs in a CEGAR-style loop. In each iteration of the loop, substitutablility checks are performed on abstract models instead of concrete components. Suppose M_i and M_i^S are the models of C_i and C_i^S respectively. Therefore we will check for the substitutability of M_i^S for M_i with respect to the property φ. Note that the final result is either a substitutable model M_i^F or a counterexample CE. However, CE may be spurious with respect to either C_i or C_i^S and therefore must be checked for validity against both. If CE is spurious we will refine M_i and/or M_i^S and repeat the CEGAR loop. Otherwise, we will report CE as an evidence of non-substitutablility of C_i^S and terminate.

In case we are able to prove substitutablility, we will also generate a set of traces in $L(M_i) \setminus L(M_i^S)$. These traces will then be used to provide constructive feedback to the developers (cf. Section 5.4).

5.1 Containment

The containment check accepts models M_i and M_i^S as inputs. In general, it might also be provided with an LKS B which captures a set of prohibited behaviors such

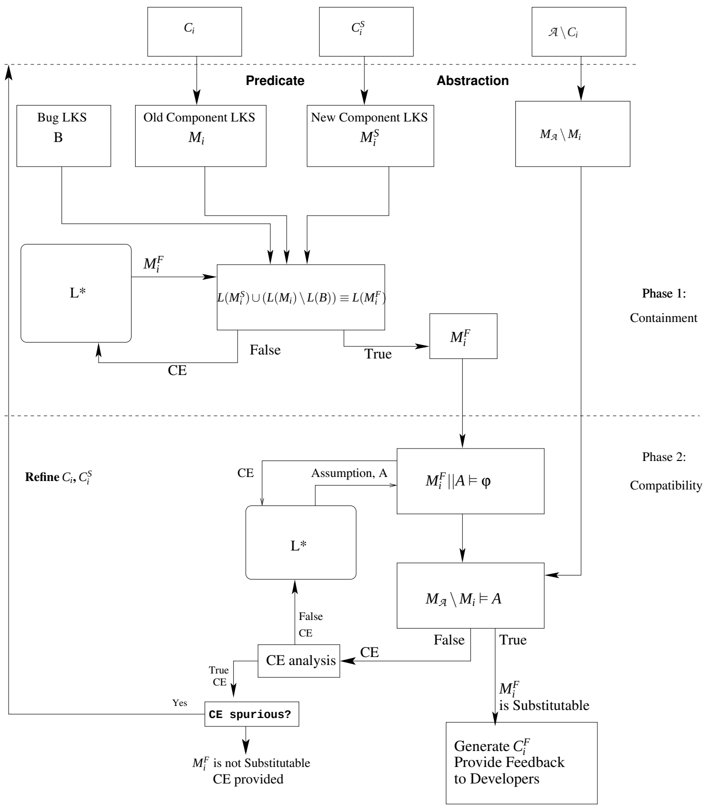

Figure 2: Substitutability framework.

as previously detected bugs. Our goal in this phase is to apply a learning algorithm to build a new DFA M_i^F, which includes all the behaviors of M_i^S and M_i except those of B. In other words we wish to learn the language $U = L(M_i^S) \cup (L(M_i) \setminus L(B))$. Additionally we wish to compute a set of traces $\mathcal{F} \subseteq L(M_i) \setminus L(M_i^S)$ which can be subsequently used for feedback generation.

Recall that $NFA(M)$ denotes the NFA induced by an LKS M. Thus the alphabet over which the learning occurs is $\Sigma_{NFA(M_i)} \cup \Sigma_{NFA(M_i^S)}$. In addition, the membership and candidate queries are discharged as follows using a model checker.

Membership. In order to check for membership of a word ρ, the model checker checks whether it is accepted by either M_i or M_i^S and not accepted by B. Note that these three checks are performed separately without composing M_i, M_i^S and B.

Candidate. The candidate query for an intermediate candidate DFA D involves the following language equivalence check: $U = L(D)$. We avoid explicit computation of U (which would defeat the entire purpose of learning) while discharging the candidate query as follows. First we subdivide the candidate query into two subset checks: (i) $U \subseteq L(D)$ and (ii) $L(D) \subseteq U$. Next we perform the first check by verifying individually: (a) $L(M_i^S) \subseteq L(D)$ and (b) $L(M_i) \setminus L(B) \subseteq L(D)$. Finally we peform the second check in the following iterative manner.

1. Check if $L(D) \subseteq L(M_i^S)$. If the answer is *yes*, then $L(D) \subseteq U$ and we return *true*. Otherwise we get a trace $CE \in L(D) \setminus L(M_i^S)$.
2. Check if $CE \in L(M_i) \setminus L(B)$. If not, then $CE \in L(D) \setminus U$ and we return $false$ along with CE as counterexample. Otherwise $CE \in L(M_i) \setminus L(M_i^S)$.
3. Add CE to $\mathcal{F}$. Repeat from step 1 but look for counterexamples other than those already in $\mathcal{F}$. We achieve this by suitably modifying our model checker that performs step 1 above.

As mentioned previously, a key feature of our framework is to allow the new component to have more behaviors than the previous one. These extra behaviors might cause the new component assembly to violate the global property φ. Therefore the compatibility of the new component with the rest of the assembly must be verified separately. This forms the basis of the next phase in substitutability.

5.2 Compatibility

Recall that the global safety property is expressed as an LKS φ and that we write $M_1 \preccurlyeq M_2$ to mean $L(M_1) \subseteq L(M_2)$. On successful completion of the containment phase we obtain a DFA M_i^F such that $L(M_i^F) = L(M_i^S) \cup (L(M_i) \setminus L(B))$. We now need to verify that the component M_i^F is *compatible*, i.e, safe under the given environment $Env(M_i)$. In other words we need to check that $M_i^F \parallel Env(M_i) \preccurlyeq \varphi$. This is done by a combination of assume-guarantee style reasoning and learning similar to Cobleigh et. al. [8]. Hence we will not describe this phase in much detail but simply summarize the salient features as follows:

1. Learn an assumption DFA A for M_i^F such that $M_i^F \parallel A \preccurlyeq \varphi$ using L^* with a model checker as a MAT.

2. Check if $Env(M_i) \preccurlyeq A$. If so, return *true*. Otherwise a counterexample CE is obtained.

3. Check if $M_i^F \parallel CE \preccurlyeq \varphi$. If so, use CE to weaken the A and repeat from step 1. Otherwise, return false along with CE as the counterexample to the compatibility phase.

Note that since $L(M_i^F)$ contains traces from both $L(M_i)$ and $L(M_i^S)$, any counterexample CE returned by the above procedure must be checked against both C_i and C_i^S for spuriousness.

5.3 Why Learn?

Our use of techniques based on L^* provides us the following advantages:

- We can use our model checker to answer the membership and candidate queries and also to generate a counterexample in the event of the failure of a candidate query.

- Our use of learning during the containment phase enables us to compute a DFA for $L(M_i^S) \cup (L(M_i) \setminus L(B))$ without having to compose M_i, M_i^S or B. As a side-effect we are also able to generate a set of traces $\mathcal{F} \subseteq L(M_i) \setminus L(M_i^S)$ which can be subsequently used for feedback generation.

- L^* is incremental, computes the smallest DFA, and also enables us to proceed without precisely defining the language to be learned, e.g., during compatibility.

- Efficiency of L^* depends only on the length of individual counterexamples and the minimum automaton representation of the unknown language. This characteristic allows us to leverage the competency of the model checker in generating suitable counterexamples.

5.4 Feedback to Developers

In this section we present several approaches for providing feedback to developers that will enable them to add missing features back to C_i^S. Recall that upon successful completion of the substitutability check, we obtain a set of traces $\mathcal{F} \subseteq L(M_i) \setminus L(M_i^S)$. Let π be any trace in $\mathcal{F}$. Hence $\pi \in L(M_i)$. Recall that $L(M_i)$ was defined to be the language of the induced NFA $NFA(M_i)$. Since the actions of the NFA induced by any LKS M contain information about the propositional labeling of M, it is possible to retrieve this information from any trace of $NFA(M)$ and convert it back to a corresponding trace of M. Thus, in particular, we can obtain a trace ρ of M_i corresponding to π. The trace ρ constitutes our first level of feedback.

By itself, trace ρ provides limited assistance to a developer. While it shows a missing behavior, it does not relate this behavior to the new component C_i^S. Our next level of feedback attempts to improve this situation by identifying portions of the actual code for C_i^S which are relevant to the missing behavior expressed by Tr. One way to achieve this would be via a mapping *Map* from statements (or control points) of C_i to those of C_i^S. We intend to investigate the automated generation of *Map* as well as the fragments of C_i^S relevant to ρ as part of our future work.

Our most advanced form of feedback is aimed at eliminating completely the need for developers to modify C_i^S. Suppose we are given a trace ρ and the portions of C_i^S relevant to ρ as described earlier. Our goal is to automatically generate a modified version C_i^F of C_i^S such that C_i^F has all the features of C_i which were missing in C_i^S. Clearly, this is an extremely difficult problem in the general case and we must impose appropriate restriction in order to find effective solutions. For instance, we can restrict our changes to only branch statements and library routine calls. Additionally such code modifications can be made in the form of templates which can be inspected, and improved for performance if necessary, by the developers.

6. IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We implemented our methodology for checking component substitutability in the ComFoRT framework developed at Carnegie Mellon Software Engineering Institute. The ComFoRT model checking engine is based on Magic model checking tool [5]. We used Magic capabilities to extract finite LKS models from C programs using predicate abstraction to construct abstract component models. The Magic model checker also serves as a *minimally adequate teacher* for the learning algorithms of the containment and compatability checks. Each of these checks instantiates its own L^* learner, which perform the task of learning their respective DFAs. If the compatibility check returns a counterexample, the counterexample validation and abstraction-refinement modules of Magic are employed to check for spuriousness and do refinement, if necessary.

We validated the component subsitutability framework while verifying upgrades of a benchmark provided to us by our industrial partner, ABB Inc. [1]. We verified part of an interprocess communication protocol (IPC-1.6) used to mediate communication in a multi-threaded robotics control automation system that must satisfy safety-critical requirements.

The IPC protocol provides multiple forms of communication including synchronous point-to-point, broadcast, publish/subscribe, and asynchronous communication, all of which are implemented in terms of messages passing between queues owned by different threads. The protocol handles the creation and manipulation of message queues, synchronizing access to shared data using various operating system primitives (e.g., semaphores and critical sections), and cleaning up internal state when a communication fails or times out.

We analyzed the portion of the IPC protocol that is used for synchronous communication among multiple threads. With this type of communication, a sender sends a message to a receiver and blocks until an answer is received or it times out. A receiver asks for its next message and blocks until a message is available or it times out. Whenever the receiver gets a synchronous message, it is then expected to send a response to the message's sender. The target of our verification was the IPC component that implements this communication scheduling, comprising of about 1500 lines of C code. We abstracted away communication with other IPC components by specifying external choice channels that were set to pass all possible inter-component inputs nondeterminstically.

We used a set of properties describing functionality of the verified portion of the IPC protocol. For example, we checked that no faults discovered during testing of older versions of the code are present in the current code. An example of such a property is an assertion that no messages are lost from the queue without being delivered to the receiver.

We upgraded the $WriteToQueue$ component of the IPC assembly by both adding and removing some behaviors. These modifications resulted in a violation of the containment check and hence a substitutable DFA was produced at the end of this check. This DFA was found to be smaller as compared to the old and new component LKSs. We believe this was the case with our examples since multiple states of the LKS obtained by predicate abstraction of a single control location in the C program accepted the language. Therefore, they were determined to be equivalent by the L^* algorithm and were collapsed into a single one in the DFA.

The compatibility check saves an order of magnitude in terms of memory (upto 50%) as compared to the verification of the composition of components directly. This is primarily because the assumptions generated during the assume-guarantee reasoning were of a small size as compared to the component LKSs, which in turn led to lesser sizes of product automata. Also, we observed verification time improvements of order of upto 10% by implementing the query cache for the membership queries in the learner. Another 10% improvement in time was obtained by doing the *prefix-closed* optimizations (cf. Section 4) during learning.

7. CONCLUSIONS AND FUTURE WORK

The current work proposes a solution to the component substitutability problem using a regular language inference along with a model checker. Although we provide an approach oriented towards the new component for verifying global assembly properties during its evolution, there are several ways to improve its efficiency. For example, we would like to use the results from verifying the previous assembly while checking the compatibility of the new component. Further, since the earlier assembly was verified to be correct, only the new behaviors of the evolved component should be considered during compatibility.

This work also brings out several interesting avenues of research. The L^* algorithm is a general framework for learning regular languages. However, our goal is to learn program behavior, which are earmarked by specific characteristics, e.g., the language is prefix-closed. We intend to investigate more such domain-specific characteristics of programs which increase the efficiency of the inference algorithm. We also aim to develop an algorithm for learning LKSs directly instead of appealing to its language definition in terms of NFA.

We have used an assume-guarantee framework for learning only safety properties in this work. In order to extend its scope to liveness properties, we plan to study the learning of $\omega - regular$ [12] languages. Finally, we plan to focus on providing improved feedback to developers which will enable them to add missing features and/or fix bugs in the updated components.

8. REFERENCES

[1] ABB Inc. `http://www.abb.com`.

[2] D. Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75(2), pages 87–106, November 1987.

[3] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin's learning. Technical Report 2003-039, Department of Information Technology, Uppsala University, Aug. 2003.

[4] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Integrated Formal Methods*, pages 128–147, 2004.

[5] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of ICSE 2003*, pages 385–395, 2003.

[6] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F. Y. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification*, pages pp. 428–441. Lecture Notes in Computer Science 2404, Springer-Verlag, 2002.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[8] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619. Springer-Verlag, April 2003.

[9] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[10] J. Ivers and N. Sharygina. Overview of ComFoRT: A model checking reasoning framework. *CMU/SEI-2004-TN-018*, 2004.

[11] MAGIC. `http://www.cs.cmu.edu/~chaki/magic`.

[12] O. Maler and L. Staiger. On syntactic congruences for omega-languages. In *Symposium on Theoretical Aspects of Computer Science*, pages 586–594, 1993.

[13] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, Olso, Norway, June 16–18, 2004.

[14] PACC website. `http://www.sei.cmu.edu/pacc`.

[15] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Information and Computation*, volume 103(2), pages 299–347, April 1993.

[16] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, London, 1997.