Verification of Evolving Software via Component Substitutability Analysis

Sagar Chaki
Carnegie Mellon University,
Software Engineering Institute (SEI)
Edmund Clarke
Carnegie Mellon University,
School of Computer Science (SCS)
Natasha Sharygina
Carnegie Mellon University, SEI and SCS
Nishant Sinha
Carnegie Mellon University, Department of Electrical and Computer Engineering

December 2005

Independent Research and Development Project 2005

Technical Report
CMU/SEI-2005-TR-008
ESC-TR-2005-008

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

Abstract

Formal verification by model checking has the potential to produce major enhancements in the reliability and robustness of software. However, a shortcoming in most model checking research is the failure to consider how to make the use of model checking routine throughout various stages of software development. This report presents results of the Independent Research and Development (IRAD) project on verification of evolving software conducted at the Software Engineering Institute in 2005. The research conducted as part of the IRAD project considered ways to reduce the effort of subsequent verifications. In particular, it resulted in the development of techniques that exploit the results of previous verification efforts and focus only on the portions of the system that have changed (components). Thus, these new techniques incorporate model checking into development processes in a much less intrusive or cumbersome manner than previous verification techniques.

The report presents an automated and compositional procedure to solve the component substitutability problem. The solution contributes two techniques for checking the correctness of software upgrades: (1) a technique based on simultaneous use of overapproximations and underapproximations obtained via existential and universal abstractions and (2) a dynamic assume-guarantee reasoning algorithm in which 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, the solution generates constructive feedback that shows developers how to improve the components. The substitutability approach has been implemented and validated in the Component Formal Reasoning Technology (COMFORT) model checking tool set. The experimental evaluation of an industrial benchmark demonstrates encouraging results.

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. Examples of serious software errors are easy to find. For instance, in 1997, the propulsion system of the Aegis missile cruiser USS Yorktown failed for over two hours due to a software bug [Slabodkin 98]. The cause turned out to be a division by zero within a database system, which resulted in an exception and a crash of all computer consoles and terminal units. The software of the USS Yorktown operated on a network of Windows NT machines and was quite complex, consisting of several million lines of C code.

Another instance is the development of the F/A-22 as part of the Joint Strike Fighter program. The project was delayed multiple times, and often the project's delay was caused by the inability of software developers to produce bug-free software for the F/A-22. Pilots often had to reboot computers while in the air [U.S. Govt. 05, Nellemann 94]. The F/A-22 has about 2.5 million lines of software written in Ada. This number is expected to rise to 6 million lines of C/C++ code on the F-35.

Computer software also plays an important role in other parts of our infrastructure. On August 14, 2003, a blackout affected more than 50 million people in large areas on the U.S. east coast, causing an estimated damage between $4 billion and $10 billion [U.S.-Canada 04]. While the blackout was triggered by trees hitting local power transmission lines, a software bug made the damage devastating. A bug in General Electric (GE) Energy's XA/21 power control system allowed the blackout to spread. The software had been in use since 1990, but the bug had not become apparent previously. The flaw was discovered by an audit of over 4 million lines of C/C++ code after the blackout and was identified as a "race condition."

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. 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 [Clarke 82, Clarke 00b]. Introduced in 1981, it has become 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		Thread B	
1	`while(x!=0) skip;`	**1**	`while(x!=0) skip;`
2	`x=1;`	**2**	`x=2;`
3		**3**	

Figure 1: A Small Program with Two Threads of Control

Model checking also has the potential to produce major enhancements in the reliability and robustness of software. The basic idea of software model checking 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 as in the case of the USS Yorktown, a race condition as in the case of GE's XA/21, or a violated assertion. Once such an erroneous state is found, it can be reported to the programmer together with a counterexample (i.e., an error trace), which demonstrates the flaw. Counterexamples can be very helpful for understanding the nature of the error and fixing it.

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. Abstractions are most often performed in an informal, manual manner and require considerable expertise.

Manual abstraction is error prone too. The person performing the abstraction will often capture the intended behavior when abstracting and not the behavior of the actual code. Thus, a bug could be hidden in the code. Industrial applications of model checking therefore favor automated ways to compute the abstract model. One such method, called *predicate abstraction* [Graf 97, Colón 98], has proven to be particularly successful when applied to large software programs. We exploited predicate abstraction while developing a solution to the problem of establishing the correctness of evolving systems. We describe predicate abstraction and its application to verification of evolving software in Section 3.2.

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 evolved systems [Pnueli 85]. We describe the assume-guarantee reasoning paradigm and its application to verification of evolving software in Section 3.3.

In this document, we describe a particular model checking problem, namely verification of evolving software. The rest of the document is organized as follows: Section 2 provides some background information on the model checking technology, the types of claims it can analyze, and the current state of research and practice of model checking. Section 3 describes the problem of verification of evolving systems and presents a detailed description of the techniques that we have developed to

overcome difficulties in the verification of evolving programs. Section 4 provides an overview of related work, and Section 5 summarizes the contributions of the Independent Research and Development (IRAD) project.

2 Model Checking

In formal verification, a system is modeled mathematically, and its specification (also called a *claim* in model checking) is described in a formal language. When the behavior in a system model does not violate the behavior specified in a claim, the model satisfies the specification. Model checking [Clarke 82] is a fully automated form of formal verification that uses algorithms that check whether a system satisfies a desired claim through an exhaustive search of all possible executions of the system. The exhaustive nature of model checking renders the typical testing question of adequate coverage unnecessary.

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 algorithms that have reasonable efficiency and that can be run on moderate-sized machines.

Although the restriction to finite-state systems may seem to be a major disadvantage, model checking is applicable to several important classes of systems [Clarke 00b]. Hardware controllers are finite-state systems, as are many communication protocols. Software, which is not finite state, can still be verified if variables are assumed to be defined over finite domains. This assumption does not restrict the applicability of model checking because many interesting behaviors of the software systems can be specified with finite-state models. For example, systems with unbounded message queues can be verified by restricting the size of the queues to a small number such as two or three.

In classical model checking, systems are modeled mathematically as state transition systems, and claims are specified using temporal logic [Pnueli 77, Clarke 86]. Temporal logic is used to define formulas that describe system behavior over time, where the propositions of the logic are behaviors of interest involving state information (current state or values of variables) or events. Temporal logic formulas combine such propositions with temporal operators to describe interesting patterns of propositions over time, such as the following:

- Whenever X is greater than Y, Z must also be greater than Y.
- Some invariant (e.g., mutual exclusion with respect to some resource) always holds once initialization is complete.
- A component can issue requests only during an allowed interval (as bounded by events granting and taking away permission).

Temporal logic model checking is extremely useful in verifying the behavior of systems composed of concurrent processes or interacting nondeterministic sequential tasks. Concurrency errors (as well as errors caused by the nondeterministic execution of actions) are among the most difficult to find by testing because they tend to be irreproducible.

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 checks all system executions captured by the model and produces the answer "yes" as output if the claim holds in the model (M) and the answer "no" otherwise. When a claim is not satisfied, most model checkers produce a *counterexample* of 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 quickly why a claim is not satisfied.

2.2 Current Research in Software Model Checking

Model checking is efficient in hardware verification, but applying it to software 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 logical formalisms 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:

- **Compositional reasoning:** Verification is partitioned 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.
- **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.

2.2.1 Compositional Reasoning

Because model checking was created to verify hardware systems and because most hardware designs have a natural division into modules, the extension of model checking to larger designs is often achieved by taking a "divide and conquer" approach. 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 $M1$ and $M2$ and a claim T, the composed system satisfies T (written $M1 \parallel M2 \models T$). A naive compositional approach proceeds by executing the following steps: (1) $M1 \models T$ and (2) $M2 \models T$ and concludes by proofs that $M1 \parallel M2 \models T$. Although this rule is sound in theory, it is often not useful in practice. Usually, both $M1$ and $M2$ behave like T only in a suitable environment. To solve this problem, the compositional principle can be strengthened to an *assume-guarantee* principle [Abadi 95, Alur 96, Clarke 89, Kurshan 95, McMillan 97]: in order to check $M \models T$, it suffices to check both $M1 \parallel T2 \models T1$ and $M2 \parallel T1 \models T2$. This obligation uses the local specifications $T1$ and $T2$ as the constraining environment (also called *assumptions*) with regard to the behavior of $M2$ and $M1$ taken in isolation from $M1$ and $M2$, respectively. In general, for a system composed of multiple modules, assume-guarantee reasoning succeeds only if it can be shown that each system component M_i satisfies a corresponding specification component T_i under a suitable constraining environment.

2.2.2 Abstraction

Abstraction is one of the principal techniques for reducing the complexity of a verification problem [Ball 01, Clarke 92, Kurshan 95]. Abstraction techniques reduce the state space by mapping the concrete set of actual system states to an abstract set of states that preserve the actual system's behavior. Abstractions are usually performed in an informal, manual manner and require considerable expertise. Predicate abstraction [Graf 97, Colón 98] 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, 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. Under this approach, software systems are rendered finite by restricting variables to finite domains. As mentioned earlier, bounded model checking does not seriously restrict the applicability of model checking, since many interesting behaviors of software systems can be specified using bounded finite-state models.

The abstract program is created using existential abstraction [Clarke 92]. This method defines the transition relation of the abstract program so it is guaranteed to be a conservative overapproximation of the original program, with respect to the set of given predicates. 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. When a spurious counterexample is encountered, refinement is performed by adjusting the set of predicates in a way that eliminates it.

2.2.3 Counterexample-Guided Abstraction Refinement (CEGAR)

Although conservative abstraction procedures (which ensure that if a claim holds for the abstract system, it also holds for the original system) are typically used, any form of abstraction may introduce behaviors not found in the concrete system. Counterexamples from model checking the abstract system are often used to detect unrealistic behaviors and refine the system. Repeatedly refining the abstractions, however, may introduce additional behaviors that result in state space explosion during the model checking phase. These drawbacks (coupled with the potential effectiveness of abstraction methods) motivated research into targeted abstractions (i.e., control abstraction, loop abstraction, and so forth), which can result in more accurate abstract systems.

The abstraction refinement process has been automated by the CEGAR paradigm [Kurshan 95, Ball 00, Clarke 00a, Das 01]. The CEGAR framework is shown in Figure 2: one starts with a coarse abstraction (for example, an abstraction of a C program). If an error trace reported by the model checker is not realistic, the error trace is used to refine the abstract program, and the process proceeds until no spurious error traces can be found. The actual steps of the loop follow the abstract-verify-refine paradigm and depend on the abstraction and refinement techniques used.

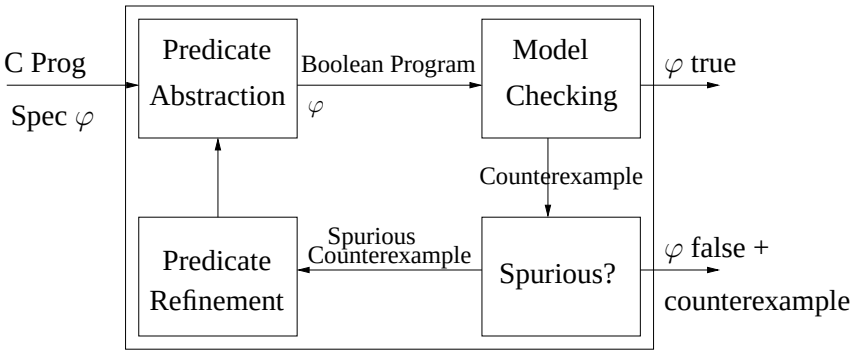

Figure 2: The CEGAR Framework

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

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 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 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. In both areas of research, low computational cost is a key factor because it enables the application of model checking to the verification of realistic programs.

This report presents techniques that use efficient abstraction and abstraction-refinement techniques of the CEGAR loop by employing techniques implemented in the COPPER model checker [Chaki 05c]. In this report, we present a solution to the model checking problem that arises during verification of evolving systems, and we refer the reader to the article by Chaki and colleagues [Chaki 04c] for details regarding the COPPER abstraction and refinement procedures. The next section describes the problem of verifying evolving software and presents our solution to address it. This solution was originally published by Chaki and colleagues [Chaki 05a].

3 Verification of Evolving Software

Successfully transitioning model checking technology has proven to be a challenging task. While the benefits of successful model checking are clear, there are several barriers to successful transition. Principally, model checking has serious scalability problems, and the techniques are difficult for software engineers to use.

A major shortcoming in most model checking research is the failure to consider how to make the use of model checking routine throughout various stages of software development. Software inevitably evolves as designs take shape, requirements change, and bugs are discovered and fixed. Model checking is useful at each such point, but the current state of model checking requires that software verification of the entire system be performed anew each time. The time and effort required to verify an entire system can be considerable, and repeating the exercise after each change, no matter how small, would likely discourage use.

In this report, we present ways to reduce the effort of subsequent verifications. In particular, by exploiting the results of previous verification efforts and focusing only on the portions of the system that have changed (components), model checking can be incorporated into development processes in a much less intrusive or cumbersome manner.

We present techniques that, while not affecting the initial model checking effort, reduce by orders of magnitude the effort to keep analysis results up to date with evolving system design. The techniques are decision procedures that determine if all system-correctness properties previously established by model checking remain valid for the new version of the system.

The key idea is to determine automatically if these properties hold for the new system without repeating each of the individual verification checks. We present a verification method [Chaki 05a] that focuses on system components that have changed during the evolution of software and determines if all behaviors of the original system are preserved in the new version of the system. Moreover, whenever behaviors are not preserved, our technique automatically provides feedback to developers showing how to improve the components whenever possible.

3.1 Background and Notation

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

Definition 1 (Words and Traces) *Given an alphabet Σ and a set of atomic propositions AP, we often say that (Σ, AP) is a state/event (SE) alphabet. For an SE alphabet $\widehat{\Sigma} = (\Sigma, AP)$, the set of words over $\widehat{\Sigma}$ is denoted by $Word(\widehat{\Sigma})$ and defined as $Word(\widehat{\Sigma}) = (\Sigma \bullet 2^{AP})^*$. The set of traces over $\widehat{\Sigma}$ is denoted by $Trace(\widehat{\Sigma})$ and defined as $Trace(\widehat{\Sigma}) = 2^{AP} \bullet Word(\widehat{\Sigma})$.*

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

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

For any DLA with transition relation δ, we write $q \xrightarrow{\alpha} q'$ to mean $q' \in \delta(q, \alpha)$. A DLA is said to be deterministic if for any $q \in S$, $\alpha \in \Sigma$, and $p \subseteq AP$, there is at most one $q' \in S$ such that $q \xrightarrow{\alpha} q'$ and $\mathcal{L}(q') = p$. DLAs are not more expressive than standard finite automata, since propositional labelings can always be rewritten in terms of actions [Clarke 00b]. However, we choose to use the DLA formalism for the sake of simplicity because it captures the essence of the SE-based notation.

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

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

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

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

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

In other words, DLAs must synchronize on shared actions and proceed independently on local actions. This notion of parallel composition is derived from the Communicating Sequential Process (CSP) formalism [Roscoe 98].

Definition 6 (Weakest Assumption) *For any DLA* M *and any safety property expressed as a DLA* φ*, there exists a weakest (w.r.t. the* $\sqsubseteq$ *preorder) DLA, which we denote as* WA*, with the following property: for any DLA* E*,* $M \parallel E \sqsubseteq \varphi$ *iff* $E \sqsubseteq WA$ *[Giannakopoulou 02]. In fact, it can be shown that* WA *is a DLA accepting the language* $\mathbb{L}(\overline{M \parallel \overline{\varphi}})$*.*

3.2 Containment

Recall that in the containment step, we verify for each $i \in \mathcal{I}$, that $C_i \sqsubseteq C_i'$ (i.e., every behavior of C_i is also a behavior of C_i'). If $C_i \not\sqsubseteq C_i'$, we construct a set $\mathcal{F}_i$ of behaviors in $Behv(C_i) \setminus Behv(C_i')$, which will be used subsequently for feedback generation. This containment check is performed iteratively and component-wise as depicted in Figure 3.

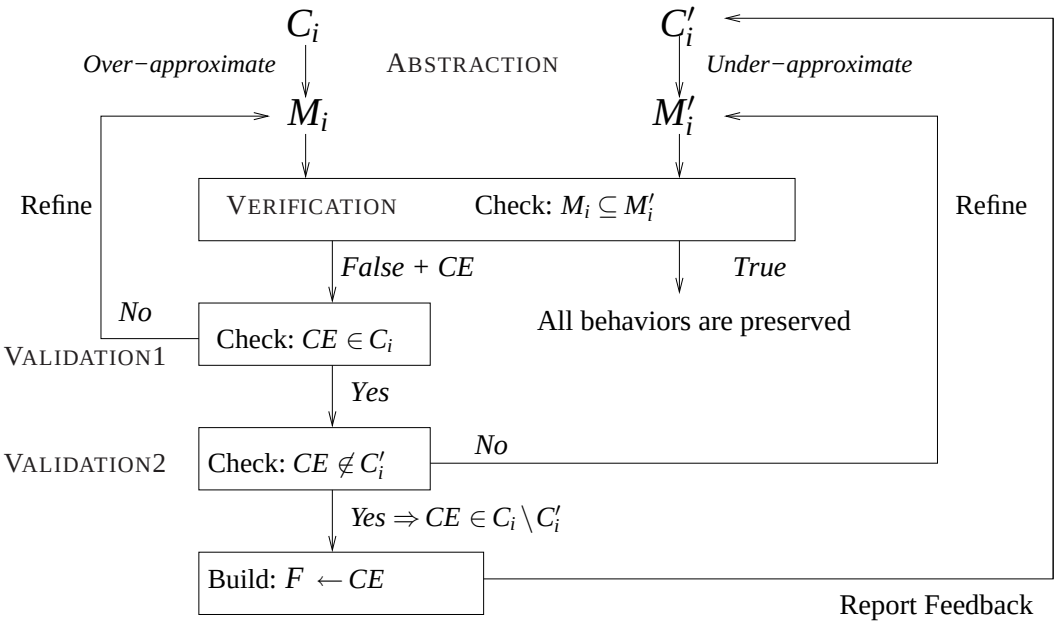

Figure 3: The Containment Phase of the Substitutability Framework

For each $i \in \mathcal{I}$, the containment check proceeds as follows:

1. **Abstraction:** Construct finite models M and M' such that **(C1)** $C_i \sqsubseteq M$ and **(C2)** $M' \sqsubseteq C_i'$. Note that M is an overapproximation of C_i and can be constructed by standard predicate abstraction. However, M' is constructed from C_i' via a modified predicate abstraction that produces an *underapproximation* of its input C program.

   Standard predicate abstraction constructs an overapproximation of the concrete system via existential abstraction. In doing so, it checks the validity of formulas using a theorem prover. Intuitively these formulas express conditions under which a transition is possible between a pair of abstract states. Our modified predicate abstraction constructs a universal approximation by modifying these formulas appropriately, so they represent conditions under which a transition is inevitable between a pair of abstract states.

2. **Verification:** Verify that $M \sqsubseteq M'$. If so, then from **(C1)** and **(C2)** above, we know that $C_i \sqsubseteq C_i'$, and we terminate with success. Otherwise we obtain a counterexample CE.

3. **Validation 1:** Check if CE is a real behavior of C_i. If so, we proceed to the next

step. Otherwise we refine model M and repeat the process from Step 2. This validation and refinement step is done according to the CEGAR procedure implemented in the MAGIC tool [Chaki 04c].

4. **Validation 2:** Check if CE is *not* a real behavior of $C_i^{'}$. If it is not, we know that $CE \in Behv(C_i) \setminus Behv(C_i^{'})$. We add CE to $\mathcal{F}_i$ and stop. Otherwise we refine M' and repeat the process from Step 2. This second validation and refinement step is an antithesis of standard abstraction refinement because it adds the valid behavior CE back to M'. However, it is conceptually similar to standard abstraction-refinement, and we omit its details in this report.

The above process terminates as soon as it adds a single behavior to $\mathcal{F}_i$. However, it can be modified easily to generate a set of behaviors in $\mathcal{F}_i$ as follows. Construct a set of counterexamples $\widehat{CE}$ in Step 2. Then process each element of $\widehat{CE}$ via Steps 3 and 4 and add to $\mathcal{F}_i$ every counterexample that belongs to C_i but not to $C_i^{'}$. The next section describes the use of $\mathcal{F}_i$ to provide feedback to developers, showing how to correct the updated components.

3.3 Compatibility

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

3.3.1 Dynamic Regular-Set Learning

Central to our compatibility check procedure is a new *dynamic* algorithm to learn regular languages. Our algorithm is based on the L^* algorithm developed by Angluin [Angluin 87]. The compatibility check uses a state/event version of the L^* that is a straightforward extension of the original algorithm (for simplicity we will refer to both as L^*). The detailed description of the state/event L^* algorithm and the proof of its correctness and complexity analysis can be found in a white paper by Chaki [Chaki 05b]. We will first present the state/event learning algorithm and then describe a dynamic version of it that we actually use for checking compatibility. We will denote the symmetric difference of two sets X and Y by $X \oplus Y$ (i.e., $\rho \in X \oplus Y$ iff $\rho \in X \setminus Y$ or $\rho \in Y \setminus X$).

3.3.1.1 The L^* Algorithm

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

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

Given an unknown regular language $U \subseteq Trace(\widehat{\Sigma})$ and a MAT for U, the L^* algorithm *iteratively* constructs a minimal DDLA D such that $L(D) = U$. It maintains an observation table (S, E, T)

where (i) S is a prefix-closed set over $Trace(\widehat{\Sigma})$ labeling the rows of the table, (ii) E is a suffix-closed set over $Word(\widehat{\Sigma})$ labeling the columns of the table, and (iii) $T\colon (S \cup S \bullet \widehat{\Sigma}) \times E \rightarrow \{0,1\}$ is the valuation of the table entries such that

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

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

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

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

Let us denote the empty word by λ. Then L^* starts with a table (S, E, T) such that $S = 2^{AP}$, $E = \{\lambda\}$, and each iteration proceeds as follows. It first updates the table using membership queries until it is closed. Next L^* builds a candidate DDLA D from the table and makes a candidate query with D. If the MAT returns TRUE to the candidate query, L^* returns D and stops. Otherwise, L^* updates E with a single word (constructed from the CE returned by the candidate query) and proceeds with the next iteration. The complexity of L^* is expressed by the following theorem [Angluin 87, Chaki 05b]:

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

3.3.1.2 Dynamic L^*

Normally L^* initializes with $S = 2^{AP}$ and $E = \{\lambda\}$. This can be a drawback in cases where a previously learned candidate (and hence a table) exists and we wish to restart learning using information from the previous table. In the following discussion, we show that if L^* begins with any non-empty valid table, it must terminate with the correct result (Theorem 2). 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.[1]

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

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

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

[1] A similar idea was also proposed in the context of adaptive model checking [Groce 02].

Proof. Let n be the number of states in the minimal DDLA M_U such that $\mathbb{L}(M_U) = U$. Note that both Theorem 1 and Lemma 5 from Angluin's correctness proof for L^* [Angluin 87] hold true for valid and closed tables and candidates consistent with them. It follows from Theorem 1 and Lemma 5 that L^* can always make a valid table closed and hence is able to construct a candidate, say D, with at most n states. We now show that every subsequent candidate must have at least one more state than D.

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

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

Suppose we have a table $\mathcal{T}$ 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 in general $\mathcal{T}$ will not be valid for U'; hence, starting from $\mathcal{T}$ will not be appropriate. However, we can first validate $\mathcal{T}$ against U' and then start L^* from the validated $\mathcal{T}$. Theorem 2 provides the key insight behind the correctness of this procedure. As we shall see, this idea forms the backbone of our dynamic compatibility-check procedure (see Section 3.3.3).

3.3.2 Assume-Guarantee Reasoning

Along with dynamic L^*, we also use assume-guarantee style compositional reasoning to check compatibility. Given a set of component DLAs $M_1, \ldots, M_n$ and a specification DLA φ, the following non-circular rule **AG** [Pnueli 85] can be used to verify $M_1 \parallel \cdots \parallel M_n \sqsubseteq \varphi$:

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

In the above equation, A_1 is a DLA representing the assumption about the environment under which M_1 is expected to operate correctly. As also observed by Cobleigh and colleagues [Cobleigh 03], the second premise is itself an instance of the top-level proof obligation with $n - 1$ component DLAs. Hence, **AG** can be applied to decompose it further.

3.3.3 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 4. 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 and is done in an iterative abstraction-refinement style as follows:

1. Use predicate abstraction to obtain finite DLA models M_i, where M_i is constructed from C_i if $i \notin \mathcal{I}$ and from C'_i if $i \in \mathcal{I}$. The abstraction is carried out component-wise. Let $\mathcal{M} = \{M_1, \ldots, M_n\}$.

2. Apply **DynamicCheck** on $\mathcal{M}$. If the result is TRUE, the compatibility check terminates successfully. Otherwise, we obtain a counterexample CE.

3. Check if CE is a valid counterexample. Once again this is done component-wise. If CE is valid, the compatibility check terminates unsuccessfully with CE as 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.

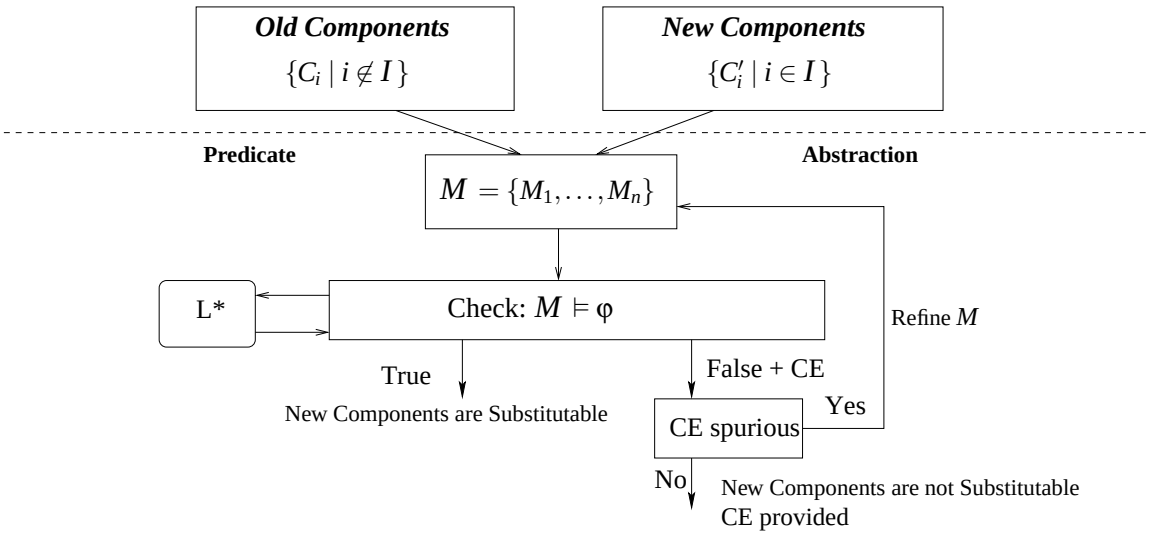

Figure 4: The Compatibility Phase of the Substitutability Framework

3.3.3.1 Overview of DynamicCheck

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

Now suppose 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** will now reuse $\mathcal{T}$ and invoke the dynamic L^* algorithm to automatically learn an assumption A' such that (i) $M'_1 \parallel A' \sqsubseteq \varphi$ and (ii) $M'_2 \sqsubseteq A'$. More precisely, **DynamicCheck** proceeds iteratively as follows:

1. It checks if $M_1 = M'_1$. If so, it starts 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 $M_2' \sqsubseteq A'$. If this check passes, it terminates with TRUE and the new assumption A'. Otherwise, it obtains a counterexample CE.

3. It analyzes CE to see if CE corresponds to a real counterexample to $M_1' \parallel M_2' \sqsubseteq \varphi$. If so, it constructs such a counterexample and terminates with FALSE. Otherwise, it updates $\mathcal{T}'$ using CE.

4. It makes $\mathcal{T}'$ closed by making membership queries and repeats the process from Step 2.

3.3.3.2 Generalized DynamicCheck

We first describe the key ideas that enable us to reuse the previous assumptions and then present the complete **DynamicCheck** algorithm for multiple DLAs. Due to its dynamic nature, the algorithm will be 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 in an unknown language U (i.e., the language of the weakest assumption for M). Upgrading the component M modifies this unknown language for the corresponding assumption from U to, say, U'. Therefore, checking compatibility after an upgrade requires that the learner must compute a new assumption A' corresponding to U'. In most cases, the languages $L(A)$ and $L(A')$ may differ only slightly; 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 intend to reuse the previously inferred set of elements of S and E to learn A'. The result in Section 3.3.1.2 (Theorem 2) precisely enables the L^* algorithm to achieve this goal. In particular, since L^* terminates starting from any *valid* table, the assumption learner first obtains a valid table by reusing words in S and E: update T by asking membership queries with regard to U' for each $\rho \in (S \cup S \bullet \widehat{\Sigma}) \bullet E$. The valid table (S, E, T') 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 **GenerateAssumption** module implements this feature (see Figure 5).

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 DLA 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 compositional rule. We denote the modified property[2] at each recursion level of the algorithm by φ'. The old and new assumptions are denoted by A and A', respectively.

Figure 5 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, an assumption

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

A' corresponding to M' and φ' is generated using dynamic L^* such that $M' \parallel A' \sqsubseteq \varphi'$. Lines 6-8 describe recursive invocation of **DynamicCheck** on $\mathcal{M} \setminus M$ against property A'. Finally, Lines 9-15 show how the algorithm detects a counterexample CE and updates A' with it or terminates with a TRUE/FALSE result. The salient features of this algorithm are the following:

- **GenerateAssumption** (Line 5) does not generate new assumptions every time **DynamicCheck** is invoked. Instead, it reuses (by revalidating if necessary) the assumption A computed in the previous compatibility check. When CE is used to update A, **GenerateAssumption** (Line 12) does not need to revalidate A because it had to be validated previously.
- 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, 7, 12). Otherwise, the previously computed result is reused (Lines 4, 8).

```
     DynamicCheck (M', φ') returns counterexample or TRUE
1:   let M' = first element of M';
2:   if (M' = {M'})
3:       if (M ≠ M' or φ ≠ φ') return (M' ⊑ φ');
4:       else return M ⊑ φ;
5:   A' := GenerateAssumption(M', φ');
6:   if (A ≠ A' or M \ M ≠ M' \ M' )
7:       CE := DynamicCheck(M' \ M', A');
8:   else CE := DynamicCheck(M \ M, A);
9:   while(CE is non-empty)
10:      if (M' ∥ CE ⊑ φ')
11:          A' := UpdateAssumption (A',CE);
12:          A' := GenerateAssumption (M', φ');
13:          CE = DynamicCheck (M' \ M', A');
14:      else return a witness counterexample CE to M' ∥ CE ⋢ φ';
15:  return TRUE;
```

Figure 5: Pseudo-Code for Efficient Compatibility Checking

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

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

Proof. The notion of weakest assumptions is used in proving the correctness of **DynamicCheck**. For any DLA M, there must exist a weakest environment assumption DLA WA such that $M \parallel E\varphi$ iff $E \sqsubseteq WA$. Suppose we have a system of components $M_1, \ldots, M_n$ and a global property φ. Consider rules of the form $M_i \parallel A_i \sqsubseteq A_{i-1} (1 \leq i \leq n-1, A_0 = \varphi)$ and $M_n \sqsubseteq A_{n-1}$ as used in the recursive procedure **DynamicCheck** to show that $M_1 \parallel .. \parallel M_n \sqsubseteq \varphi$. It is clear that a weakest assumption WA_1 exists such that $M_1 \parallel WA_1 \sqsubseteq \varphi$. Given WA_1, it follows that WA_2 must exist so that $M_2 \parallel WA_2 \sqsubseteq WA_1$. Therefore, by induction on i, there must exist weakest assumptions WA_i for $1 \leq i \leq n-1$, such that $M_i \parallel WA_i \sqsubseteq WA_{i-1} (1 \leq i \leq n-1, WA_0 = \varphi)$ and $M_n \sqsubseteq A_{n-1}$. Also, by

Theorem 2, **UpdateAssumption**(A, CE) must terminate starting from any valid assumption A' with respect to U' and a counterexample $CE \in L(A') \oplus U'$.

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

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

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

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

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

3.4 Feedback

Recall that for some $i \in \mathcal{I}$, if our containment check detects that $C_i \not\sqsubseteq C'_i$, it also computes a set $\mathcal{F}_i$. Intuitively each element of $\mathcal{F}_i$ represents a behavior of C_i 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 will 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 a DLA M obtained by predicate abstraction of C. By simulating π on M, we construct an alternating sequence $Rep(\pi) = \langle s_1, \alpha_1, \ldots, s_n \rangle$ of states and actions of M corresponding to π. Recall from our earlier discussion of predicate abstraction

(see Section 3.2) that each s_i is of the form $(st_i, \mathcal{V}_i)$, where st_i is a statement of C and $\mathcal{V}_i$ is a predicate valuation. Thus, $Rep(\pi) = \langle (st_1, \mathcal{V}_1), \alpha_1, \ldots, (st_n, \mathcal{V}_n) \rangle$.

We also know that 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, any extension of $Pref(\pi)$ is no longer 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, we produce as output, for each $\pi \in \mathcal{F}$, 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.

3.5 Implementation and Experimental Evaluation

The procedures for checking, in a dynamic manner, the substitutability of components, were implemented in the COPPER model checker [Chaki 05c]. 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. We reused the above features of COPPER 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. Doing this involved keeping multiple learner instances across calls to the verification engine and implementing algorithms to validate multiple, previous observation tables in an efficient way during learning. We also implemented the underapproximation generation algorithms for performing the containment check on small program examples. Doing this involved procedures for implementing must-abstractions from C code using predicates obtained from C components. The automated refinement procedures are still under implementation and would enable containment check of larger benchmarks.

We validated the component substitutability framework while verifying upgrades of a benchmark provided to us by our industrial partner, ABB Inc. [ABB 05]. The benchmarks consist of seven components which together implement an interprocess 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 6 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 6, $\#Mem.\ Queries$ 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**	**T$_{orig}$ (msec)**	**T$_{ug}$ (msec)**
$ipc_1(P_1)$	279	2260	13
$ipc_1(P_2)$	308	1694	14
$ipc_2(P_1)$	358	3286	17
$ipc_2(P_2)$	232	805	10
$ipc_3(P_1)$	363	3624	17
$ipc_3(P_2)$	258	1649	14
$ipc_4(P_1)$	355	1102	24

Figure 6: Summary of Results for **DynamicCheck**

We observed that the previously generated assumptions 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.

4 Related Work

Related projects 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 and colleagues [de Alfaro 01, Chakrabarti 02] 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 DLA models (which are similar to finite-state interface automata) from component implementations. Moreover, we do not require refinement among the old components and their new versions.

McCamant and Ernst [McCamant 04] 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 [Groce 02]. The use of learning for automated assume-guarantee reasoning was proposed originally by Cobleigh and colleagues [Cobleigh 03]. The use of learning along with predicate abstraction has also been applied in the context of interface synthesis [Alur 05] and various types of assume-guarantee proof rules for automated software verification [Chaki 04a].

This work is related to our earlier project [Chaki 04b] 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 report gives a new solution to the containment-check problem presented by Chaki and colleagues [Chaki 04b]. 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 overapproximations and underapproximations obtained via existential and universal abstractions.

5 Conclusion

This report presents results of the SEI IRAD project on verification of evolving software via component-substitutability analysis. It addresses a critical and vital problem of component-substitutability analysis and provides a solution that consists of two phases: (1) containment and (2) compatibility checks. 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 revalidation of the system after an upgrade. The containment check detects behaviors that 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 runtimes and, hence, needs investigation. We would further like to investigate a modification of **DynamicCheck** based on a more efficient L^* algorithm by Rivest and colleagues [Rivest 93] to improve its performance.

The component-substitutability analysis has been implemented in the COPPER tool [Chaki 05c] that can be invoked within the ComFoRT framework. The verification framework was validated on an industrial benchmark provided by our industrial partner, ABB [ABB 05], and the framework demonstrated encouraging results.

References

[Abadi 95] Abadi, M. & Lamport, L. "Conjoining Specifications". *ACM Transactions on Programming Languages and Systems (TOPLAS) 17*, 3 (May 1995): 507–534.

[ABB 05] *ABB*. http://www.abb.com, 2005.

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

[Alur 05] Alur, R.; Cerny, P.; Gupta, G.; Madhusudan, P.; Nam, W.; & Srivastava, A. "Synthesis of Interface Specifications for Java Classes", 98–109. *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. Long Beach, CA, January 12–14, 2005. New York, NY: Association for Computing Machinery (ACM), 2005.

[Angluin 87] Angluin, D. "Learning Regular Sets from Queries and Counterexamples". *Information and Computation 75*, 2 (November 1987): 87–106.

[Ball 00] Ball, T. & Rajamani, S. *Boolean Programs: A Model and Process for Software Analysis* (MSR-TR-2000-14). Redmond, WA: Microsoft Research, February 2000. ftp://ftp.research.microsoft.com/pub/tr/tr-2000-14.pdf.

[Ball 01] Ball, T.; Majumdar, R.; Millstein, T.; & Rajamani, S. "Automatic Predicate Abstraction of C Programs", 203–213. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Snowbird, UT, June 20-22, 2001. New York, NY: Association for Computing Machinery, 2001.

[Chaki 04a] Chaki, S.; Clarke, E.; Giannakopoulou, D.; & Păsăreanu, C. S. *Abstraction and Assume-Guarantee Reasoning for Automated Software Verification* (05.02). Mountain View, CA: Research Institute for Advanced Computer Science (RIACS), 2004.

[Chaki 04b] Chaki, S.; Sharygina, N.; & Sinha, N. "Verification of Evolving Software", 55–61. *Proceedings of the Third Workshop on Specification and Verification of Component Based Systems (SAVCBS)*. Newport Beach, CA, October 31–November 1, 2004. Ames, Iowa: Iowa State University, 2004.

[Chaki 04c] Chaki, S.; Clarke, E.; Groce, A.; Ouaknine, J.; Strichman, O.; & Yorav, K. "Efficient Verification of Sequential and Concurrent C

Programs". *Formal Methods in System Design (FMSD) 25*, 2–3 (September–November 2004): 129–166.

[Chaki 05a] Chaki, S.; Clarke, E.; Sharygina, N.; & Sinha, N. "Dynamic Component Substitutability Analysis", 512–528. *Proceedings of the International Symposium on Formal Methods Europe*, volume 3582 of *Lecture Notes in Computer Science*. New Castle, UK, July 18–22, 2005. New York, NY: Springer-Verlag, 2005.

[Chaki 05b] Chaki, S. *Learning Doubly Labeled Automata Using Queries and Counterexamples*. http://www.sei.cmu.edu/staff/chaki/publications/learn-se-trace.pdf, 2005.

[Chaki 05c] Chaki, S.; Ivers, J.; Sharygina, N.; & Wallnau, K. "The ComFoRT Reasoning Framework", 164–169. *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, 2005.

[Chakrabarti 02] Chakrabarti, A.; de Alfaro, L.; Henzinger, T. A.; Jurdzinski, M.; & Mang, F. Y. C. "Interface Compatibility Checking for Software Modules", 428–441. *Proceedings of the 14th International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*. Copenhagen, Denmark, July 27–31, 2002. New York, NY: Springer-Verlag, 2002.

[Clarke 82] Clarke, E. & Emerson, A. "Design and Synthesis of Synchronization Skeletons for Branching Time Temporal Logic", 52–71. *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Yorktown Heights, New York, May 4–6, 1982. Berlin, Germany: Springer-Verlag, 1982.

[Clarke 86] Clarke, E. M.; Emerson, E. A.; & Sistla, A. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems (TOPLAS) 8*, 2 (1986): 244–263.

[Clarke 89] Clarke, E.; Long, D.; & McMillan, K. "Compositional Model Checking", 353–362. *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS '89)*. Pacific Grove, CA, June 5–8, 1989. Washington, DC: IEEE Computer Society Press, 1989.

[Clarke 92] Clarke, E.; Grumberg, O.; & Long, D. "Model Checking and Abstraction", 343–354. *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. Albuquerque, NM, January 19–22, 1992. New York, NY: Association for Computing Machinery (ACM), 1992.

[Clarke 00a] Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; & Veith, H. "Counterexample-Guided Abstraction Refinement", 154–169. *Proceedings of the 12th International Conference on Computer Aided Verification (CAV '00)*, volume 1855 of *Lecture Notes in Computer Science*. Chicago, IL, July 15–19, 2000. Berlin, Germany: Springer-Verlag, 2000.

[Clarke 00b] Clarke, E. M.; Grumberg, O.; & Peled, D. *Model Checking*. Cambridge, MA: MIT Press, 2000.

[Cobleigh 03] Cobleigh, J. M.; Giannakopoulou, D.; & Păsăreanu, C. S. "Learning Assumptions for Compositional Verification", 331–346. *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*. Warsaw, Poland, April 7–11, 2003. New York, NY: Springer-Verlag, 2003.

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

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

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

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

[Graf 97] Graf, S. & Saïdi, H. "Construction of Abstract State Graphs with PVS", 72–83. *Proceedings of the Ninth International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*. Haifa, Israel, June 22–25, 1997. New York, NY: Springer-Verlag, 1997.

[Groce 02] Groce, A.; Peled, D.; & Yannakakis, M. "Adaptive Model Checking", 357–370. *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*. Grenoble, France, April 8–12, 2002. New York, NY: Springer-Verlag, 2002.

[Kurshan 95] Kurshan, R. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton, NJ: Princeton University Press, 1995.

[McCamant 04] McCamant, S. & Ernst, M. D. "Early Identification of Incompatibilities in Multi-Component Upgrades", 440–464. *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, volume 3086 of *Lecture Notes in Computer Science*. Oslo, Norway, June 14–18, 2004. New York, NY: Springer-Verlag, 2004.

[McMillan 97] McMillan, K. "A Compositional Rule for Hardware Design Refinement", 24–35. *Proceedings of the Ninth International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*. Haifa, Israel, June 22–27, 1997. New York, NY: Springer-Verlag, 1997.

[Nellemann 94] Nellemann, D. *Air Force F-22 Embedded Computers* . http://archive.gao.gov/t2pbat2/152615.pdf, September 1994.

[Pnueli 77] Pnueli, A. "The Temporal Logic of Programs", 46–57. *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. Providence, RI, October 31–November 2, 1977. New York, NY: IEEE Computer Society Press, 1977.

[Pnueli 85] Pnueli, A. "In Transition from Global to Modular Temporal Reasoning About Programs", 123–144. *Logics and Models of Concurrent Systems*. New York, NY: Springer-Verlag, 1985.

[Rivest 93] Rivest, R. L. & Schapire, R. E. "Inference of Finite Automata Using Homing Sequences". *Information and Computation 103*, 2 (1993): 299–347.

[Roscoe 98] Roscoe, A. W. *The Theory and Practice of Concurrency*. New York: Prentice-Hall International, 1998.

[Slabodkin 98] Slabodkin, G. "Software Glitches Leave Navy Smart Ship Dead in the Water". *Government Computer News 17*, 7 (July 13 1998). http://appserv.gcn.com/17_17/news/33727-1.html.

[U.S.-Canada 04] U.S.-Canada Power System Outage Task Force. *Final Report on the August 14 Blackout in the United States and Canada*. https://reports.energy.gov/, April 2004.

[U.S. Govt. 05] *F-35 Joint Strike Fighter Program*. http://www.jsf.mil/, 2005.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	December 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Verification of Evolving Software via Component Substitutability Analysis	FA8721-05-C-0003

6. AUTHOR(S)

Sagar Chaki, Edmund Clarke, Natasha Sharygina, Nishant Sinha

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2005-TR-008

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2005-008

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Formal verification by model checking has the potential to produce major enhancements in the reliability and robustness of software. However, a shortcoming in most model checking research is the failure to consider how to make the use of model checking routine throughout various stages of software development. This report presents results of the Independent Research and Development (IRAD) project on verification of evolving software conducted at the Software Engineering Institute in 2005. The research conducted as part of the IRAD project considered ways to reduce the effort of subsequent verifications. In particular, it resulted in the development of techniques that exploit the results of previous verification efforts and focus only on the portions of the system that have changed (components). Thus, these new techniques incorporate model checking into development processes in a much less intrusive or cumbersome manner than previous verification techniques.

The report presents an automated and compositional procedure to solve the component substitutability problem. The solution contributes two techniques for checking the correctness of software upgrades: (1) a technique based on simultaneous use of overapproximations and underapproximations obtained via existential and universal abstractions and (2) a dynamic assume-guarantee reasoning algorithm in which 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, the solution generates constructive feedback that shows developers how to improve the components. The substitutability approach has been implemented and validated in the Component Formal Reasoning Technology (COMFORT) model checking tool set. The experimental evaluation of an industrial benchmark demonstrates encouraging results.

14. SUBJECT TERMS	15. NUMBER OF PAGES
component substitutability, model checking, software development, independent research and development, IRAD, verification	40

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL