Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Verification of Evolving Software via Component Substitutability Analysis ⋆

Sagar Chaki[1], **Edmund Clarke**[2], **Natasha Sharygina**[3,2], **Nishant Sinha**[4]

[1] Software Engineering Institute, Pittsburgh, USA

[2] Carnegie Mellon University, School of Computer Science, Pittsburgh, USA

[3] Universita della Svizzera Italiana, Lugano, Switzerland

[4] Carnegie Mellon University, Electrical and Computer Engineering Department, Pittsburgh, USA

Received: August 7, 2009/ Revised version: August 7, 2009

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

⋆ This is an extended version of a paper, *Dynamic Component Substitutability Analysis*, published in the proceedings of the Formal Methods 2005 conference, Lecture Notes in Computer Science, Vol. 3582, by the same authors. This research was sponsored by the National Science Foundation under grant nos. CNS- 0411152, CCF-0429120, CCR-0121547, and CCR-0098072, the Semiconductor Research Corporation under grant no. TJ-1366, the US Army Research Office under grant no. DAAD19-01-1-0485, the Office of Naval Research under grant no. N00014-01-1-0796, the ICAST project and the Predictable Assembly from Certifiable Components (PACC) initiative at the Software Engineering Institute, Carnegie Mellon University. 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 any sponsoring institution, the U.S. government or any other entity.

how to improve the components. The substitutability approach has been implemented and validated in the COMFORT reasoning framework, and we report encouraging results on an industrial benchmark.

1 Introduction

Correctness of computer software is critical in today's information society, especially for software that runs on computers embedded in our transportation and communication infrastructure. Errors in complex software systems have caused large-scale economic losses in the past. Software bugs, especially in multi-threaded systems, are notoriously difficult to detect and fix. Therefore, it is necessary to employ automated formal verification methods to validate and debug critical software systems.

Programs in imperative languages like C or C++ are executed line-by-line in what is called a *thread of control*. It is tempting to hope that a line-by-line inspection of the code, following this thread of control, will uncover all the flaws in a program. The problem is that complex systems have many software components running in parallel, so there are many different threads of control that run simultaneously. While one of these threads may be executing some statement in its program, another thread, with exactly the same program, may be executing an entirely different line of code concurrently. Consequently, in the presence of multiple threads, any combination of program lines that the threads can execute must be considered.

The *state* of the program is the location of the control in each thread and the values of the program variables. To discover flaws, the possible states of the program must be explored. To illustrate the large number of states that concurrency can cause, consider the small program in Figure 1. It has one variable x, which is initialized with zero. It has two threads (A and B) of control and only four lines of code in total. The first line in both threads simply idles until x becomes zero. The second line sets x to 1 or 2, respectively. We assume that each program step is atomic. Despite its tiny size, the program has 10 reachable states. The explosion in the number of reachable states is due to the different combinations of program locations in the two threads A and B. Thus, a manual search for errors in large concurrent programs is infeasible.

Model checking is an automated technique for the exploration of all the states of a system [20,22]. Introduced in 1981, it is now a standard verification technique in the hardware industry. It has been successfully used to find bugs in circuitry that would have been hard to find by inspection alone.

Thread A

```
1  while(x!=0) skip;
2  x=1;
3
```

Thread B

```
1  while(x!=0) skip;
2  x=2;
3
```

Fig. 1 A Small Program with Two Threads of Control

The use of model checking has lead to major enhancements in the reliability and robustness of software. The basic idea of software model checking [12,44] is to explore all the states of the software system systematically. The states are checked for errors. Such an error may be division by zero, a race condition or a violated assertion. Once such an erroneous state is found, it is reported to the programmer together with a counterexample (i.e., an error trace), which demonstrates the flaw. In practice, counterexamples are very helpful for understanding the nature of errors and fixing them.

However, the effectiveness of the model checking of such systems is severely constrained by the state space explosion problem (by the sheer number of states a program can be in). If there are too many states, it becomes impossible to explore all of them, even on a powerful computer.

Much of the research in this area is therefore targeted at reducing the state space of the model used for verification. One principal method in state space reduction of software systems is *abstraction*. Abstraction techniques reduce the program state space by generating a smaller set of states in a way that preserves the relevant behaviors of the system. Manual abstractions of large software systems require considerable expertise and are error prone. Industrial applications of model checking therefore favor automated ways to compute the abstract model. One such method, called *predicate abstraction* [32,25], has proven to be particularly successful when applied to large software programs. We have exploited predicate abstraction while developing a solution to the problem of establishing the correctness of

evolving systems. We describe predicate abstraction in Section 2 and its application to verification of evolving software in Section 4.

The other principal approach in reducing the state space of the verifiable model is *compositional reasoning*. Compositional reasoning partitions verification into checks of individual modules, while the global correctness of the composed system is established by constructing a proof outline that exploits the modular structure of the system. We used the *assume-guarantee* style of compositional reasoning to support verification of evolving systems [47, 41, 49]. We describe the assume-guarantee reasoning paradigm and its application to verification of evolving software in Section 5.

In this article, we focus on a particular model checking problem, namely verification of evolving software. 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 behaviors can lead to unavailability of previously provided services. Second, the addition of new behaviors can lead to violation of global correctness properties that were previously being respected. Although software evolution may involve both changing the component decomposition of the system as well as communication structure between components, the approach presented in this work focuses on assemblies in which both the decomposition and the communication structure remain the same.

In this context, the *substitutability* problem can be defined as the verification of the following two criteria: (i) any *updated portion* of a software system 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. The above two criteria correspond to changes due to addition and removal of behaviors or services of software components respectively; we believe that they are sufficient to model a large variety of software upgrades.

Model checking can be used at each stage of a system's evolution to solve both the above problems. 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 effort 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 if 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 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 write $C \preccurlyeq C'$ 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 φ (e.g., the system can enter an error state upon execution). 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 φ. This article presents a framework that achieves this goal by performing 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., $C_i \preccurlyeq C_i^{'}$. If $C_i \not\preccurlyeq C_i^{'}$, we also construct a set $\mathcal{F}_i$ of behaviors in $Behv(C_i) \setminus Behv(C_i^{'})$ which is used subsequently for providing feedback to the assembly designer. Note that the upgrade may involve the removal of behaviors designated as errant, say B. In this case, we check $C_i \setminus B \preccurlyeq C_i^{'}$ since behaviors of B are clearly absent in $C_i^{'}$. In general, B should contain the set of behaviors that have been intentionally removed (buggy or otherwise), so that they do not occur as spurious counterexamples in the containment check.

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 $i \in \mathcal{I}$. In general, it is not the case that for each $i \in \mathcal{I}$, $C_i^{'} \preccurlyeq C_i$. Therefore, the new assembly $\mathcal{C}'$ may have more behaviors than the old assembly $\mathcal{C}$. Hence $\mathcal{C}'$ might violate φ even though $\mathcal{C}$ did not. Thus, our second task is to verify that $\mathcal{C}'$ satisfies the safety property φ (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 an automated assume-guarantee reasoning approach for a fixed system assembly, developed by Cobleigh et al. [23] which is based on a combination of learning algorithms for regular languages with model checking. This paper, in contrast, proposes a *dynamic* assume-guarantee reasoning procedure for evolving systems. The procedure is dynamic, in the sense that it learns appropriate environment assumptions for the new components by *reusing* the environment assumptions for their older versions.

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 to be replaceable. The *extra* behaviors are critical since they provide vendors with the 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.

We have implemented the substitutability check as part of the ComFoRT [40] reasoning framework. For the compatibility check, we experimented with an industrial benchmark and report encouraging results in Section 6.

The article is organized as follows: Section 2 provides some background on model checking, abstraction and compositional reasoning. Section 3 defines the notation used throughout the article and presents the L^* learning algorithm

that forms the basis of the compatibility analysis. Sections 4,5 describe the problem of verification of evolving systems and present a detailed description of the containment and compatibility algorithms that we have developed to overcome difficulties in the verification of evolving programs. Section 7 provides an overview of related work, and Section 8 summarizes the contributions of this article.

2 Overview of the Model Checking Approach

In formal verification, a system is modeled mathematically, and its specification (also called a *claim* or *property* in model checking) is described in a formal language. Model checking [20] is an automated formal verification technique which checks whether a system satisfies a desired claim through an exhaustive search of all possible executions of the system. The exhaustive nature of model checking addresses the issue of inadequate coverage that is typically a drawback of testing.

Model checking is a technique for verifying finite-state concurrent systems. One benefit of this restriction to finite-state systems is that verification can be performed automatically. Given sufficient resources, model checking always terminates with a "yes" or "no" answer. Moreover, it can be implemented by efficient algorithms.

2.1 The Process of Model Checking

Model checking involves the following steps:

1. The system is modeled using the description language of a model checker, producing a model M.
2. The claim to check is defined using the specification language of the model checker, producing a temporal logic formula ϕ.
3. The model checker automatically checks whether $M \models \phi$ (i.e., whether M satisfies ϕ).

The model checker explores all system executions captured by the model and outputs "yes" if the claim holds in the model (M) and "no" otherwise. When the claim is not satisfied, the model checker produces a *counterexample* consisting of a system behavior that causes the failure. A counterexample defines an execution trace that violates the

claim. Counterexamples are one of the most useful features of model checking, as they allow users to understand why a claim is not satisfied.

2.2 Model Checking Software

Applying model checking to software, as opposed to hardware, is complicated by several factors, ranging from the difficulty of modeling computer systems (due to the complexity of programming languages as compared to hardware description languages) to difficulties in specifying meaningful claims for software using the usual temporal logic formalism of model checking. The most significant limitation, however, is the *state space explosion* problem (which applies to both hardware and software), whereby the complexity of model checking becomes prohibitive.

State space explosion results from the fact that the size of the state transition system is exponential in the number of variables and concurrent units in the system. When the system is composed of several concurrent units, its combined description may lead to an exponential explosion as well. The state space explosion problem is the subject of most model checking research.

The following state space reduction techniques are commonly used during verification of software:

- **Abstraction:** A smaller abstract system is constructed such that the claim holds for the original system if it holds for the abstract system.
- **Counterexample-guided abstraction refinement:** Abstracted systems are refined iteratively using information extracted from counterexamples until an error is found or it is proven that the system satisfies the verification claim.
- **Compositional reasoning:** Verification is partitioned into checks of individual modules, while the global correctness of the composed system is established by constructing a correctness proof that exploits the modular structure of the system.

2.2.1 Abstraction. Abstraction is one of the principal techniques for reducing the complexity of a verification problem [19, 42, 18, 8]. Abstraction techniques reduce the state space by mapping the set of actual system states to an

abstract set of states that preserve the behavior of the actual system. Abstractions are usually performed in an informal, manual manner and require considerable expertise. Predicate abstraction [32, 25] is one of the most popular and widely applied methods for the systematic abstraction of systems. It maps concrete data types to abstract data types through predicates over the concrete data. However, the computational cost of the predicate abstraction procedure may be too high, making generation of a full set of predicates for a large system infeasible. In practice, the number of computed predicates is bounded [8, 12], and model checking is guaranteed to deliver sound results within this bound. The bound limit is increased when errors (if any) are found within the bound and fixed. Moreover, in many cases, software systems are first rendered finite by restricting variables to finite domains and then abstraction techniques are used to obtain smaller models.

The abstract program is created using existential abstraction [19]. This method defines the transition relation of the abstract program so it is guaranteed to be a conservative over-approximation of the original program, with respect to the set of given predicates. This ensures that if a claim holds for the abstract over-approximate system, it must also hold for the original system. The use of a conservative abstraction, as opposed to an exact abstraction, produces considerable reductions in the state space. The drawback of the conservative abstraction is that when model checking of the abstract program fails, it may produce a counterexample that does not correspond to a concrete counterexample. Such a counterexample is usually called *spurious* [18]. When a spurious counterexample is encountered, the abstract model is refined (e.g., by adjusting the set of predicates) such that the counterexample is eliminated.

2.2.2 Counterexample-Guided Abstraction Refinement (CEGAR). The abstraction refinement process has been automated by the CEGAR paradigm [42, 18, 9, 26]. The CEGAR framework is shown in Figure 2.2.2: one starts by computing a coarse abstraction (for example, an abstraction of a C program) and model checking it. If an error trace reported by the model checker is spurious, the error trace is used to refine the abstract program, and the process repeated until no spurious error traces can be found. In short, the CEGAR framework consists of the abstract-verify-refine steps and the actual details vary based on the abstraction and refinement techniques used.

The steps of the CEGAR procedure are described below in the context of predicate abstraction.

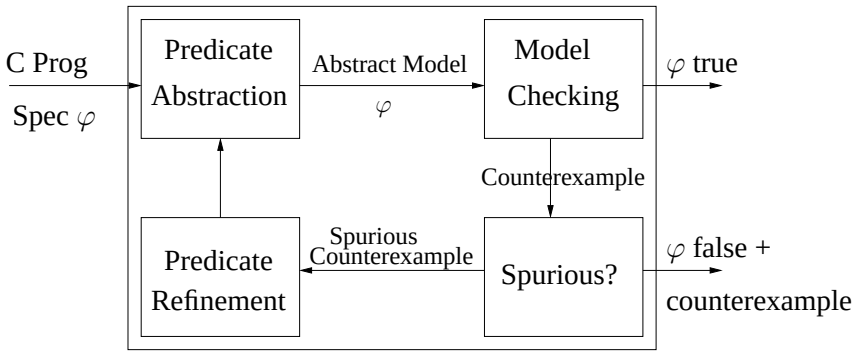


Fig. 2 The CEGAR Framework

1. **Program abstraction:** Given a set of predicates, a finite-state model is extracted from the code of a software system, and the abstract transition system is constructed.
2. **Verification:** A model checking algorithm is run to check whether the model created by applying predicate abstraction satisfies the desired behavioral claim φ. If the claim holds, the model checker reports success (φ is *true*), and the CEGAR loop terminates. Otherwise, the model checker extracts a counterexample, and the computation proceeds to the next step.
3. **Counterexample validation:** The counterexample is examined to determine whether it is spurious. This examination is done by simulating the (concrete) program using the abstract counterexample as a guide, to find out if the counterexample represents an actual program behavior. If this is the case, the bug is reported (φ is *false*), and the CEGAR loop terminates. Otherwise, the CEGAR loop proceeds to the next step.
4. **Predicate refinement:** The set of predicates is changed to eliminate the detected spurious counterexample and possibly other spurious behaviors introduced by predicate abstraction. Given the updated set of predicates, the CEGAR loop proceeds to Step 1.

The efficiency of this process depends on the efficiency of the program abstraction, verification and predicate refinement procedures. While program abstraction focuses on constructing the transition relation of the abstract program, the focus of predicate refinement is to define efficient techniques for choosing the set of predicates in a way that eliminates spurious counterexamples.

2.2.3 Compositional Reasoning. Compositional reasoning [21,42,4,46,28] allows model checking to scale to large systems by using a "divide and conquer" approach that exploits the modular structure of hardware and software systems. More specifically, the verification claim for a system is first decomposed into a set of local claims, one for each system module. These local claims are then verified separately. The compositional approach establishes whether for given systems M_1 and M_2 and a claim T, the composed system satisfies T (written $M_1 \parallel M_2 \models T$). A naive compositional approach proceeds by executing the following steps: (1) $M_1 \models T$ and (2) $M_2 \models T$ and concludes by proving that $M_1 \parallel M_2 \models T$. Although this rule is sound in theory, it is often not useful in practice. Usually, both M_1 and M_2 behave like T only under a suitable environment. To solve this problem, the compositional principle can be strengthened to an *assume-guarantee* principle [47,41,49,1]: in order to check $M \models T$, it suffices to check that both $M_1 \parallel A \models T$ and $M_2 \models A$ hold. This technique uses a local specification A as the constraining environment (also called an *assumption*) for M_1. In general, for a system composed of multiple modules, assume-guarantee reasoning tries to prove that each system component M_i satisfies a corresponding specification component T_i under a suitable constraining environment A_i and that the environment indeed satisfies the constraint A_i. Recently, an approach was proposed by Cobleigh et al. [23] to automate assume-guarantee reasoning with the help of using learning algorithms for regular languages to compute these environment assumptions. The proposed compatibility check is based on this automated assume-guarantee reasoning procedure.

3 Notation and Background

In this section we present some basic definitions. Let X be a sequence. Let $\bullet$ denote the concatenation operator over *sequences*, and let X^* denote zero or more applications of $\bullet$ over X as usual. As a special case, the empty sequence λ denotes zero applications of $\bullet$ over X. For any two sets X and Y, we denote the set $\{x \bullet y \mid x \in X \wedge y \in Y\}$ by $X \bullet Y$. In the following, we use the terms sequence and *trace* interchangeably.

Definition 1 (Finite Automaton) *A finite automaton (FA) is a 5-tuple $(Q, Init, \Sigma, T, F)$ where (i) Q is a finite set of states, (ii) $Init \subseteq Q$ is the set of initial states, (iii) Σ is a finite alphabet of actions, (iv) $T \subseteq Q \times \Sigma \times Q$ is the transition relation, and (v) $F \subseteq Q$ is a set of accepting states.*

For any FA $M = (Q, \text{Init}, \Sigma, T, F)$, we write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in T$. We define the function δ as follows: $\forall \alpha \in \Sigma \,.\, \forall s \in Q \,.\, \delta(\alpha, s) = \{s' | s \xrightarrow{\alpha} s'\}$. We extend δ to operate on strings and sets of states in the natural manner: for any $\sigma \in \Sigma^*$ and $Q' \subseteq Q$, $\delta(\sigma, Q')$ denotes the set of states of M reached by simulating σ on M starting from any $s \in Q'$.

The *language* accepted by a FA M, denoted by $L(M)$, is defined as follows: $L(M) = \{\sigma \in \Sigma^* \mid \delta(\sigma, \text{Init}) \cap F \neq \emptyset\}$. An element of $L(M)$ is said to be a trace of M.

Definition 2 (Deterministic and Complete Finite Automaton) *A FA $M = (Q, \mathit{Init}, \Sigma, T, F)$ is said to be a deterministic FA, or DFA, if $|\mathit{Init}| = 1$ and $\forall \alpha \in \Sigma \,.\, \forall s \in Q \,.\, |\delta(\alpha, s)| \leq 1$. Also, M is said to be complete if $\forall \alpha \in \Sigma \,.\, \forall s \in Q \,.\, |\delta(\alpha, s)| \geq 1$.*

Thus, for a complete DFA, we have the following: $\forall \alpha \in \Sigma \,.\, \forall s \in Q \,.\, |\delta(\alpha, s)| = 1$. Unless otherwise mentioned, all DFA we consider in the rest of this paper are also complete. It is well-known that a language is regular if and only if it is accepted by some FA (or DFA, since FA and DFA have the same accepting power). Also, every regular language is accepted by a unique (up to isomorphism) minimum DFA. Given any FA M, its complement $\overline{M}$ is defined to be $\overline{M'}$ where M' is the DFA obtained from M by the subset construction [39].

We now define a notion of asynchronous parallel composition between FAs which is based on the notion of composition defined for CSP [51].

Definition 3 (Parallel Composition) *Given two FA $M_1 = (Q_1, \mathit{Init}_1, \Sigma_1, T_1, F_1)$ and $M_2 = (Q_2, \mathit{Init}_2, \Sigma_2, T_2, F_2)$, their parallel composition $M_1 \parallel M_2$ is the FA $(Q_1 \times Q_2, \mathit{Init}_1 \times \mathit{Init}_2, \Sigma_1 \cup \Sigma_2, T, F_1 \times F_2)$ such that $\forall s_1, s_1' \in Q_1 \,.\, \forall s_2, s_2' \in Q_2$, $(s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')$ if and only if :*

(a) $\alpha \in \Sigma_1 \wedge \alpha \notin \Sigma_2 \wedge s_1 \xrightarrow{\alpha} s_1' \wedge (s_2 = s_2')$ *or,*

(b) $\alpha \in \Sigma_2 \wedge \alpha \notin \Sigma_1 \wedge s_2 \xrightarrow{\alpha} s_2' \wedge (s_1 = s_1')$ *or,*

(c) $\alpha \in (\Sigma_1 \cap \Sigma_2) \wedge \forall i \in \{1, 2\}\ s_i \xrightarrow{\alpha} s_i'$.

Given a string t, we write $M \parallel t$ to denote the composition of M with the automaton representation of t.

Definition 4 (Language Containment) *For any FA M_1 and M_2 (with alphabets Σ_1 and Σ_2 respectively, where $\Sigma_2 \subseteq \Sigma_1$), we write $M_1 \preccurlyeq M_2$ to mean $L(M_1 \parallel \overline{M_2}) = \emptyset$. A counterexample to $M_1 \preccurlyeq M_2$ is a string $\sigma \in L(M_1 \parallel \overline{M_2})$.*

If $M_1 \preccurlyeq M_2$, then we sometimes also say that M_2 is an abstraction of M_1. We now define the notion of weakest assumptions. We assume that a safety property can be represented as a FA in the usual way [31].

Definition 5 (Weakest Assumption [31]) *For any FA M and any safety property expressed as a FA φ, the weakest (i.e., maximal w.r.t. the language-containment preorder $\preccurlyeq$) assumption FA, denoted by WA, is defined as follows: (i) $M \parallel WA \preccurlyeq \varphi$ and (ii) for any FA E, $M \parallel E \preccurlyeq \varphi$ iff $E \preccurlyeq WA$.*

Lemma 1 (Existence and Uniqueness of Weakest Assumption) *Given a FA M and a property FA φ, the weakest assumption WA exists and can be represented by a FA accepting the language $L(\overline{M \parallel \overline{\varphi}})$.*

Proof. The proof follows from the construction given in [31].

The following lemma shows that we can check if a given trace t is in $L(WA)$ without constructing WA directly by checking if $M \parallel t \preccurlyeq \varphi$ holds.

Lemma 2 *Let WA be the weakest assumption automata for component automaton M and specification automaton φ. Given a trace t, if $M \parallel t \preccurlyeq \varphi$, then $t \in L(WA)$.*

Proof. It follows from the definition of weakest assumptions (cf. Definition 5) that for all assumptions such that $M \parallel A \preccurlyeq \varphi$, $L(A) \subseteq L(WA)$ holds. Let M_t be the automaton representation of t. Since $M \parallel M_t \preccurlyeq \varphi$ holds, it follows that $L(M_t) \subseteq L(WA)$. Since $L(M_t) = \{t\}$, $t \in L(WA)$.

3.1 L^* algorithm

The L^* algorithm for learning DFAs was developed by Angluin [6] and later improved by Rivest and Schapire [50]. In this paper, we use the improved version of L^* due to Rivest and Schapire. The algorithm learns an unknown regular language U, over an alphabet Σ, by generating the minimum DFA that accepts U.

3.1.1 Preliminaries. Let U be an unknown regular language over some alphabet Σ.

Definition 6 (Prefix Closed) *A set $X \subseteq \Sigma^*$ is said to prefix-closed if for each $x \in X$, all the prefixes of x are also in X.*

In the rest of this section, we present the core L^* algorithm. We begin with the notion of a minimally adequate teacher.

3.1.2 Minimally Adequate Teacher. In order to learn an unknown language U, L^* interacts with an oracle, also known as a *minimally adequate teacher*, MAT, for U. The teacher can provide answers to the following two kinds of queries:

1. *Membership*. Given a trace $\sigma \in \Sigma^*$, MAT returns TRUE iff $\sigma \in L(U)$.
2. *Candidate*. Given a DFA D, MAT returns TRUE iff $L(D) = U$. If MAT returns FALSE, it also returns a counterexample trace w which either lies in $L(D) \setminus U$ or $U \setminus L(D)$.

3.1.3 Observation Table. The L^* algorithm constructs iteratively a minimal DFA D such that $L(D) = U$. To this goal, it maintains an observation table data structure $\mathcal{T} = (S, E, T)$, where:

– $S \subseteq \Sigma^*$ is a prefix-closed set of traces,
– $E \subseteq \Sigma^*$ is a set of experiment traces, used to distinguish states in D, and,
– $T : (S \cup (S \bullet \Sigma)) \times E \rightarrow \{0, 1\}$ is a function such that:

$$\forall s \in (S \cup (S \bullet \Sigma)) \,.\, \forall e \in E \,.\, T(s, e) = 1 \equiv s \bullet e \in U$$

Intuitively, one can think of $\mathcal{T}$ as a two-dimensional table. The rows of $\mathcal{T}$ are labeled with the elements of $S \cup (S \bullet \Sigma)$ while the columns are labeled with elements of E. Finally T denotes the table entries. In other words, the entry corresponding to row s and column e is simply $T(s, e)$. The value of $T(s, e)$ is 1 if $s \bullet e \in U$, otherwise $T(s, e)$ is 0. Figure 5(left) shows an example of an observation table.

Table Congruence. We define an equivalence relation $\equiv$ as follows: for $s, s' \in (S \cup S \bullet \Sigma)$, $s \equiv s'$ iff $\forall e \in E$, $T(s, e) = T(s', e)$. Also, for all $s \in (S \cup S \bullet \Sigma)$, we denote the set of traces equivalent to s by $[s]$, where

$$[s] = \{s' \in (S \cup S \bullet \Sigma) \mid s \equiv s'\}$$

.

Well-formed Table. An observation table $\mathcal{T}$ is said to be *well-formed* if for all s, $s' \in S$, $s \not\equiv s'$. The L^* algorithm always keeps $\mathcal{T}$ well-formed[1].

[1] We omit the notion of *consistency* usually used while presenting of L^* [7] since a well-formed table is consistent by definition.

Algorithm CloseTable

1: **forever do**

2: **if** ($\forall t \in S \bullet \Sigma$ **.** $\exists s \in S$ **.** $s \equiv t$) **return**;

3: **find** $t \in S \bullet \Sigma$ **such that** $\forall s \in S$ **.** $s \not\equiv t$;

4: $S := S \cup \{t\}$;

5: Update T using membership queries;

Fig. 3 Pseudo-code for algorithm **CloseTable**.

Table Closure. The observation table $\mathcal{T}$ is said to be closed if for each $t \in S \bullet \Sigma$, there is a $s' \in S$, so that $t \equiv s'$. Given any observation table $\mathcal{T}$, we assume that a procedure **CloseTable** makes it closed. Figure 3.1.3 shows the pseudo-code for the procedure. In words, the procedure iteratively selects some $t \in S \bullet \Sigma$ so that for all $s \in S$, $s \not\equiv t$. Then, it adds t to S and updates the function T by asking membership queries for extensions of t on each alphabet symbol. The procedure **CloseTable** terminates with a closed table when no such t can be found. In each iteration the size of S increases by one. Lemma 5 (described below) shows that the procedure **CloseTable** cannot increase the size of S indefinitely and must terminate in finite number of steps.

DFA Construction. Given a closed table $\mathcal{T}$, L^* obtains a DFA $D = \langle Q, q_0, \Sigma, \Delta, F\rangle$, as follows:

- $Q = \{[s] \mid s \in S\}$, where a state $q \in Q$ corresponds to the equivalence class $[s]$ of a trace $s \in S$,
- $q_0 = [\lambda]$,
- $\Delta = \{([s], a, [s \bullet a]) \mid s \in S, a \in \Sigma\}$.
- $F = \{[s] \mid s \in S \wedge T(s, \lambda) = 1\}$.

Suppose that a procedure called **MkDFA** implements this construction. Note that D is both deterministic and complete.

3.1.4 The L^* Algorithm. Figure 4 shows the pseudo-code for the L^* algorithm. Recall that λ denotes any empty sequence. L^* starts with a table $\mathcal{T} = (S, E, T)$ such that $S = E = \{\lambda\}$ and in each iteration proceeds as follows.

Algorithm L^*

1: $S := E := \{\lambda\}$;
2: **forever do**
3: **CloseTable**();
4: M := **MkDFA**($\mathcal{T}$);
5: **if** (**IsCandidate**(M)) $\mathbf{return} M$;
6: **let** CE be the counterexample returned by **IsCandidate**;
7: Obtain a *distinguishing suffix* e from CE
8: $E := E \cup \{e\}$;

Fig. 4 Pseudo-code for algorithm L^*.

1. It first updates $\mathcal{T}$ using the **CloseTable** procedure until $\mathcal{T}$ is closed.
2. Next L^* builds a candidate DFA D from the closed table (using **MkDFA** procedure) and makes a candidate query with D.
3. If the MAT returns TRUE to the candidate query, L^* returns D as the result and stops.
4. Otherwise, a counterexample CE is obtained. Now L^* constructs a new experiment e from CE using the algorithm proposed by Rivest and Schapire [50] and adds e to E. The new experiment e (also known as a *distinguishing suffix*) has the property that it causes the observation table $\mathcal{T}$ to be no longer closed, and thereby forces the number of rows of $\mathcal{T}$ to increase strictly in the next iteration of L^*.

Example 1 Consider Figure 5. On the left is an observation table (S, E, T) where S and E correspond to rows and columns respectively and T corresponds to the table entries. Here, $\Sigma = \{\alpha, \beta\}$. From this table we see that $\{\alpha, \alpha \bullet \alpha\} \subseteq U$. On the right is the corresponding candidate DFA. The states s_0 and s_1 of the DFA correspond to the elements λ and α of S respectively. The state s_0 is marked initial since it corresponds to word λ. The state s_1 is marked final since the table entry $T(\alpha, \lambda) = 1$. Finally, the transitions are determined as described in the procedure **MkDFA**. □

		E	
		λ	
S	λ	0	(s_0)
	α	1	(s_1)
$S \bullet \Sigma$	β	0	
	$\alpha\alpha$	1	
	$\alpha\beta$	0	

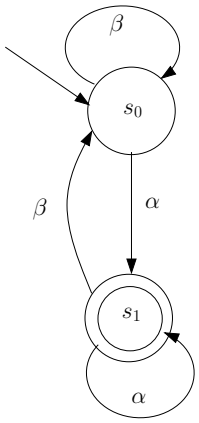


Fig. 5 An Observation Table and the Corresponding Candidate DFA

3.1.5 Results on L^*. In order to make our presentation more self-contained, we now prove some results about the L^* algorithm and the procedures **CloseTable** and **MkDFA**. We use these results to prove the correctness of a new *dynamic* version of the L^* algorithm that we propose later in this paper (cf. Section 5.1.1).

Lemma 3 *The L^* algorithm always maintains a well-formed table.*

Proof. Consider the pseudo-code of L^* in Figure 4. Given an observation table $\mathcal{T} = (S, E, T)$, the set S is updated only in line 3 by the **CloseTable** procedure. Hence, we need to show that **CloseTable** always maintains a well-formed table at each iteration.

We proceed by induction. Note that at the first iteration of the L^* loop (first call to **CloseTable**), S only has a single element and hence the table is well-formed. Assume that the input observation table $\mathcal{T}$ to **CloseTable** is well-formed at k^{th} iteration ($k > 1$). The procedure **CloseTable** (cf. Figure 3.1.3) only adds a new element t to S (line 4) if for all $s \in S$, $s \not\equiv t$ (line 3). Therefore, all the elements in $S \cup \{t\}$ are non-equivalent and the resultant table is also well-formed.

□

The following lemma is crucial for proving termination of L^*. It essentially provides an upper bound on the size of S.

Lemma 4 *Let $\mathcal{T} = (S, E, T)$ be a well-formed observation table. Let U be an unknown regular language and n be the number of states in the minimum DFA M such that $L(M) = U$. Then the size of the trace set S cannot exceed n.*

Proof. Let δ denote the transition relation of M (Δ extended to words) (cf. procedure **MkDFA**) and q_0 denote the initial state for M.

The proof is by contradiction. Suppose that the size of S exceeds n. Then by the pigeon-hole principle, there exist two elements s_1 and s_2 of S such that $\delta(s_1, \{q_0\}) = \delta(s_2, \{q_0\})$ = q (say), i.e., s_1 and s_2 must reach the same state q in M. Since M is the minimum DFA for U, we know that the states of M correspond to equivalence classes of the Nerode congruence [39] for U. Since s_1 and s_2 reach the same state in M (same Nerode equivalence class), it follows that

$$\forall e \in \Sigma^*, s_1 \bullet e \in U \text{ iff } s_2 \bullet e \in U \tag{1}$$

But, $\mathcal{T}$ is well-formed and hence $s_1 \not\equiv s_2$. Therefore there exists some $e \in E$, such that $T(s_1 \bullet e) \neq T(s_2 \bullet e)$, i.e., $s_1 \bullet e \in U$ and $s_2 \bullet e \notin U$ or vice versa. Together with (1), we reach a contradiction.

□

The following lemma shows that the procedure **CloseTable** cannot increase the size of S indefinitely and must terminate in finite number of steps.

Lemma 5 *The procedure* **CloseTable** *always terminates with a closed table. Moreover, the procedure maintains a well-formed observation table $\mathcal{T} = (S, E, T)$ at each iteration.*

Proof. It follows from the pseudo-code (Figure 3.1.3) that the procedure **CloseTable** keeps adding new elements to S until $\mathcal{T}$ is closed. Since the size of S is bounded by the number of states in the minimum DFA for the unknown language U (Lemma 4), **CloseTable** terminates with a closed table in finite number of steps. It follows from Lemma 3 that the procedure always maintains a well-formed table.

□

The following lemma shows that the **MkDFA** procedure always constructs a candidate DFA starting from a well-formed and closed observation table.

Lemma 6 *Given a well-formed and closed observation table as an input, the procedure* **MkDFA** *always terminates with a candidate DFA D as a result.*

Proof. Since the input table $\mathcal{T}$ is well-formed, the states of D are uniquely defined by the elements of S. Moreover, the initial state is unique by definition and the final states are well-defined. Since the table is closed, the transition relation of D is also well-defined. Hence, the candidate DFA is well-defined.

□

Theorem 1 *Given an unknown regular language U, the algorithm L^* always terminates with a DFA M such that $L(M) = U$.*

Proof. The fact that L^* algorithm terminates with the correct result M ($L(M) = U$) is obvious since it stops only after a candidate query has passed. To prove that it terminates, it suffices to show that there can only be a finite number of failed candidate queries and therefore only a finite number of iterations of the top-level loop (Figure 4, line 2). It has been shown [50] that for each failed candidate query, the procedure **CloseTable** must add at least one element to S in the next iteration of the top-level loop. However, the size of S is bounded (cf. Lemma 4) and hence the loop executes only a finite number of times.

□

4 Containment Analysis

Recall that the containment step verifies for each $i \in \mathcal{I}$, that $C_i \preccurlyeq C_i^{'}$, i.e., every behavior of C_i is also a behavior of $C_i^{'}$. If $C_i \not\preccurlyeq C_i^{'}$, we also generate a counterexample behavior in $Behv(C_i) \setminus Behv(C_i^{'})$ which is subsequently provided as user feedback. This containment check is performed as depicted in Figure 6 for each modified component. (CE refers to the counterexample generated during the verification phase). For each $i \in \mathcal{I}$, the containment check proceeds as follows:

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

$$(\mathbf{C1})\ C_i \preccurlyeq M \qquad (\mathbf{C2})\ M' \preccurlyeq C_i^{'} \tag{2}$$

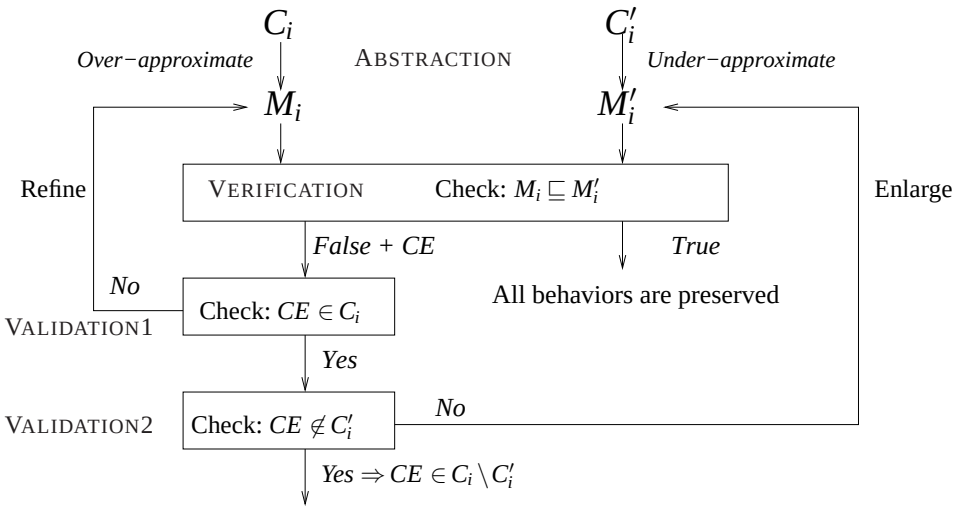


Fig. 6 The containment phase of the substitutability framework.

Here M is an *over-approximation* of C_i and can be constructed by standard predicate abstraction [33]. M' is constructed from C_i' via a modified predicate abstraction which produces an *under-approximation* of its input C component. We now describe the details of the abstraction steps.

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

Predicates are functions that map a concrete state $\bar{v} \in S$ into a Boolean value. Let $\mathcal{P} = \{\pi_1, \ldots, \pi_k\}$ be the set of predicates over the given program. On evaluating the set of predicates in $\mathcal{P}$ in a particular concrete state $\bar{v}$, we obtain a vector of Boolean values $\bar{b}$, where $\bar{b}[i] = \pi_i(\bar{v})$. The Boolean vector $\bar{b}$ represents an abstract state. We represent this predicate evaluation using an abstraction function α: $\bar{b} = \alpha(\bar{v})$. Also, the *concretization* function γ is defined as follows:

$$\gamma(\bar{b}) = \{\bar{v} \mid \bar{b} = \alpha(\bar{v})\}$$

"May" Predicate Abstraction: Over-approximation. This step corresponds to the standard predicate abstraction [12]. Each statement (or basic block) St in C_i is associated with a transition relation $T(\bar{v}, \bar{v}')$. Here, $\bar{v}$ and $\bar{v}'$ represent a concrete state before and after execution of St, respectively. Given the set of predicates $\mathcal{P}$ and associated

vector of Boolean variables $\bar{b}$ as before, we compute an abstract transition relation $\hat{T}(\bar{b}, \bar{b}')$ [19] as follows:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \exists \bar{v}, \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{b} = \alpha(\bar{v}) \wedge \bar{b}' = \alpha(\bar{v}') \tag{3}$$

$\hat{T}$ is the existential abstraction [19] of T (with respect to the abstraction function α) and is also referred to as its *may* abstraction $\hat{T}_{may}$ [52]. In practice, we compute this abstraction using the weakest precondition (WP) transformer [29] on predicates in $\mathcal{P}$ along with an automated theorem prover [33] as follows:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \gamma(\bar{b}) \wedge \mathit{WP}(St, \gamma(\bar{b}')) \text{ is satisfiable} \tag{4}$$

where $\mathit{WP}(St, \phi)$ denotes the weakest precondition expression for formula ϕ with respect to statement St and γ is the concretization function as defined above. By the definition of weakest preconditions, we have

$$\bar{v} \in \mathit{WP}(St, \gamma(\bar{b}')) \equiv \exists \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{v}' \in \gamma(\bar{b}')$$

Note that Equation 3 is equivalent to Equation 4 since:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \exists \bar{v} \,.\, \exists \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{b} = \alpha(\bar{v}) \wedge \bar{b}' = \alpha(\bar{v}')$$

$$\equiv \exists \bar{v} \,.\, (\bar{v} \in \gamma(\bar{b}) \wedge \exists \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{v}' \in \gamma(\bar{b}'))$$

$$\equiv \exists \bar{v} \,.\, (\bar{v} \in \gamma(\bar{b}) \wedge \bar{v} \in \mathit{WP}(St, \gamma(\bar{b}')))$$

$$\equiv \gamma(\bar{b}) \wedge \mathit{WP}(St, \gamma(\bar{b}')) \text{ is satisfiable}$$

Note that even though we are checking software consisting of several communicating program components, it is sufficient to use standard weakest preconditions for sequential programs, since the abstraction is performed component-wise.

"Must" Predicate Abstraction: Under-approximation. The modified predicate abstraction constructs an under-approximation of the concrete system via universal or *must* [52] abstraction. Given a statement St in the modified component C_i' and its associated transition relation $T(\bar{v}, \bar{v}')$ as before, we compute its must abstraction with respect to predicates $\mathcal{P}$ as follows:

$$\hat{T}(\bar{b}, \bar{b}') \equiv \forall \bar{v} \,.\, \bar{b} = \alpha(\bar{v}) \implies \exists \bar{v}' \,.\, T(\bar{v}, \bar{v}') \wedge \bar{b}' = \alpha(\bar{v}') \tag{5}$$

We use $\hat{T}_{must}$ to denote the above relation. Note that $\hat{T}_{must}$ contains a transition from an abstract state $\bar{b}$ to $\bar{b}'$ iff for every concrete state $\bar{v}$ corresponding to $\bar{b}$, there exists a concrete transition to a state $\bar{v}'$ corresponding to $\bar{b}'$ [52]. Further, it has been shown [52] that the concrete transition relation T simulates the abstract transition relation $\hat{T}_{must}$. Hence, $\hat{T}_{must}$ is an under-approximation of T. Again, in practice, we compute $\hat{T}_{must}$ using the WP transformer on the predicates together with a theorem prover [36] in the following way:

$$\hat{T}(\bar{b}, \bar{b}') \equiv (\gamma(\bar{b}) \implies \mathit{WP}(St, \gamma(\bar{b}'))) \tag{6}$$

Note that Equation 5 is equivalent to Equation 6 since:

$$\hat{T}(\bar{b}, \bar{b}') \equiv (\forall \bar{v} \centerdot \bar{b} = \alpha(\bar{v}) \implies \exists \bar{v}' \centerdot T(\bar{v}, \bar{v}') \wedge \bar{b}' = \alpha(\bar{v}'))$$

$$\equiv (\forall \bar{v} \centerdot \bar{v} \in \gamma(\bar{b}) \implies \exists \bar{v}' \centerdot T(\bar{v}, \bar{v}') \wedge \bar{v}' \in \gamma(\bar{b}'))$$

$$\equiv (\forall \bar{v} \centerdot \bar{v} \in \gamma(\bar{b}) \implies \bar{v} \in \mathit{WP}(St, \gamma(\bar{b}')))$$

$$\equiv (\gamma(\bar{b}) \implies \mathit{WP}(St, \gamma(\bar{b}')))$$

At the end of the abstraction phase, we obtain M as an over-approximation of C_i and M' as an under-approximation of C_i', as defined in Equation 2. The containment check now proceeds to the next stage involving verification.

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

3. Validation and Refinement 1. Check that CE is a real behavior of C_i. This step is done in a manner similar to the counterexample validation techniques employed in software model checkers based on CEGAR [8, 38, 12]. If CE is a real behavior of C_i, we proceed to Step 4. Otherwise we refine model M (i.e., remove the spurious CE) by constructing a new set of predicates $\mathcal{P}'$ and repeat from Step 2. The procedure for refining the model M has been presented elsewhere [12] in detail, and we do not describe it here further.

4. Validation and Refinement 2. Check that CE is *not* a real behavior of C_i'. The operations involved in this check are the same as those used for the validation check in Step 3. The only difference is that we complement the

final result, since in this step we are interested in checking whether CE **is not** a real behavior of $C_i^{'}$, while in Step 3, we were interested in checking whether CE **is** a real behavior of C_i.

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 the user feedback step and stop. Otherwise we enlarge M' (i.e., add CE) by constructing a new set of predicates $\mathcal{P}'$ and repeat from Step 2. The procedure for enlarging the model M' has been presented elsewhere [36] in detail, and we do not describe it here further.

Figure 6 depicts the individual steps of this containment check. Similar to ordinary abstraction-refinement procedures for programs, the containment check may not terminate because a sufficient set of predicates is never found. Otherwise, the check terminates either with a successful result (all behaviors of C_i are verified to be present in $C_i^{'}$) or returns an actual diagnostic behavior CE as feedback to the developers. The following theorem proves this result.

Theorem 2 (Correctness of Containment Check) *Upon termination, if the Containment Check is successful, then* $C_i \preccurlyeq C_i^{'}$ *holds. Otherwise, a witness counterexample* $CE \in C_i \setminus C_i^{'}$ *is returned.*

Proof. The containment check terminates either when the verification check (Step 2) succeeds or both the Validation and Refinement checks (Steps 3 and 4) fail. Note that at each iteration $C_i \preccurlyeq M_i$ and $M_i' \preccurlyeq C_i^{'}$. If the verification step (Step 2) succeeds, then it follows that $M_i \preccurlyeq M_i'$, and hence $C_i \preccurlyeq M_i \preccurlyeq M_i' \preccurlyeq C_i^{'}$. Therefore, $C_i \preccurlyeq C_i^{'}$ holds. Otherwise, suppose that both the Validation and Refinement phases (Steps 3 and 4) fail. Then, from Step 3 we know that $CE \in C_i$, and from Step 4 we know that $CE \notin C_i^{'}$. Hence, we have a counterexample $CE \in C_i \setminus \in C_i^{'}$ which is returned by the containment check.

□

4.1 Feedback

Recall that for some $i \in \mathcal{I}$, if our containment check detects that $C_i \not\preccurlyeq C_i^{'}$, it also computes a set $\mathcal{F}_i$ of erroneous behaviors. Intuitively, each element of $\mathcal{F}_i$ represents a behavior of C_i that 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 write $C, C^{'}$, and $\mathcal{F}$ to mean C_i, $C_i^{'}$, and $\mathcal{F}_i$, respectively.

Consider any behavior π in $\mathcal{F}$. Recall that π is a trace of an automaton M obtained by predicate abstraction of C. By simulating π on M, we construct a sequence $Rep(\pi) = \langle \alpha_1, \ldots, \alpha_n \rangle$ of states and actions of M corresponding to the trace π.

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

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

5 Compatibility Analysis

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 the 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 described in Section 3. In this section we first present a dynamic version of the L^* learning algorithm and then describe how it can be applied for checking compatibility.

5.1.1 Dynamic L^*. Normally L^* initializes with $S = E = \{\lambda\}$. 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 discussion, we show that if L^* begins with any non-empty valid table, it must terminate with the correct result (Theorem 3). In particular, this theorem allows us to perform our compatibility check dynamically by restarting L^* with any previously computed table by revalidating it instead of starting from an empty table.[2]

Definition 7 (Agreement) *An observation table $\mathcal{T} = (S, E, T)$ is said to agree with a regular language U iff:*

$$\forall (s, e) \in (S \cup S \bullet \Sigma) \times E \ . \ T(s, e) = 1 \equiv s \bullet e \in U$$

Definition 8 (Validity) *Recall the notion of a well-formed observation table from Section 3.1.3. An observation table $\mathcal{T} = (S, E, T)$ is said to be valid for a language U iff $\mathcal{T}$ is well-formed and agrees with U. Moreover, we say that a candidate automaton derived from a table $\mathcal{T}$ is valid for a language U if $\mathcal{T}$ is valid for U.*

Theorem 3 *L^* terminates with a correct result for any unknown language U starting from any valid table for U.*

Proof. It was shown earlier (cf. Theorem 1) that for a given unknown language U, the L^* algorithm terminates if it is able to perform a finite number of candidate queries. Therefore, it remains to show that starting from a valid observation table, the algorithm must be able to perform a candidate query in a finite number of steps. Note that each iteration of the L^* algorithm involves executing the **CloseTable** and **MkDFA** procedures before making a candidate query (cf. Figure 4). Therefore, we need to show that the procedures **CloseTable** and **MkDFA** terminate in a finite number of steps starting from a valid table.

Let the valid observation table be $\mathcal{T}_1$. Since $\mathcal{T}_1$ agrees with U, the **CloseTable** procedure terminates in a finite number of steps with a closed table $\mathcal{T}_2$ (cf. Lemma 5). Moreover, $\mathcal{T}_2$ is well-formed since the initial table $\mathcal{T}_1$ is well-formed (cf. Lemma 5). Since $\mathcal{T}_2$ is well-formed and closed, the **MkDFA** algorithm is able to compute a DFA candidate D (cf. Lemma 6) from $\mathcal{T}_2$ and terminates. Therefore, after the execution of **MkDFA** finishes, L^* must perform a candidate query.

□

Suppose we have a table $\mathcal{T}$ that 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 since U and U' differ in general,

[2] A similar idea was also proposed in the context of adaptive model checking [34].

$\mathcal{T}$ may not agree with U' and hence may not be valid with respect to U'; hence, L^* may not terminate starting from $\mathcal{T}$. Thus, we first *revalidate* $\mathcal{T}$ against U' and then start L^* from the valid $\mathcal{T}$. Theorem 3 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 (see Section 5.3).

In the context of assume-guarantee reasoning, U represents a weakest assumption language. When an upgrade occurs, U may change to a different language U'. However, since the change was caused by an upgrade, we expect that the language U' will differ from U only slightly. We will see that the efficiency of our revalidation procedure depends crucially on this hypothesis.

Revalidation Procedure. Suppose we have a table $\mathcal{T}$ which is valid for an unknown language U. Given a Teacher for a different unknown language U', the table revalidation procedure **Reval** (shown in Figure 7) makes $\mathcal{T}$ valid with respect to U' by executing the following two steps. In Step 1, **Reval** updates all the table entries in $\mathcal{T}$ by asking membership queries. The table $\mathcal{T}'$ obtained as a result may not be well-formed since the function T is updated. More precisely, for some $s_1, s_2 \in S$ where $s_1 \not\equiv s_2$ in $\mathcal{T}$, it may happen that $s_1 \equiv s_2$ in $\mathcal{T}'$. However, the construction of a candidate DFA requires that the observation table be well-formed (cf. Lemma 6). Therefore, in Step 2, **Reval** uses the procedure **MkWellFormed** to make $\mathcal{T}'$ well-formed. In order to describe **MkWellFormed**, we need the concepts of the *well-formed cover* and the *experiment cover* for an observation table $\mathcal{T}$.

Procedure Reval

Input: An observation table $\mathcal{T} = (S, E, T)$ and a teacher for a language U'.

Output: An observation table $\mathcal{T}'$ that is valid for U'.

1. **(Step 1)** For all $s \in S$ and $e \in E$, ask membership query for $s \bullet e$ with respect to U' and update T.
   Let the table obtained as a result be $\mathcal{T}'$.
2. **(Step 2)** Make $\mathcal{T}'$ well-formed (cf. Section 3.1.3) by using the procedure **MkWellFormed**.

Fig. 7 The table revalidation procedure **Reval**.

Definition 9 (Well-formed Cover) *Given a prefix-closed set S, a well-formed subset of S is a set $S' \subseteq S$ such that (i) S' is prefix-closed, and (ii) for all $s_1, s_2 \in S'$, $s_1 \not\equiv s_2$ holds. A well-formed cover S' of S is a maximal well-formed subset of S.*

Given a prefix-closed set S, a well-formed cover S' of S can be obtained by performing a depth-first tree search on the tree representation of S in the following way: for each newly visited node in the tree, the corresponding string in S is added to S'. However, a node (with the corresponding string s) is visited only if for all s' in the current cover S', s and s' are non-equivalent, i.e., $s \not\equiv s'$. The search terminates when for every $s \in S$ there exists some $s' \in S'$ so that $s \equiv s'$. Note that the final S' obtained in this way is prefix-closed and no two elements of S' are equivalent. For example, let $S = \{a, a \bullet b, a \bullet c, d\}$ where $a \equiv a \bullet c$ and $d \equiv a \bullet b$. A well-formed cover of S is $S' = \{a, a \bullet b\}$. Note that S' is prefix-closed and $a \not\equiv a \bullet b$.

Definition 10 (Column Function) *Given an observation table $\mathcal{T} = (S, E, T)$, and some $e \in E$, $Col(e)$ is defined to be a function from $(S \cup S \bullet \Sigma)$ to $\{0, 1\}$ such that $Col(e)(s) = T(s, e)$ for all $s \in (S \cup S \bullet \Sigma)$. For $e_1, e_2 \in E$, we say that $Col(e_1) = Col(e_2)$ if for all $s \in (S \cup S \bullet \Sigma)$, $T(s, e_1) = T(s, e_2)$.*

Intuitively, for an experiment $e \in E$, $Col(e)$ denotes the vector of Boolean values in the column corresponding to e in an observation table $\mathcal{T}$. Two elements e_1 and e_2 are equivalent under the Col function if the vector of Boolean values in the corresponding columns of the observation table are same.

Definition 11 (Experiment Cover) *An experiment cover of E is a set $E' \subseteq E$, such that (i) for all $e_1, e_2 \in E'$, $Col(e_1) \neq Col(e_2)$, and (ii) for each $e \in E$, there exists an $e' \in E'$, such that $Col(e) = Col(e')$.*

An experiment cover for E can be obtained by finding the set of elements equivalent under Col function and picking a representative element from each set. For example, consider the observation table in Figure 8(d). Here, $E = \{\lambda, \alpha\}$. Note that $Col(\lambda) \neq Col(\alpha)$. Hence, the experiment cover E' for E is the same as E.

The **MkWellFormed** procedure is described by the pseudo-code in Figure 9. Intuitively, the procedure removes duplicate elements from S (which are equivalent under the $\equiv$ relation) and E (having the same value under the Col function).

		E
		λ
S	λ	1
	α	0
$S \bullet \Sigma$	β	1
	$\alpha\alpha$	1
	$\alpha\beta$	1

(a)

		E
		λ
S	λ	1
	α	1
$S \bullet \Sigma$	β	1
	$\alpha\alpha$	0
	$\alpha\beta$	1

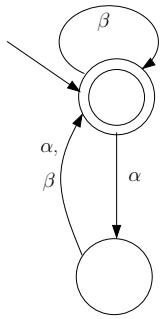
(b)

		E
		λ
S	λ	1
$S \bullet \Sigma$	β	1
	α	1

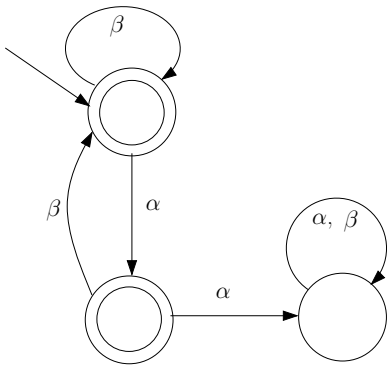
(c)

		E	
		λ	α
S	λ	1	1
	α	1	0
	$\alpha\alpha$	0	0
$S \bullet \Sigma$	β	1	1
	$\alpha\beta$	1	1
	$\alpha\alpha\alpha$	0	0
	$\alpha\alpha\beta$	0	0

(d)



(e)



(f)

Fig. 8 Illustration of the revalidation procedure described in Example 2; (a) Observation table for original language $U = (\beta \mid (\alpha \bullet (\alpha|\beta)))^*$; (b) New observation table after recomputing the entries with respect to the new language $U' = ((\beta \mid \alpha \bullet \beta)^*) \mid ((\beta \mid \alpha \bullet \beta)^* \bullet \alpha)$; e.g., $\alpha \in U'$ implies $T(\alpha, \lambda) = 1$ (c) Observation table after revalidating with respect to U' and (d) after an L^* learning iteration with respect to U'; (e) DFA for language U (corresponding to observation table in (a)); and (f) DFA for language U' (corresponding to table in (d)).

Example 2 **(Revalidation Example)** Figure 8 shows an illustration of the revalidation procedure in the dynamic L^* algorithm. Let the initial unknown language (the weakest assumption language) be $U = (\beta \mid (\alpha \bullet (\alpha|\beta)))^*$. The observation table $\mathcal{T}_1$ and the DFA for U are shown in Figure 8(a) and Figure 8(e) respectively. Suppose that an upgrade happens and the new weakest assumption language is $U' = ((\beta \mid \alpha \bullet \beta)^*) \mid ((\beta \mid \alpha \bullet \beta)^* \bullet \alpha)$. In particular, note that

Procedure MkWellFormed

Input: Observation table $\mathcal{T}$ = (S, E, T)

Output: Well-formed observation table $\mathcal{T}'$ = (S', E', T')

1. Set S' to a well-formed cover (cf. Definition 9) of S.
2. Set E' to an experiment cover (cf. Definition 11) of E with respect to $(S' \cup S' \bullet \Sigma)$.
3. Obtain T' by restricting T to $(S' \cup S' \bullet \Sigma) \times E'$

Fig. 9 Pseudo-code for the **MkWellFormed** procedure

$\alpha \in U'$ but not in U and $\alpha \bullet \alpha \in U$ but not in U'. Our goal is to start learning with respect to U' from the observation table $\mathcal{T}_1$ computed for U previously. So, the **Reval** procedure is applied to $\mathcal{T}_1$. Figure 8(b) shows the table obtained after applying the Step 1 of the revalidation procedure with respect to the new language U'. Note that the entries for $T(\alpha, \lambda)$ and $T(\alpha \bullet \alpha, \lambda)$ are updated with respect to U'. This, in turn, results in $\alpha \equiv \lambda$ (cf. Figure 8(b)). Now, the Step 2 of the **Reval** procedure is applied: since $\alpha \equiv \lambda$ and S = $\{\lambda, \alpha\}$, the well-formed cover $S' = \{\lambda\}$. The experiment cover E' remains the same as E. Hence, α is removed from S during computation of the well-formed cover in this step (Note that the extensions $\alpha \bullet \alpha$ and $\alpha \bullet \beta$ are also in turn removed from $S \bullet \Sigma$). The resultant observation table (after making it closed) is shown in Figure 8(c). Since this table is closed, learning proceeds in the normal fashion from here by computing the next candidate and making a candidate query. Figure 8(d) shows the final observation table and Figure 8(f) shows the DFA obtained after learning completes with respect to U'.

Note that our example is small, and therefore the revalidation step gives rise to a trivial intermediate observation table (Figure 8(b)). However, as noted earlier, in the case when an upgrade causes the change from U to U', the languages U and U' may differ only slightly. Therefore, in this case, the **Reval** procedure may modify the observation table only slightly. In particular, during revalidation, the well-formed cover of S may remain very similar to S (i.e., a large number of elements of S may continue to remain non-equivalent after revalidation), leading to reuse of information about many traces ($S \bullet E$) in the observation table. In the experimental evaluation of our approach, we observed that the above expectation was true in most of the cases.

We now show that the output of **MkWellFormed** procedure is a well-formed table.

Lemma 7 *The* **MkWellFormed** *procedure returns a well-formed observation table.*

Proof. Given an observation table $\mathcal{T} = (S, E, T)$, the **MkWellFormed** procedure restricts S to a well-formed cover (say S') and E to an experiment cover (say E'). Let the table obtained as a result be $\mathcal{T}'$. It follows from Definition 9 that for all s_1,$s_2 \in S'$, $s_1 \not\equiv s_2$. Using the definition of $\equiv$ (cf. Section 3.1.3), we know that for some $e \in E$, $T(s_1 \bullet e) \neq T(s_2 \bullet e)$. Now, consider the following two cases:

Case 1. If $e \in E'$, $s_1 \not\equiv s_2$ still holds in the result table since $T(s_1 \bullet e) \neq T(s_2 \bullet e)$.

Case 2. Otherwise, $e \notin E'$. However, by Definition 11, there exist some $e' \in E'$, so that $Col(e') = Col(e)$. By using the definition of Col (Definition 10), it follows that for all $s \in S$, $T(s \bullet e) = T(s \bullet e')$. Hence, $T(s_1 \bullet e') = T(s_1 \bullet e) \neq T(s_2 \bullet e) = T(s_2 \bullet e')$. Therefore, $s_1 \not\equiv s_2$ holds and so the output table $\mathcal{T}'$ is well-formed.

□

Lemma 8 *The* **Reval** *procedure always computes a valid observation table for the unknown language U' as an output.*

Proof. Refer to Figure 7 describing the **Reval** procedure. By construction, the table obtained at the end of Step 1 must *agree* with U'. In Step 2, the procedure **MkWellFormed** is applied. Therefore, it follows from Lemma 7 that the resultant table is *well-formed*. As a result, the final table both agrees with U' and is well-formed; hence, by Definition 8, it is valid.

□

It follows from Lemma 8 and Theorem 3 that starting from an observation table computed by the **Reval** procedure, the L^* algorithm must terminate with the correct minimum DFA for an unknown language U'.

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 finite automata $M_1, \ldots, M_n$ and a specification automaton φ, the following non-circular rule **AG** [49] can be used to verify $M_1 \parallel \cdots \parallel M_n \preccurlyeq \varphi$:

$$M_1 \parallel A_1 \preccurlyeq \varphi$$
$$\frac{M_2 \parallel \cdots \parallel M_n \preccurlyeq A_1}{M_1 \parallel \cdots \parallel M_n \preccurlyeq \varphi}$$

In the above equation, A_1 is a finite automaton representing the assumption about the environment under which M_1 is expected to operate correctly. As also observed by Cobleigh et al. [23], the second premise is itself an instance of the top-level proof obligation with $n-1$ component finite automata. Hence, **AG** can be applied to decompose it further. It has been shown that the **AG** rule is both sound and complete [23]. The proof of completeness relies on the existence of an unique weakest assumption (cf. Lemma 1) for a component automaton M and property φ.

As mentioned above, the rule **AG** can be instantiated recursively for n components [23] as follows.

$$M_i \parallel A_i \preccurlyeq A_{i-1} (1 \leq i \leq n-1, A_0 = \varphi)$$
$$\frac{M_n \preccurlyeq A_{n-1}}{M_1 \parallel .. \parallel M_n \preccurlyeq \varphi}$$

Our algorithm for checking compatibility uses this instantiation of rule **AG** for n components. We can show that this rule is *complete* using the notion of weakest assumptions. Recall (cf. Definition 5) that for any finite automaton M and a specification automaton φ, there must exist a weakest finite automaton assumption WA such that $M \parallel A \preccurlyeq \varphi$ iff $A \preccurlyeq WA$ and $M \parallel WA \preccurlyeq \varphi$. For the above instantiation of **AG** rule, we can define a set of weakest assumptions WA_i ($1 \leq i \leq n-1$) as follows. It is clear that a weakest assumption WA_1 exists such that $M_1 \parallel WA_1 \preccurlyeq \varphi$. Given WA_1, it follows that WA_2 must exist so that $M_2 \parallel WA_2 \preccurlyeq WA_1$. Therefore, by induction on i, there must exist weakest assumptions WA_i for $1 \leq i \leq n-1$, such that $M_i \parallel WA_i \preccurlyeq WA_{i-1} (1 \leq i \leq n-1, WA_0 = \varphi)$ and $M_n \preccurlyeq A_{n-1}$.

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 10. 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 $\mathcal{I} \subseteq \{1, \ldots, n\}$), the compatibility-check procedure checks if a safety property φ 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 (cf. Section 5.3.2) and is done in an iterative abstraction-refinement style as follows:

1. Use predicate abstraction to obtain finite automaton 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 a counterexample. Otherwise we go to the next step.
4. Refine a specific model, say M_k, such that the spurious CE is eliminated. Repeat the process from Step 2.

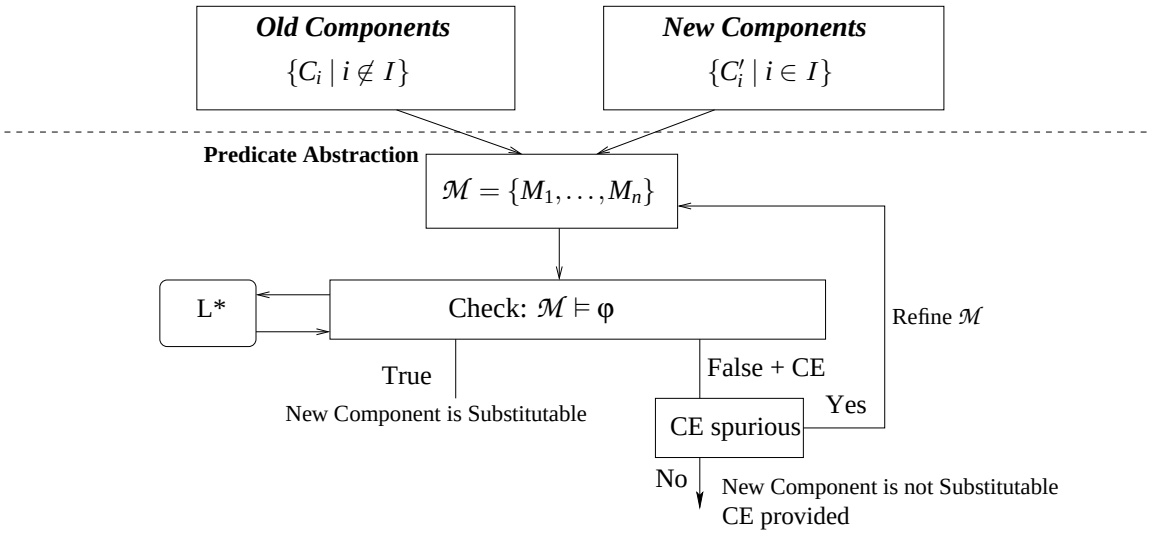


Fig. 10 The Compatibility Phase of the Substitutability Framework

5.3.1 Overview of DynamicCheck. We first present an overview of the algorithm for two finite automata and then generalize it to an arbitrary collection of finite automata. Suppose we have two old finite automata, M_1 and M_2, and a property finite automaton φ. We assume that we previously tried to verify $M_1 \parallel M_2 \preccurlyeq \varphi$ 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 $M_1 \parallel M_2 \preccurlyeq \varphi$, **DynamicCheck** had constructed an observation table $\mathcal{T}$.

Now suppose that we have new versions M_1' and M_2' for M_1 and M_2. Note that, in general, either M_1' or M_2' could be identical to its old version. **DynamicCheck** now reuses $\mathcal{T}$ and invokes the dynamic L^* algorithm to automatically learn an assumption A' such that (i) $M_1' \parallel A' \preccurlyeq \varphi$ and (ii) $M_2' \preccurlyeq A'$. More precisely, **DynamicCheck** proceeds iteratively as follows:

1. It checks if $M_1 = M_1'$. If so, it initializes learning from the previous table $\mathcal{T}$ (i.e., it sets $\mathcal{T}' := \mathcal{T}$). Otherwise, it revalidates $\mathcal{T}$ against M_1' to obtain a new table $\mathcal{T}'$.
2. It derives a conjecture A' from $\mathcal{T}'$ and checks if $M_2' \preccurlyeq A'$. 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 $M_1' \parallel M_2' \preccurlyeq \varphi$. If so, it constructs such a counterexample and terminates with FALSE. Otherwise, it adds a new experiment to $\mathcal{T}'$ using CE. This is done via the algorithm by Rivest and Schapire [50] as explained in Section 3.1.4. Therefore, once the new experiment is added, $\mathcal{T}'$ is no longer closed.
4. It makes $\mathcal{T}'$ closed by making membership queries and repeats the process from Step 2.

We now describe the key ideas that enable us to reuse the previous assumptions and then present the complete **DynamicCheck** algorithm for multiple finite automata. Due to its dynamic nature, the algorithm is able to locally identify the set of assumptions that must be modified to revalidate 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 ($S \bullet E$) in an unknown language U. 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'. As mentioned earlier, in most cases, the languages $L(A)$ and $L(A')$ may differ only slightly; hence, the information about the behaviors of A is reused in computing A'.

Table Revalidation. The original L^* algorithm computes A' starting from an empty table. However, as mentioned before, a more efficient algorithm would try to reuse the previously inferred set of elements of S and E to learn A'. The result in Section 5.1.1 (Theorem 3) precisely enables the L^* algorithm to achieve this goal. In particular, since L^* terminates starting from any *valid* table, the algorithm uses the **Reval** procedure to obtain a valid table by reusing traces in S and experiments in E. The valid table thereby obtained is subsequently made closed, and then learning proceeds in the normal fashion. Doing this allows the compatibility check to restart from any previous set of assumptions by revalidating them. The **RevalidateAssumption** module implements this feature (see Figure 12).

5.3.2 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 new versions of a component finite automaton by M and M' and the previous and new versions of component assemblies by $\mathcal{M}$ and $\mathcal{M}'$, respectively. For ease of description, we always use a property, φ, to denote the right-hand side of the top-level proof obligation of the **AG** rule. We denote the modified property[3] at each recursion level of the algorithm by φ'. The old and new assumptions are denoted by A and A', respectively.

Figure 12 presents the pseudo-code of the **DynamicCheck** algorithm to perform the compatibility check. Lines 1-4 describe the case when $\mathcal{M}$ contains only one component. In Line 5-6, if the previous assumption is found to be not valid (using **IsValidAssumption** procedure) with respect to the weakest assumption corresponding to M' and φ', it is revalidated using the **RevalidateAssumption** procedure. Lines 8-10 describe the recursive invocation of **DynamicCheck** on $\mathcal{M}' \setminus M'$ against property A'. Finally, Lines 11-16 show how the algorithm detects a counterexample CE and uses it to update A' or terminates with a TRUE result or a counterexample. The salient features of this algorithm are the following:

– We assume that there exists a set of previously computed assumptions from the earlier verification check. Suppose we have a component automaton M and a property automaton φ, such that the corresponding weakest assumption

[3] Under the recursive application of the compatibility-check procedure, the updated property φ' corresponds to an assumption from the previous recursion level.

GenerateAssumption (A, CE)

// Let (S,E,T) be the L^* observation table corresponding to an assumption A;

1: Obtain a *distinguishing suffix* e from CE;

2: $E := E \cup \{e\}$;

3: **forever do**

4: **CloseTable**();

5: A' := **MkDFA**($\mathcal{T}$);

6: **if** (**IsCandidate**(A')) $\mathbf{return} A'$;

7: **let** CE' be the counterexample returned by **IsCandidate**;

8: Obtain a *distinguishing suffix* e from CE';

9: $E := E \cup \{e\}$;

Fig. 11 Pseudo-code for procedure **GenerateAssumption**.

is WA. In order to find out if a previously computed assumption (say A) is valid against $L(WA)$ (cf. Definition 8), the **IsValidAssumption** procedure is used. More precisely, the **IsValidAssumption** procedure checks if the observation table (say $\mathcal{T}$) corresponding to A is valid with respect to $L(WA)$ by asking a membership query for each element of the table (cf. Lemma 2).

– The procedure **GenerateAssumption** (cf. Figure 11) essentially models the L^* algorithm. Given a counterexample CE, the procedure **GenerateAssumption** computes the next candidate assumption in a manner similar to the original L^* algorithm (cf. Section 3.1.4). The termination of the **GenerateAssumption** procedure directly follows from that of the L^* algorithm.

– Verification checks are repeated on a component M' (or a collection of components $\mathcal{M}' \setminus M'$) only if it is (or they are) found to be different from the previous version M ($\mathcal{M} \setminus M$) or if the corresponding property φ has changed (Lines 3, 8). Otherwise, the previously computed and cached result (returned by the procedure **CachedResult**) is reused (Lines 4, 9).

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

1: **let** M' = *first* element of $\mathcal{M}'$;

//M and φ denote the first element of $\mathcal{M}$ and the corresponding property before upgrade

//and A denotes the assumption computed previously for M and φ

2: **if** ($\mathcal{M}' = \{M'\}$)

3: **if** ($M \neq M'$ or $\varphi \neq \varphi'$) **return** ($M' \preccurlyeq \varphi'$);

4: **else return CachedResult**($M \preccurlyeq \varphi$);

//check if A is a valid assumption for M' and φ'

5: **if** ($\neg$ **IsValidAssumption**(A, M', φ'))

//make assumption A valid for M' and φ'

6: A' := **RevalidateAssumption**(A, M', φ');

7: **else** $A' := A$;

//Now check the rest of the system $\mathcal{M}' \setminus M'$ against A'

8: **if** ($A \neq A'$ or $\mathcal{M} \setminus M \neq \mathcal{M}' \setminus M'$)

9: res := **DynamicCheck**($\mathcal{M}' \setminus M'$, A');

10: **else** res := **CachedResult**($\mathcal{M} \setminus M \preccurlyeq A$);

11: **while**(res is not TRUE)

//Let CE be the counterexample obtained

12: **if** ($M' \parallel CE \preccurlyeq \varphi'$)

13: A' := **GenerateAssumption** (A',CE); *// Obtain A' so that $M' \parallel A' \preccurlyeq \varphi'$*

14: res = **DynamicCheck** ($\mathcal{M}' \setminus M'$, A'); *// Check if $\mathcal{M}' \setminus M' \preccurlyeq A'$*

15: **else return** a witness counterexample CE' to $M' \parallel CE \not\preccurlyeq \varphi'$;

16: **return** TRUE;

Fig. 12 Pseudo-Code for Compatibility Checking on an upgrade. The procedure returns TRUE if $\mathcal{M}' \preccurlyeq \varphi'$ holds, otherwise returns a counterexample witness CE.

Note that for a component automaton M and a counterexample trace CE, we write $M \parallel CE$ to denote the composition of M with the automaton representation of the trace CE (where the last state is the only accepting state). In order to prove the correctness of **DynamicCheck**, we need the following lemma.

Lemma 9 *Suppose $\mathcal{M}$ is a set of component automata (with $M \in \mathcal{M}$) and φ be a specification automaton. Let $\mathcal{M} \setminus M \not\preccurlyeq \varphi$ hold and CE be a witness to it. Moreover, suppose $M \parallel CE \not\preccurlyeq \varphi$ holds, and CE' is a witness to it. Then $\mathcal{M} \not\preccurlyeq \varphi$ holds and CE' is a witness to it.*

Proof. Let $M_2 = \mathcal{M} \setminus M$. Since CE is a witness to $M_2 \not\preccurlyeq \varphi$, we know that $CE \in L(M_2)$. Also, since $M \parallel CE \not\preccurlyeq \varphi$ holds and CE' is a witness to it, there is a $CE'' \in L(M)$ such that $CE' = (CE'' \parallel CE)$ (using the automaton representation of both CE and CE''). Also, $CE' \notin L(\varphi)$. Since $CE'' \in L(M)$ and $CE \in L(M_2)$, it follows that $CE' = (CE'' \parallel CE)$ is in $L(M \parallel M_2) = L(\mathcal{M})$. Hence, CE' is in $L(\mathcal{M})$ but not in $L(\varphi)$. Therefore, CE' is a witness to $\mathcal{M} \not\preccurlyeq \varphi$.

Theorem 4 shows the correctness of **DynamicCheck**. The proof relies on the fact that the rule **AG** for a system of n component automata is complete due to the existence of an unique set of weakest assumptions (cf. Section 5.2). Note that we never construct the weakest assumptions directly; they are only used to show that the procedure **DynamicCheck** terminates with the correct result.

Theorem 4 *Given modified $\mathcal{M}'$ and φ', the* **DynamicCheck** *algorithm always terminates with either $true$ or a counterexample CE to $\mathcal{M}' \preccurlyeq \varphi'$.*

Proof. We assume that for the earlier system $\mathcal{M}$, a set of previously computed assumption automata $A_1 \ldots A_{n-1}$ exist. Suppose one or more components in $\mathcal{M}$ are upgraded resulting in the system $\mathcal{M}'$.

The proof proceeds by induction over the number of components k in $\mathcal{M}'$. In the base case $\mathcal{M}'$ consists of a single component automaton M'; hence we need to model check M' against φ' only if either M or φ changed. This is done in Lines 3-4. Hence, **DynamicCheck** returns the correct result in this case.

Assume for the inductive case that **DynamicCheck**$(\mathcal{M}' \setminus M', A')$ terminates with either $true$ or a counterexample CE. If Line 8 holds (i.e., $A' \neq A$ or $\mathcal{M} \setminus M \neq \mathcal{M}' \setminus M'$), then, by the inductive hypothesis, execution of Line 9 terminates with the correct result: either $true$ or a counterexample CE. Otherwise, the previously computed correct

result res is used (Line 10). Based on this result, Lines 11-16 update the current assumption in an iterative manner. Therefore, it remains to be shown that Lines 11-16 compute the correct return value based on this result.

If the result in Line 9 or Line 10 is $true$, it follows from the soundness of the assume-guarantee rule that $\mathcal{M}' \preccurlyeq \varphi'$ and **DynamicCheck** returns $true$ (Line 16). Otherwise, a counterexample CE is found which is a witness to $\mathcal{M} \setminus M \not\preccurlyeq \varphi'$. This counterexample is used in Line 12 to check if $M' \parallel CE \preccurlyeq \varphi'$. If this holds, then CE is used to improve the current assumption in Lines 13-14. Otherwise, the procedure returns a suitable witness CE' (Line 15). In order to show that Lines 11-16 compute the correct result, we need to show that (i) the counterexample CE' is indeed a witness to $\mathcal{M}' \not\preccurlyeq \varphi'$ and, (ii) the loop in Lines 11-15 can execute only a finite number of times.

Using the fact that CE is a witness to $\mathcal{M}' \setminus M' \not\preccurlyeq \varphi'$ (from Lines 9-10) and $M' \parallel CE \not\preccurlyeq \varphi'$ (Line 12), it follows from Lemma 9 that $\mathcal{M}' \not\preccurlyeq \varphi'$ and CE' is a suitable witness to $\mathcal{M}' \not\preccurlyeq \varphi'$.

It remains to show that Lines 11-15 can execute only a finite number of times. Note that in Line 13, A' is valid since it was computed by **RevalidateAssumption** (Line 6). Hence, **GenerateAssumption** (Line 13) must terminate (cf. Theorem 3) by learning a new assumption, say A'', such that $M' \parallel A'' \preccurlyeq \varphi'$. Note that by Lemma 4, the number of states of A' or A'' cannot exceed that of the corresponding weakest assumption WA'. Also, it follows from the proof of correctness of L^* (cf. Theorem 1) that $|A'| < |A''|$. Moreover, by the inductive hypothesis, Line 14 must terminate with the correct result. Hence, each iteration of Lines 11-14 of the **while** loop will lead to increase in the number of states of the assumption candidates until $|A''| = |WA'|$. In this case, the loop terminates. If no counterexample is generated at Line 14, then the loop terminates with a true result at Line 16. Otherwise, if a counterexample CE is generated at Line 14 (with $A'' = WA'$), then it follows that $CE \in L(\mathcal{M}' \setminus M')$ and $CE \notin L(WA')$. Therefore it follows from Lemma 2 that $M' \parallel CE \preccurlyeq \varphi'$ does not hold. Hence, by Lemma 9, CE is an actual witness to $\mathcal{M}' \not\preccurlyeq \varphi'$. Therefore, the procedure returns by generating the correct witness CE' at Line 15.

□

6 Implementation and Experimental Evaluation

The procedures for checking, in a dynamic manner, the substitutability of components, were implemented in the ComFoRT reasoning framework [15]. The tool includes a front end for parsing and constructing control-flow graphs

from C programs. Further, it is capable of model checking properties on programs based on automated may-abstraction (existential abstraction), and it allows compositional verification by employing learning-based, automated assume-guarantee reasoning. Specifically, we implemented the compatibility check in full while for the containment check, we only implemented the Abstraction and Verification steps (cf. Section 4) since they were sufficient for the examples we considered.

We reused the above features of COMFORT in the implementation of the substitutability check. The tool interface was modified so a collection of components and corresponding upgrades could be specified. We extended the learning-based, automated assume-guarantee to obtain its dynamic version, as required in the compatibility check. Multiple learner instances are kept across calls to the verification engine and implementing algorithms to validate multiple, previous observation tables in an efficient way during learning. For the Abstraction step in containment checking, we implemented procedures for computing must-abstractions from C code using a given set of predicates [36, 37].

We performed the compatibility check while verifying upgrades of a benchmark provided to us by our industrial partner, ABB Inc. [2]. The benchmarks consist of seven components which together implement an inter-process communication (IPC) protocol. The combined state space is over 10^6.

We used a set of properties describing the 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*. Figure 13 shows the comparison between the time required for initial verification of the IPC system, and the time taken by **DynamicCheck** for verifying the upgrades. In Figure 13, $\#Mem.\ Queries$ denotes the total number of membership queries made during verification of the original assembly, T_{orig} denotes the time required for the verification of the original assembly, and T_{ug} denotes the time required for the verification of the upgraded assembly.

Upgrade # (Prop.)	# Mem. Queries	$\mathbf{T}_{orig}$ (msec)	$\mathbf{T}_{ug}$ (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

Fig. 13 Summary of Results for **DynamicCheck**

We observed that the previously generated assumptions (after revalidation) in all the cases were also sufficient to prove the properties on the upgraded system. 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.

7 Related Work

Related projects on checking software systems across modifications often impose the restriction that every behavior of a 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. [27,17] 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 component implementations are not presented. In contrast, our approach automatically extracts conservative finite state automaton models from component implementations. Moreover, we do not require refinement among the old components and their new versions.

McCamant and Ernst [45] 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 it is gaining rapid popularity in formal verification [34]. The use of learning for automated assume-guarantee reasoning was proposed originally by Cobleigh et al. [23]. The initial methodology was followed by a symbolic approach [5], application to checking component substitutability [13], extensions to different notions of conformance [14, 16], combination with automated system decomposition using hyper-graph partitioning [48], optimized learning and iterative alphabet enlargement approaches [16, 30], lazy learning approach [53] and a technique for computing minimal assumptions [35]. The problem of choosing a suitable order of components for assume-guarantee reasoning has been addressed in Gheorghiu et al. [30]. Cobleigh et al. investigate the advantages of automated AGR methods over monolithic verification techniques in the context of LTSA and FLAVERS tools [24] by experimenting with different two-way system decompositions. The use of learning along with predicate abstraction has also been applied in the context of interface synthesis [3] and various types of assume-guarantee proof rules for automated software verification [10].

This paper is related to our earlier project [11] 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 by Chaki and et al. [11]. 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 report defines a new technique based on the simultaneous use of over-approximations and under-approximations obtained via existential and universal abstractions.

Another approach to preserve behavioral properties of a component across an upgrade is based on the principle of behavioral sub-typing [43]: type T' is a subtype of type T if for every property $\phi(t)$ provable about objects t of type T, $\phi(t')$ is provable about objects t' of type T'. The notion of subtypes is extended to system behaviors by augmenting object types with invariants and constraints and showing that these constraints are maintained for objects of the subtype. However, this approach focuses only on the given behavior specification of a single component and does not take into account the way it is used in the component assembly. In contrast, the assumptions in our approach reflect the behavior of environment components. Therefore, although the upgraded component may not satisfy a property ϕ in all possible environments, it may continue to satisfy ϕ in context of the current environment components. In other words, the new component may not be a behavioral subtype of the earlier one, but still be compatible with its environment.

8 Conclusions

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.

References

1. M. Abadi and L. Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–534, May 1995.
2. *ABB*. http://www.abb.com, 2005.

3. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 98–109, Long Beach, CA, January 12–14, 2005. New York, NY, 2005. Association for Computing Machinery (ACM).

4. R. Alur and T. Henzinger. Reactive Modules. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 207–218, New Brunswick, NJ, July 27–30, 1996. Los Alamitos, CA, 1996. IEEE Computer Society Press.

5. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. of 17th Int. Conf. on Computer Aided Verification*, 2005.

6. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.

7. Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*, 75(2):87–106, November 1987.

8. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, Snowbird, UT, June 20-22, 2001. New York, NY, 2001. Association for Computing Machinery.

9. T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, Redmond, WA, February 2000. ftp://ftp.research.microsoft.com/pub/tr/tr-2000-14.pdf.

10. S. Chaki, E. Clarke, D. Giannakopoulou, and C. S. Păsăreanu. Abstraction and assume-guarantee reasoning for automated software verification. Technical Report 05.02, Research Institute for Advanced Computer Science (RIACS), Mountain View, CA, 2004.

11. S. Chaki, N. Sharygina, and N. Sinha. Verification of Evolving Software. In *Proceedings of the Third Workshop on Specification and Verification of Component Based Systems (SAVCBS)*, pages 55–61, Newport Beach, CA, October 31–November 1, 2004. Ames, Iowa: Iowa State University, 2004.

12. Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2–3), 2004.

13. Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. Dynamic component substitutability analysis. In *Proc. of Conf. on Formal Methods*, 2005.

14. Sagar Chaki, Edmund Clarke, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proc. of 17th Int. Conf. on Computer Aided Verification*, 2005.
15. Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau. The ComFoRT Reasoning Framework. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 164–169, Edinburgh, Scotland, July 6–10, 2005. New York, NY, 2005. Springer-Verlag.
16. Sagar Chaki and Ofer Strichman. Optimized L* for assume-guarantee reasoning. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, 2007.
17. 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 (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 428–441, Copenhagen, Denmark, July 27–31, 2002. New York, NY, 2002. Springer-Verlag.
18. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Chicago, IL, July 15–19, 2000. Berlin, Germany, 2000. Springer-Verlag.
19. E. Clarke, O. Grumberg, and D. Long. Model Checking and Abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*, pages 343–354, Albuquerque, NM, January 19–22, 1992. New York, NY, 1992. Association for Computing Machinery (ACM).
20. Edmund Clarke and Allen Emerson. Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, New York, May 4–6, 1982. Berlin, Germany, 1982. Springer-Verlag.
21. Edmund Clarke, David Long, and Kenneth McMillan. Compositional Model Checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS '89)*, pages 353–362, Pacific Grove, CA, June 5–8, 1989. Washington, DC, 1989. IEEE Computer Society Press.
22. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000.
23. J. M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning Assumptions for Compositional Verification. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346, Warsaw, Poland, April 7–11, 2003. New York, NY, 2003. Springer-Verlag.

24. Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *Proc. of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–108, 2006.

25. M. Colón and T. E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 293–304, Vancouver, Canada, June 28–July 2, 1998. Berlin, Germany, 1998. Springer-Verlag.

26. S. Das and D.L. Dill. Successive Approximation of Abstract Transition Relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 51–60, Boston, MA, June 16–19, 2001. Los Alamitos, CA, 2001. IEEE Computer Society Press.

27. L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proceedings of the Ninth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '01)*, pages 109–120, Vienna, Austria, September 10–14, 2001. New York, NY, 2001. ACM Press.

28. Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors. *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, volume 1536 of *Lecture Notes in Computer Science*. Springer, 1998.

29. Edsger Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

30. Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Refining interface alphabets for compositional verification. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, 2007.

31. D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption Generation for Software Component Verification. In *Proceedings of the 17th International Conference on Automated Software Engineering (ASE '02)*, pages 3–12, Edinburgh, Scotland, September 23–27, 2002. Los Alamitos, CA, 2002. IEEE Computer Society Press.

32. Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the Ninth International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 22–25, 1997. New York, NY, 1997. Springer-Verlag.

33. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, June 1997.

34. A. Groce, D. Peled, and M. Yannakakis. Adaptive Model Checking. In *Proceedings of the Eighth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 357–370, Grenoble, France, April 8–12, 2002. New York, NY, 2002. Springer-Verlag.
35. Anubhav Gupta, Ken McMillan, and Zhaohui Fu. Automated assumption generation for compositional verification. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, 2007.
36. Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 212–226, 2006.
37. Arie Gurfinkel, Ou Wei, and Marsha Chechik. Yasm: A software model-checker for verification and refutation. In *Proc. of Computer-Aided Verification*, pages 170–174, 2006.
38. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '02)*, volume 37(1) of *SIGPLAN Notices*, pages 58–70. ACM Press, January 2002.
39. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
40. James Ivers and Natasha Sharygina. Overview of ComFoRT: A model checking reasoning framework. *CMU/SEI-2004-TN-018*, 2004.
41. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
42. R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, 1995.
43. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
44. MAGIC. `http://www.cs.cmu.edu/~chaki/magic`.
45. S. McCamant and M. D. Ernst. Early Identification of Incompatibilities in Multi-Component Upgrades. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 440–464, Oslo, Norway, June 14–18, 2004. New York, NY, 2004. Springer-Verlag.
46. K. McMillan. A Compositional Rule for Hardware Design Refinement. In *Proceedings of the Ninth International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 24–35, Haifa, Israel, June 22–27, 1997. New York, NY, 1997. Springer-Verlag.

47. Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.

48. Wonhong Nam and Rajeev Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In *4th International Symposium on Automated Technology for Verification and Analysis (ATVA '06)*, pages 170–185, 2006.

49. 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, 1985. Springer-Verlag.

50. R. L. Rivest and R. E. Schapire. Inference of Finite Automata Using Homing Sequences. *Information and Computation*, 103(2):299–347, 1993.

51. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, New York, 1998.

52. Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for CTL. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, pages 546–560, 2004.

53. Nishant Sinha and Edmund Clarke. SAT-based compositional verification using lazy learning. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, 2007.