# Assume-Guarantee Verification of Evolving Component-Based Software

by

Pham Ngoc Hung

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

*Supervisor:* Associate Professor Toshiaki Aoki

*School of Information Science*
*Japan Advanced Institute of Science and Technology*

September, 2009

## Abstract

Assume-guarantee verification method has been recognized as a promising approach to verify component-based software (CBS) with model checking. The method is not only fitted to component-based software but also has a potential to solve the *state space explosion problem* in model checking. This method allows us to decompose a verification target into components so that we can model check each of them separately. Model-based verification methods in general and the assume-guarantee verification method in particular of a system is performed with respect to its model which exactly describes the behavior of the system. This means that we have to obtain the accurate model of the system before applying the verification techniques. However, these methods generally assume that the ways to obtain the model and its correctness are available. This means that the model-based verification methods assume the availability and correctness of the model which describes the behavior of the system under study. Nonetheless, this assumption may not always hold in practice due to the modelling errors, bug fixing, etc. Even if the assumptions hold, the model could be invalidated when the software is evolved by adding or removing some behaviors because evolving of the existing components seems to be an unavoidable task during the software life cycle. Unfortunately, the consequence of these tasks is the whole evolved software must be rechecked. The purpose of this research is to provide a effective framework for modular verification of evolving component-based software in the context of the component evolution. When a component is evolved after adapting some refinements, the proposed framework focuses on this component and its model in order to update the model and to recheck the whole evolved system. The framework also reuses the previous verification results and the previous models of the evolved components to reduce the number of steps required in the model update and modular verification processes.

This dissertation has three main contributions. The first contribution of the research is to propose a method for generating minimal assumptions for the assume-guarantee verification of component-based software. The proposed method is an improvement of the L*-based assumption generation method. The key idea of this method is finding the minimal assumptions in the search spaces of the candidate assumptions. These assumptions are seen as the environments needed for the components to satisfy a property and for the rest of the system to be satisfied. The minimal assumptions generated by the proposed method can be used to recheck the whole system at much lower computational cost.

The second contribution is to propose an effective framework for assume-guarantee verification of component-based software in the context of the component evolution at the design level. In this framework, if a component model is evolved after adapting some refinements, the whole component-based software of many existing component models and the evolved component model is not required to be rechecked. The framework only checks whether the evolve model satisfies the assumption of the old system. If it is, the evolved component-based software still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, a new assumption is regenerated. We propose two methods for new assumption regeneration: assumption regeneration and minimized assumption regeneration. The methods reuses the current assumption as the previous verification result to regenerate the new assumption at much lower computational cost.

The third contribution of the research is to propose a framework for modular conformance testing and modular verification of component-based software in the context of component evolution at the source code level. This framework includes two stages: modular conformance testing for updating inaccurate models of the evolved components and modular verification for evolving component-based software. When a component is evolved after adapting some refinements, the proposed framework focuses on this component and its model in order to update the model and to recheck the whole evolved system. The framework also reuses the previous verification results and the previous models of the evolved components to reduce the number of steps required in the model update and modular verification processes.

**Keyword:** verification, model checking, assume-guarantee reasoning, assume-guarantee verification, modular verification, component evolution, conformance testing, learning algorithm, assumption, component-based software.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Component-based development is one of the most important technical initiatives in software engineering as it is considered to be an open, effective and efficient approach to reduce development cost and time while increasing software quality. Component-based software (CBS) technology also supports rapid development of complex evolving CBS by enhancing reuse and adaptability. CBS can be evolved by evolving one or more software components. As a result, most large-scale information systems today are built based on software component technology [3, 4, 6]. With component technology, applications are developed by composing well-defined independent components together. Components used in an application can be developed by the application developers or can be provided by third parties.

To realize such an ideal CBS paradigm, one of the key issues is to ensure that those separately specified and implemented components do not conflict with each other when composed - the *component consistency* issue. The current well-known technologies such as CORBA (OMG), COM/DCOM or .NET (Microsoft), Java and JavaBeans (Sun), etc. only support component *plugging*. However, components often fail to co-operate, i.e., the *plug-and-play* mechanism fails. Currently, the popular solution to dealing with this issue is the verification of CBS via model checking [1, 2]. Model checking has been recognized as a practical approach for improving software reliability. It verifies that whether the formal description of a system satisfies a formal specification. The formal description called model is generally represented by a state transition system and the model checking process decides that if the model satisfies the required properties by exploring successive states of the system. Despite significant advances in the development of model checking, it remains a difficult problem called the *state space explosion* in the hands of experts to make it scale to the size of industrial systems. In order to dealing with this problem, a powerful method called assume-guarantee verification was proposed in [7, 8, 10, 16] by decomposing a verification target about a component-based system into smaller parts about the individual components such that we can model check each of them. The key idea of this method is to generate assumptions as environments needed for components to satisfy a property. These assumptions are then discharged by the rest of the system. In the method, the number of states of the assumptions should be minimized because the computational cost of model checking is influenced by that number. This method also should be improved for recheck the evolved CBS in the context of the component evolution because it is rather closed for the static systems. Thus it is not prepared for future changes.

Model-based verification techniques have been recognized as the promising approaches in improving the software reliability. Model-based verification of a system is performed with respect to its model which exactly describes the behavior of the system. This means that we have to obtain the accurate model of the system before applying the verification techniques. Currently, these techniques generally assume that the ways to obtain a model of the system under checking and its correctness are available. This means that this model is available and accurate. However, these assumptions may not always hold in practice due to the modelling errors, bug fixing, etc. Even if the assumptions hold, the model could be invalidated when software is evolved by adding and removing some behaviors because evolving of existing components seems to be an unavoidable task during the software life cycle. Unfortunately, the consequence of the tasks is the whole evolved software must be rechecked. Moreover, the CBS verification is a difficult target due to the frequent lack of information about software components that may be provided by third parties without source codes and with incomplete documentations. Even if we have source code and complete documentation, it is very hard to understand them. The best way is to see the software component implementations as black boxes. In this case, obtaining accurate models which accurately describe behaviors of the software components under study is an interesting problem because verification of a system is performed with respect to its accurate model. There are some works that have been recently proposed in obtaining the accurate models of software systems [5, 12, 46]. In those works, the L* learning algorithm is used to generate models of software systems which can then be analyzed with model checking and testing techniques. When the software is evolved after adapting some refinements, its model may be inaccurate. The works reuse a part of the model to obtain an accurate model of the evolved software. However, the works are not prepared for modular verification because the generated models describes the behaviors of the whole software. As a result, the *state space explosion* problem may occur when rechecking large-scale software. Rechecking the evolved component-based software has also been investigated in [21, 22, 23]. The work focuses on component substitutability directly from the verification point of view. The purpose of this work is to provide an effective verification procedure that decides whether a component can be replaced with a new one without violation. For each upgraded component, this work uses abstraction techniques to obtain a new model of the component. This means that the new model is obtained from scratch. It should be better to reuse the model of the old component to obtain the new model. Consequently, our main motivation is to study an effective framework for modular verification of component-based software in the context of the component evolution.

## 1.2 Problem Statement

Although there are many works that have been recently proposed in assume-guarantee verification [7, 8, 10, 11, 14, 16, 39], our research focuses on the method proposed in [10] because it has been recognized as a promising, incremental and fully automatic fashion for modular verification of component-based software. This work proposes an iterative method based on the L* learning algorithm for learning regular languages. The learning process is based on queries to a software component, and on counterexamples obtained by model checking the component and its environment, alternately. At each iteration, the method may conclude that the required property is satisfied or violated in the system analyzed. This process is guaranteed to terminate and it converges to an assumption such that the assumption is strong enough for the component to satisfy the property and weak enough to be discharged by the rest of the CBS. However, the L* learning algorithm often terminates before reaching this point, and returns the first assumption that satisfies the requirements of the verification. Moreover, the assumptions generated by this method are not minimal. As mentioned above, the number of states of the generated assumptions should be minimized because this number influences on the computational cost of model checking. Thus, our first aim is to optimize the method in order to generate minimal assumptions for the assume-guarantee verification of component-based software.

Consider a popular architecture of component-based software where the CBS contains a base component as a fixed framework, and some extensional components. In this kind of CBS, the component evolution occurs only on the extensional components. It is known that the component evolution is an unavoidable task during the software life cycle. When an extensional component is evolved after adapting some refinements, the whole CBS of many existing components and the evolved component is required to be rechecked. Suppose that models which exactly describes the behaviors of the software components are available at the design level. In order to recheck the evolved CBS, we can apply the described method proposed in [10] by rechecking the evolved CBS as a new system from scratch. However, rechecking of the whole evolved CBS is unnecessary because the evolution often focuses on a few existing components. It should be better to focus only on the evolved components and try to reuse previous verification results to verify the evolved CBS. Therefore, the next aim of our research is to propose an effective framework for assume-guarantee verification of component-based software in the context of the component evolution at the design level.

Obtaining accurate models of the evolved components and rechecking of evolving software systems has been investigated in the study about adaptive model checking (AMC) [12] which necessitates an iterative construction of a model for software by applying a learning algorithm called L* [9, 17]. However, the model in AMC describes the behavior of the whole software. In order to recheck the evolved CBS, the *state space explosion* problem may occur when checking large-scale software. In this case, rechecking of the whole evolved CBS is unnecessary. It should be better to focus only on the evolved components and try to reuse previous verification results to verify the evolved CBS system. Moreover, the AMC approach can not reuse the whole given model because it does not ensure the achievement of an updated model from the inaccurate model because the *software evolution* concept in AMC means adding some new behaviors and removing some existing behaviors. Furthermore, when system is changed, the model is required to update including comparisons of software with the new candidate model via the Vasilevskii-Chow (VC) algorithm [45, 47]. If the model is inaccurate then updating the whole model is not necessary (and very expensive) because the changes often focus on a few existing components with small changes. Consequently, our third aim is to propose a framework for modular conformance testing and assume-guarantee verification of evolving component-based software to dealing with the above issues in the context of the component evolution. Suppose that there is a simple component-based software which contains a base component $C_1$ as a fixed framework, and a component $C_2$ as an extension. The extension $C_2$ is plugged into the framework $C_1$ via some mechanisms. Let $M_1$ and $M_2$ be accurate models of $C_1$ and $C_2$ respectively. It is known that the compositional system $M_1 \| M_2$ satisfies the property $p$. During the life cycle of this system, the extension $C_2$ is evolved to a new component $C_2'$ by adding some new behaviors to $C_2$. In this case, the current model $M_2$ may be an inaccurate model of the evolved component $C_2'$. We proposes a method called modular conformance testing to compare $C_2'$ with $M_2$. If they are not in conformance, $M_2$ is used as the initial model for the L* algorithm to obtain an accurate model $M_2'$ for the evolved component $C_2'$. The evolved CBS then must be rechecked for whether it satisfies the property $p$ or not. For this purpose, we only check whether the evolve model $M_2'$ satisfies the assumption $A(p)$ of the CBS before evolving. If it is, the evolved CBS still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by $M_2'$, a new assumption is regenerated in a effective way. The process for rechecking the evolved CBS in this context is presented in Figure 1.1.

## 1.3 Contributions

This dissertation has three main contributions as follows.

The first contribution of the research is to propose a method for generating minimal assumptions for the assume-guarantee verification of component-based software. The proposed method is an improvement of the L*-based assumption generation method. The key idea of this method is finding the minimal assumptions in the search spaces of the candidate assumptions. These assumptions are seen as the environments needed for the components to satisfy a property and for the rest of the system to be satisfied. The minimal assumptions generated by the proposed method can be used to recheck the whole system at much lower computational cost.

The second contribution is to propose an effective framework for assume-guarantee verification of component-based software in the context of the component evolution at the design level. In this framework, if a component model is evolved after adapting some refinements, the whole component-based software of many existing component models and the evolved component model is not required to be rechecked. The framework only checks whether the evolve model satisfies the assumption of the old system. If it is, the evolved component-based software still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, a new assumption is regenerated. We propose two methods for new assumption regeneration: assumption regeneration and minimized assumption regeneration. The methods reuses the current assumption as the previous verification result to regenerate the new assumption at much lower computational cost.

The third contribution of the research is to propose a framework for modular conformance testing and modular verification of component-based software in the context of component evolution at the source code level. This framework includes two stages: modular conformance testing for updating inaccurate models of the evolved components and modular verification for evolving component-based software. When a component is evolved after adapting some refinements, the proposed framework focuses on this component and its model in order to update the model and to recheck the whole evolved system. The framework also reuses the previous verification results and the previous models of the evolved components to reduce the number of steps required in the model update and modular verification processes.

## 1.4 Dissertation Organization

The dissertation is organized as follows.

The first chapter is the introduction about the research. This chapter sets the context, describes the motivation and the main contributions, and presents the structure of the remaining part of the dissertation.

The second chapter is about the background knowledge related to the research. This chapter includes the model and component specifications, safety property and satisfiability, assume-guarantee reasoning, L* learning algorithm, component evolution, and others related concepts.

Chapter 3 presents a method for generating minimal assumptions for the assume-guarantee verification of component-based software. In this chapter, we define a new technique for answering membership queries which are needed for generating the minimal assumptions. Some improvements of the proposed method are discussed in order to reduce the computational cost for generating the minimal assumptions. An implemented tool for generating the minimal assumptions and experimental results are also presented and discussed.

In chapter 4, we propose an effective framework for modular verification of evolving component-based software at the design level. In this framework, the whole evolved component-based software of many existing components and the evolved component is rechecked by focusing only on checking the evolved component model. If the evolve model satisfies the current assumption of the old CBS, the whole evolved CBS still satisfies the required property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, a new assumption is regenerated by using one of the two proposed methods for assumption regeneration. A tool for assumption regeneration is also presented in this chapter.

Still based on the concept of component evolution, in Chapter 5, we propose a method called modular conformance testing (MCT) for testing conformance between the evolved component and its model and updating the inaccurate model. After that, the evolved CBS is rechecked by applying the proposed framework presented in Chapter 4. We also integrate the MCT method and assume-guarantee verification into a framework for modular conformance testing and modular verification of evolving component-based software at the source code level.

Finally, we conclude the research and present the future works in Chapter 6.

In this dissertation, Chapter 3, 4 and 5 are the main work of our research. Figure 1.1 presents the position and relation of these chapters in the modular verification process of component-based software in the context of the component evolution. When a software component is evolved, the evolved component-based software is required to recheck that whether it still satisfies a property. We first perform modular conformance testing to compare the evolved component with its model. If the model is inaccurate then it is used as the initial model for the L* learning algorithm in order to update itself. Otherwise, the component and its model are in conformance. The accurate models of the evolved CBS

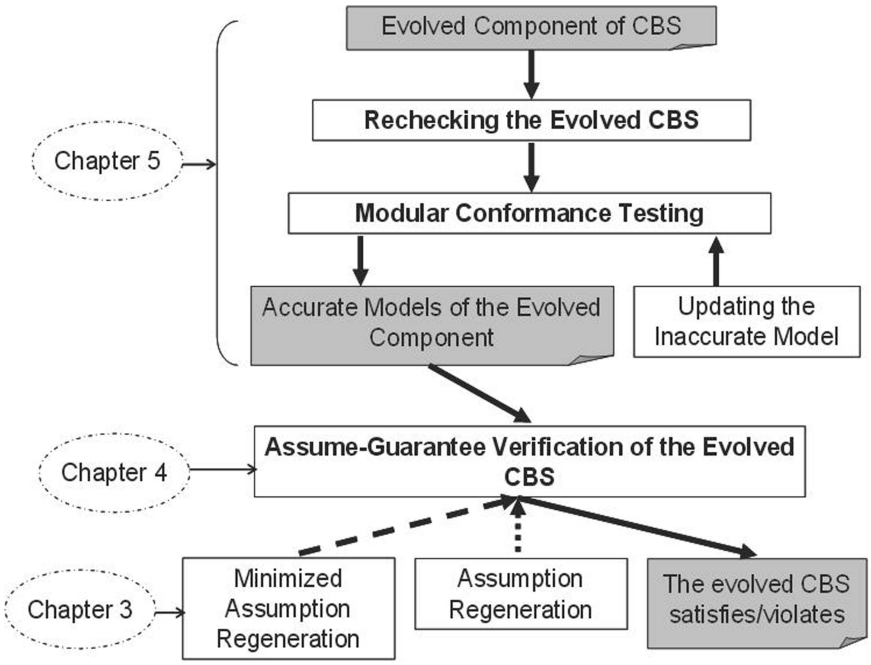



Figure 1.1: The relations between the main chapters of the dissertation and the process for modular verification of evolved CBS.

then are used to recheck the evolved CBS by applying the assume-guarantee verification method. If the model of the evolve component satisfies the assumption of the CBS before evolving, the evolved CBS still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by the model, a new assumption is regenerated. One of the two proposed methods for new assumption regeneration (i.e., assumption regeneration method and minimized assumption regeneration method) is applied to regenerate the new assumption. The applied method returns a new assumption if the evolved CBS satisfies the property, and a counterexample *cex* otherwise.

# Chapter 2

# Background

This chapter reviews basic concepts from the theory of assume-guarantee verification for component-based software in the context of the component evolution. Most of these notions can be found in [10, 12, 16]. Details of the LTSA tool can be found in the text book by J. Magee and J. Kramer [13].

## 2.1 Model and Component

In this dissertation, the component models which describe the behaviors of communicating components are represented by the *Labeled Transition Systems* (LTSs). A LTS is a directed graph with labeled edges. In addition to states and transitions, a set of labels called alphabet is associated with the system. All labels on transitions must be from that alphabet. Let $\mathcal{A}ct$ be the universal set of observable actions and let $\tau$ denote a local/internal action unobservable to a component's environment. We use $\pi$ to denote a special error state, which models the fact that a safety violation has occurred in the associated system. We require that the error state has no outgoing transition. A LTS is defined as follows:

### 2.1.1 Labeled Transition Systems

A LTS $M$ is a quadruple $\langle Q, \alpha M, \delta, q_0 \rangle$ where:

- $Q$ is a non-empty set of states,
- $\alpha M \subseteq \mathcal{A}ct$ is a finite set of observable actions called the alphabet of $M$,
- $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$ is a transition relation, and
- $q_0 \in Q$ is the initial state.

The size of a LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is the number of states of $M$, denoted $|M|$ (i.e., $|M| = |Q|$).

We use $\prod$ to denote the LTS $\langle \{\pi\}, Act, \phi, \pi \rangle$. An LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is *non-deterministic* if it contains $\tau$-transition or if $\exists (q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, $M$ is *deterministic*.

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q_0' \rangle$. We say that $M$ transits into $M'$ with action a, denoted $M \longrightarrow^a M'$ if and only if $(q_0, a, q_0') \in \delta$ and $\alpha M = \alpha M'$ and $\delta = \delta'$.

### 2.1.2 Traces

A trace $\sigma$ of a LTS $M$ is a sequence of observable actions that $M$ can perform starting at its initial state. For $\Sigma \subseteq \mathcal{A}ct$, we use $\sigma \uparrow \Sigma$ to denote the trace obtained by removing from $\sigma$ all occurrences of actions $a \notin \Sigma$. The set of all traces of $M$ is called the language of $M$, denoted $L(M)$.

Let $\sigma = a_1 a_2 ... a_n$ be a finite trace of a LTS $M$. We use $[\sigma]$ to denote the LTS $M_\sigma = \langle Q, \alpha M, \delta, q_0 \rangle$ with $Q = \langle q_0, q_1, ..., q_n \rangle$, and $\delta = \{(q_{i-1}, a_i, q_i)\}$, where $1 \leq i \leq n$. We say that an action $a \in \alpha M$ is enabled from a state $s \in Q$, if there exists $s' \in Q$, such that $(s, a, s') \in \delta$. Similarly, a trace $a_1 a_2 ... a_n$ is enabled from $s$ if there is a sequence of states $s_0$, $s_1$, ..., $s_n$ with $s_0 = q_0$ such that for $1 \leq i \leq n$, $(s_{i-1}, a_i, s_i) \in \delta$.

### 2.1.3 Parallel Composition

The parallel composition operator $\|$ is a commutative and associative operator that combines the behavior of two component models by synchronizing the actions common to their alphabets and interleaving the remaining actions. Consider two LTSs; $M_1 = \langle Q_1, \alpha M_1, \delta_1, q_0^1 \rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q_0^2 \rangle$. The *parallel composition* between $M_1$ and $M_2$, denoted $M_1 \| M_2$, is defined as follows. If $M_1 = \prod$ or $M_2 = \prod$, then $M_1 \| M_2 = \prod$. Otherwise, $M_1 \| M_2$ is a labeled transition system $M = \langle Q, \alpha M, \delta, q_0 \rangle$ where $Q = Q_1 \times Q_2$, $\alpha M = \alpha M_1 \cup \alpha M_2$, $q_0 = q_0^1 \times q_0^2$, and the transition relation $\delta$ is given by the rules:

$$(i)\frac{\alpha \in \alpha M_1 \cap \alpha M_2, (p, \alpha, p') \in \delta_1, (q, \alpha, q') \in \delta_2}{((p, q), \alpha, (p', q')) \in \delta} \tag{2.1}$$

$$(ii)\frac{\alpha \in \alpha M_1 \backslash \alpha M_2, (p, \alpha, p') \in \delta_1}{((p, q), \alpha, (p', q)) \in \delta} \tag{2.2}$$

$$(iii)\frac{\alpha \in \alpha M_2 \backslash \alpha M_1, (q, \alpha, q') \in \delta_2}{((p, q), \alpha, (p, q')) \in \delta} \tag{2.3}$$

For example, when composing the two component models represented by two LTSs *Input* and *Output* illustrated in Figure 2.1, the actions *send* and *ack* will be synchronized while the others will be interleaved. By removing all states which unreachable from the initial state $(0, a)$ and their ingoing transitions, we obtain the parallel composition LTS $Input \| Output$ shown in this Figure.

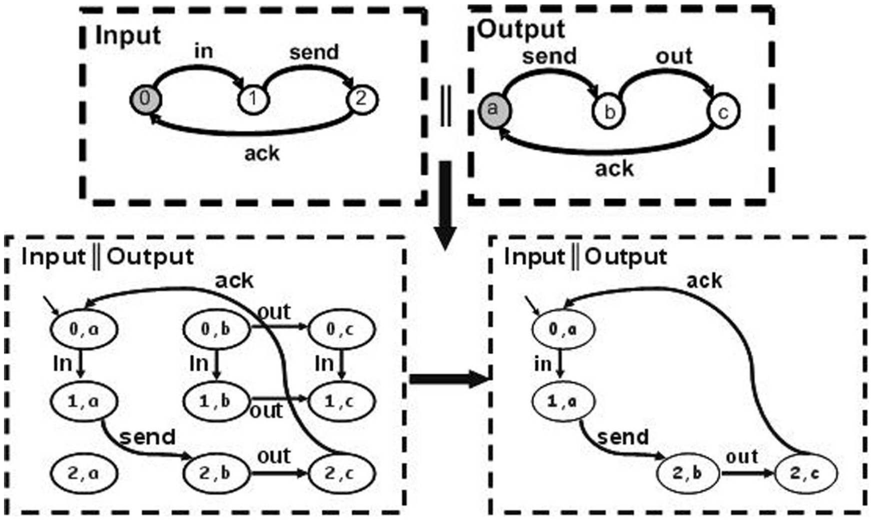



Figure 2.1: An illustration of parallel composition.

### 2.1.4 Safety LTS, Safety Property, Satisfiability, and Error LTSs

We call a deterministic LTS that contains no $\pi$ states a *safety LTS. A safety property* is specified as a safety LTS $p$, whose language $L(p)$ defines the set of acceptable behaviors over $\alpha p$. An LTS $M$ satisfies $p$, denoted as $M \models p$, if and only if $\forall \sigma \in L(M)$: $(\sigma \uparrow \alpha p) \in L(p)$. When checking the LTS $M$ which satisfies the property $p$, an *error LTS*, denoted $p_{err}$, is created which traps possible violations with the $\pi$ state. Formally, the *error LTS* of a property $p = \langle Q, \alpha p, \delta, q_0 \rangle$ is $p_{err} = \langle Q \cup \{\pi\}, \alpha p_{err}, \delta', q_0 \rangle$, where $\alpha p_{err} = \alpha p$ and $\delta' = \delta \cup \{(q, a, \pi) \mid a \in \alpha p$ and $\nexists q' \in Q : (q, a, q') \in \delta\}$.

The error LTS is complete, meaning each state other than the error state has outgoing transitions for every action in the alphabet. For example, Figure 2.2 describes the LTS of a property $p$ and the corresponding error LTS $p_{err}$. The property $p$ means that the *int* action has to occur before *out* action. It captures a desired behavior of the concurrent system containing two component models *Input* and *Output* shown in Figure 2.1. The error LTS $p_{err}$ is created from the safety LTS $p$ by applying the above definition. The dashed arrows illustrate the transitions to the error state that are added to the property to obtain LTS $p_{err}$ [10].

To verify a component $M$ satisfying a property $p$, both $M$ and $p_{err}$ are represented by safety LTSs, the parallel composition $M \| p_{err}$ is then computed. If state $\pi$ is reachable in the composition then $M$ violates $p$. Otherwise, it satisfies.

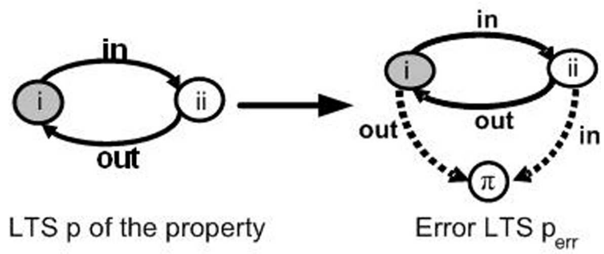



Figure 2.2: The LTS $p$ of the property and the corresponding error LTS $p_{err}$.

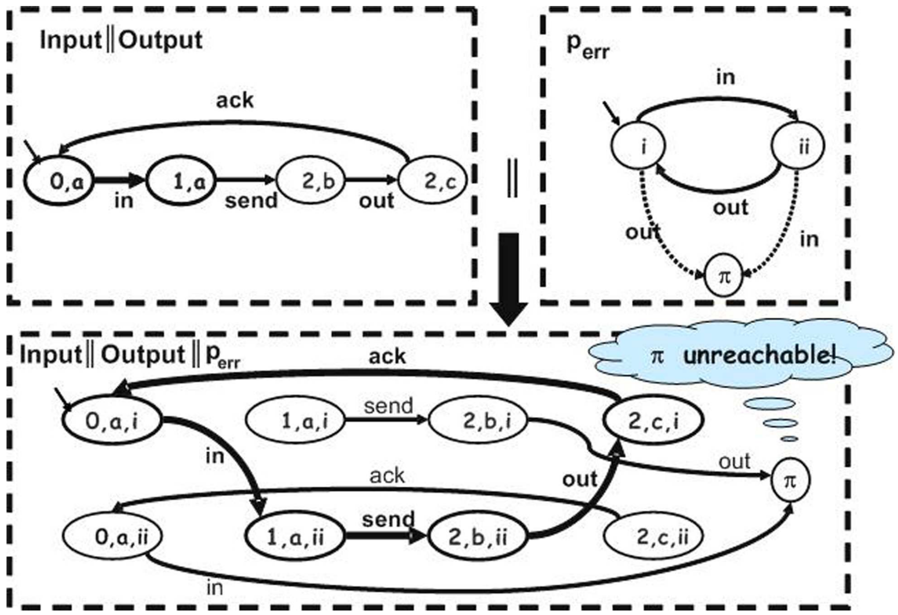



Figure 2.3: Computing the composition $Input \| Output \| p_{err}$.

For example, in order to verify the composition system $Input \| Output$ whether it satisfies the property $p$, the parallel composition $Input \| Output \| p_{err}$ is computed in Figure 2.3. It's easy to check that the error state $\pi$ is not reachable in this composition. Thus, we conclude that the compositional system $Input \| Output$ satisfies the property $p$.

### 2.1.5 Component

We view a software component $C = (\Sigma, T)$ as a prefix closed set of strings $T \subseteq \Sigma^*$ over a finite alphabet of actions $\Sigma$. $T$ is a prefix closed set if and only if for every $v \in T$ then any prefix of $v$ is in $T$. The strings in $T$ reflect the allowed executions of $C$ [12]. In order to check conformance between $C$ and its model, we assume that the proposed method can perform the following experiments on $C$:

- Reset the component to its initial state (called **Reset**). The current experiment is reset to the empty string $\epsilon$

- Check whether an action $a$ can currently be executed by the component $C$. The action $a$ is added to the current experiment. We assume that the component provides us with information on whether $a$ was executable. If the current experiment was $v \in T$ so far, then by attempting to execute $a$, we check whether $va \in T$. If this is so, then the current experiment becomes $va$. Otherwise it remains $v$.

### 2.1.6 Accurate Model

Let $M$ be a component model which describes behavior of a software component $C$. The component model $M$ accurately models the software component $C$ if for every $v \in \Sigma^*$, $v$ is a successful experiment (after applying a **Reset**) on $C$ exactly when $v$ is a trace of $M$.

## 2.2 Deterministic Finite State Automata (DFAs)

In this dissertation, we use a learning algorithm called L* [9, 17] to generate an assumption from two component models. A framework for assumption generation will be described in Chapter 3. In the framework presented in Figure 3.5, at each iteration $i$, the *Learning* module produces a Deterministic Finite State Automata (DFA) $M_i$ such that it is unique and minimal automata and $L(M_i) = L(A_W)$, where $A_W$ is the weakest assumption under which $F$ satisfies the property $p$, defined in [16]. The DFA $M_i$ then is transformed into a candidate assumption $A_i$, where $A_i$ is represented by a safety LTS. A Deterministic Finite State Automata is defined as follows:

A DFA $M$ is a five tuple $\langle Q, \alpha M, \delta, q_0, F\rangle$ where:

- $Q$, $\alpha M$, $\delta$, $q_0$ are defined as for deterministic LTSs, and
- $F \subseteq Q$ is a set of accepting states.

For a DFA $M$ and a string $\sigma$, we use $\delta(q, \sigma)$ to denote the state that $M$ will be in after reading $\sigma$ starting at state $q$. A string $\sigma$ is said to be *accepted* by a DFA $M$ $= \langle Q, \alpha M, \delta, q_0, F\rangle$ if $\delta(q_0, \sigma) \in F$. The language of a DFA $M$ is defined as $L(M) = \{\sigma \mid \delta(q_0, \sigma) \in F\}$.

Figure 2.4 describes an illustration of DFA $M$, where:

- $q_0$ is initial state,
- $Q = \{q_0, q_1\}$,
- $\alpha M = \{a, b\}$,
- $\delta = \{(q_0, a, q_1), (q_0, b, q_1), (q_1, a, q_1), (q_1, b, q_0)\}$, and

- $F = \{q_1\}$.

It's easy to check that the string $aaaa \in L(M)$ but the string $aaaab \notin L(M)$.

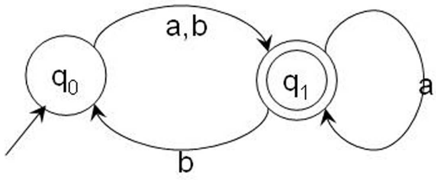



Figure 2.4: An illustration of DFA.

A DFA $M$ is *prefix-closed* if $L(M)$ is prefix-closed, i.e., for every $\sigma \in L(M)$, every prefix of $\sigma$ is also in $L(M)$. The DFAs returned by the learning algorithm in the proposed approach are *complete, minimal, and prefix-closed.* These DFAs therefore contains a single non-accepting state *nas*.

To get a safety LTS $A$ from a DFA $M$, we remove the non-accepting state *nas* and all its ingoing transitions. Formally, we can define the way to transform a DFA $M$ to a safety LTS $A$ as follows:

Let a DFA $M = \langle Q \cup \{nas\}, \alpha M, \delta, q_0, Q \rangle$, the safety LTS $A = \langle Q, \alpha M, \delta \cap (Q \times \alpha M \times Q), q_0 \rangle$. For example, Figure 2.5 describes an illustrative example to transform a DFA $M$ into a safety LTS $A$.

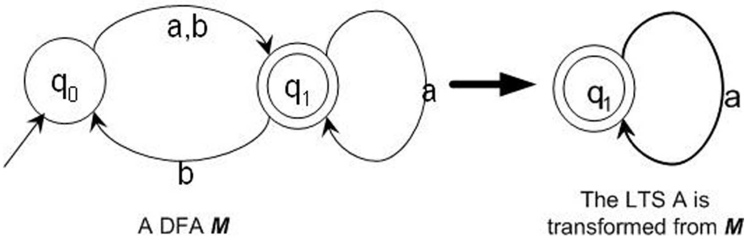



Figure 2.5: An illustration of getting the safety LTS $A$ from the DFA $M$.

## 2.3 Assume-Guarantee Verification

### 2.3.1 Assume-Guarantee Reasoning

The assume-guarantee paradigm is based on a powerful *divide-and-conquer* mechanism for decomposing a verification task about a system into subtasks about the individual components of the system. The key to assume-guarantee reasoning is to consider each component not in isolation, but in conjunction with assumptions about the context of the component. Assume-guarantee principles are known for purely concurrent contexts, which constrain the input data of a component, as well as for purely sequential contexts, which constrain the entry configurations of a component.

In the assume-guarantee paradigm, a formula is a triple $\langle A(p)\rangle\ M\ \langle p\rangle$, where $M$ is a component, $p$ is a property, and $A(p)$ is an assumption about the environment of $M$. The formula is *true* if whenever $M$ is a part of a system satisfying $A(p)$, then the system must also guarantee $p$. In our work, to check an assume-guarantee formula $\langle A(p)\rangle\ M\ \langle p\rangle$, where both $A(p)$ and $p$ are safety LTSs, we use a tool called LTSA [13] to compute $A(p)\|M\|p_{err}$ and check if state $\pi$ is reachable in the composition. If it is, then the formula is violated, otherwise it is satisfied.

The assumption $A(p)$ is generated by applying the L* learning algorithm such that $A(p)$ is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$ (i.e., $\langle A(p)\rangle\ M_1\ \langle p\rangle$ and $\langle \text{true}\rangle\ M_2\ \langle A(p)\rangle$, called compositional rules, both hold). From these rules, this system satisfies $p$. Formally, given $M_1$, $M_2$ and $p$, assume-guarantee reasoning finds an assumption $A(p)$ such that $L(A(p)\|M_1)\uparrow\alpha p \subseteq L(p)$ and $L(M_2)\uparrow\alpha A(p) \subseteq L(A(p))$. The iterative fashion for generating $A(p)$ is illustrated in Figure 3.5. Details of this fashion will be presented in Chapter 3.

### 2.3.2 Weakest Assumption

An assumption with which the compositional rules is guaranteed to return conclusive results is the weakest assumption $A_W$ defined in [16], which restricts the environment of $M_1$ no more and no less than necessary for $p$ to be satisfied. Assumption $A_W$ describes exactly those traces over the alphabet $\Sigma = (\alpha M_1 \cup \alpha p) \cap \alpha M_2$ which, the error state $\pi$ is not reachable in the compositional system $M_1\|p_{err}$. Weakest assumption $A_W$ means that for any environment component $E$, $M_1\|E\models p$ if and only if $E\models A_W$.

### 2.3.3 Minimal Assumption

The number of states of the assumptions generated by the current method for assume-guarantee verification proposed in [7, 8, 10, 16] is not mentioned. Thus, the assumptions generated by the method are not minimal. Our first aim of the dissertation is to optimize the method in order to generate minimal assumptions for the assume-guarantee verification of component-based software. The concept about minimal assumption is defined as follows.

Given two component models $M_1$, $M_2$ and a property $p$, $A(p)$ is an assumption if and only if $A(p)$ satisfies the compositional rules. *An assumption $A(p)$ represented by a LTS is minimal if and only if the number of states of $A(p)$ is less than or equal to the number of states of any other assumptions.*

### 2.3.4 LTSA

The Labelled Transition Systems Analyzer (LTSA) [13] is an automated tool that supports Compositional Reachability Analysis (CRA) of a concurrent software based on its architecture. In general, the software architecture of a concurrent software has a hierarchical structure. CRA incrementally computes and abstracts the behaviors of composite components based on the behaviors of their immediate children in the hierarchy. Abstraction consists of hiding the actions that do not belong to the interface of a component, and minimizing with respect to observational equivalence.

The input language "*Finite State Processes (FSP)*" of this tool is a process-algebra style notation with Labelled Transition Systems (LTS) semantics. A property is also expressed as an LTS, but with extended semantics, and is treated as an ordinary component during composition. Properties are combined with the components to which they refer. They do not interfere with system behaviors, unless they are violated. In the presence of violations, the properties introduced may reduce the state space of the (sub)systems analyzed.

The LTSA tool also features graphical display of LTSs, interactive simulation and graphical animation of behavior models, all helpful aids in both design and verification of system models.

This dissertation uses the LTSA tool to check correctness of the assumptions generated by our implemented tools. For this purpose, we check that whether a generated assumption $A(p)$ satisfies the compositional rules (i.e., $\langle A(p)\rangle\ M_1\ \langle p\rangle$ and $\langle \text{true}\rangle\ M_2\ \langle A(p)\rangle$ both hold) by checking the compositional systems $A(p)\|M_1\|p_{err}$ and $M_2\|A(p)_{err}$ via the LTSA tool. If the LTSA tool returns the same result as our verification result for each illustrative system, the generated assumption $A(p)$ is correct.

## 2.4 Component Evolution

Component evolution is an important concept in software engineering. It is a general notion and there are many meanings of this concept, depending on the context in which it is used. For example, in analysis and design software, the component evolution concept expresses the relationship between a specification of a component ($A_S$) and it's implementation ($A_I$). In this case, the evolution means that more detailed information is added. The relation "$A_I$ evolves $A_S$" is intuitively meant to say that "the specification $A_S$ has more behavioral options than it's implementation $A_I$," or equivalently, "every behavioral option realized by the implementation $A_I$ is allowed by the specification $A_S$". In the object-oriented programming, component evolution means adding some methods or attributes or constraints into a class. In the open incremental model checking (OIMC) approach proposed in [11, 14, 15], this concept means adding (or plugging) a new component as an (*extension*) into the *Base* component via compatible interface states.

In this dissertation, we define a new concept in the *component evolution*: adding only some new behaviors to the old component without losing the old behaviors. Even for this work where we consider the software components as black boxes, the components can be represented by LTSs. Intuitively, evolving the component $C_2$ to a new component $C'_2$ means that the component $C'_2$ is created by adding some states and transitions to $C_2$. Formally, we can define the evolution relation between $C_2$ and $C'_2$ as follows. Let $C_2 = \langle Q_2, \alpha C_2, \delta_2, q_0^2\rangle$ and $C'_2 = \langle Q'_2,\ \alpha C'_2,\ \delta'_2,\ q_0'^2\rangle$ be two components. $C'_2$ is an evolution of $C_2$ if and only if $Q_2 \subseteq Q'_2$, $\alpha C_2 = \alpha C'_2$, $\delta_2 \subseteq \delta'_2$, and $q_0^2 = q_0'^2$. Equivalently, if the new component $C'_2$ is an evolution of $C_2$, it implies that $L(C_2) \subseteq L(C'_2)$. During the software life-cycle, suppose that we have a mechanism to manage the old versions of the software components. A new software component version is produced by adding some new behaviors to the current version. When removing some old behaviors of the component, the new version is exactly one of the old versions. This means that only adding some new behaviors is enough for the software component evolution. By this definition, we ensure that we achieve an accurate model $M'_2$ of the evolved component $C'_2$ by reusing the entire inaccurate model $M_2$ of the old component $C_2$.

For example, the model $M_2$ of a component $C_2$ is evolved to the evolved model $M'_2$ of the evolved component $C'_2$ illustrated in Figure 2.6. After some data is sent to $M_2$, it produces output using the action *out* and acknowledges that it has finished, by using the action *ack*. The evolved component model $M'_2$ is created by adding the transition (*b*,*send*,*b*) into the component model $M_2$. It means that $M'_2$ allows multiple *send* actions to occur before producing output.

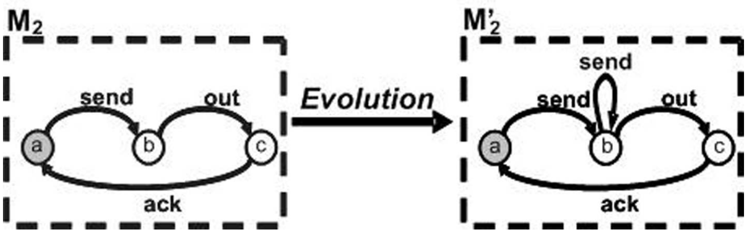



Figure 2.6: An illustration of the component evolution concept.

## 2.5 Summary

In this chapter, we briefly present the background of the dissertation. The first part of the chapter is about model and component specification, some related concepts for compositional verification, and the accurate model. The second part is about deterministic finite state automata which is used by the L* learning algorithm for generating assumptions. The third one briefly presents the assume-guarantee paradigm for modular verification of component-based software, the concepts of weakest assumption and minimal assumption, and an brief introduction to the LTSA tool supporting the compositional checking for the concurrent systems. Finally, we present the component evolution concept which is used in our work.

# Chapter 3

# A Minimized Assumption Generation Method for Component-Based Software Verification

This chapter proposes a method for generating minimal assumptions for the assume-guarantee verification of component-based software. The key idea of this method is finding the minimal assumptions in the search spaces of the candidate assumptions. These assumptions are seen as the environments needed for the components to satisfy a property and for the rest of the system to be satisfied. The minimal assumptions generated by the proposed method can be used to recheck the whole system at much lower computational cost. We have implemented a tool for generating the minimal assumptions. Experimental results are also presented and discussed.

## 3.1 Introduction

An assume-guarantee verification method proposed in [10] has been recognized as a promising approach to verify component-based software (CBS) with model checking. The method is not only fitted to component-based software but also has a potential to solve the *state space explosion problem* in model checking. This method allows us to decompose a verification target into components so that we can model check each of them separately. The key idea of this method is to generate assumptions as environments needed for components to satisfy a property. These assumptions are then discharged by the rest of the system. For example, consider a simple case where a CBS is made up of two components $M_1$ and $M_2$. The method proposed in [10] verifies that whether this system satisfies a property $p$ *without composing* $M_1$ with $M_2$. For this goal, an assumption $A(p)$ is generated by applying a learning algorithm called L* [9, 17] such that $A(p)$ is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$ (i.e., $\langle A(p)\rangle\ M_1\ \langle p\rangle$ and $\langle \text{true}\rangle\ M_2\ \langle A(p)\rangle$ which are called compositional rules, both hold). From these rules, this system satisfies $p$. In order to check these compositional rules, the number of states of the assumption $A(p)$ should be minimized because the computational cost of model checking of these rules is influenced by that number. This means that the cost of verification of CBS is reduced with a smaller assumption. Moreover, when a component is evolved after adapting some refinements in the context of the software evolution, the whole evolved CBS of many existing components and the evolved component is required to be rechecked [27, 28]. In this case, we also can reduce the cost of rechecking the evolved CBS by reusing the smaller assumption. These observations imply that the size of the generated assumptions is primary importance. However, the method proposed in [10, 16] focuses only on generating the assumptions which satisfies the compositional rules. The number of states of the generated assumptions is not mentioned in this work. Thus, the assumptions generated by the method are not minimal. A more detailed discussion of this issue can be found in Section 3.3.

This chapter proposes a method for generating the minimal assumptions for assume-guarantee verification of component-based software to deal with the above issue. The key idea of this method is finding the minimal assumption that satisfies the compositional rules thus is considered as a search problem in a search space of the candidate assumptions. These assumptions is seen as the environments needed for components to satisfy a property and for the rest of the CBS to be satisfied. With regard to the effectiveness, the proposed method can generate the minimal assumptions which have the minimal sizes and a smaller number of transitions than the assumptions generated by the method proposed in [10]. These minimal assumptions generated by the proposed method can be used to recheck the whole CBS by checking the compositional rules at much lower computational costs.

The rest of the chapter is organized as follows. We first describes the L* learning algorithm and the current method for assumption generation by using the L* learning algorithm in Section 3.2. Section 3.3 is about a minimized L*-based assumption generation method to find the minimal assumptions for component-based software verification. Section 3.4 presents some improvements for reducing the search space which is needed for the minimized assumption generation method to find the minimal assumptions. Section 3.5 shows the implemented tools and experimental results. We discuss the proposed method in Section 3.6. Section 3.7 presents related works. Finally, we summarize the chapter in Section 3.8.

## 3.2 L*-Based Assumption Generation Method

### 3.2.1 The L* Learning Algorithm

The proposed method uses the learning algorithm developed by Angluin [9] and later improved by Rivest and Schapire [17]. In this dissertation, we refer to the improved version by the name of the original algorithm called L*. L* learns an unknown regular language and produces a DFA that accepts it. The main idea of the L* learning algorithm is based on the "*Myhill-Nerode Theorem*" [29] in the theory of formal languages. It said that for every regular set $U \subseteq \Sigma^*$, there exists *a unique minimal deterministic automata* whose states are isomorphic to the set of equivalence classes of the following relation: $w \approx w'$ if and only if $\forall u \in \Sigma^*$: $wu \in U \iff w'u \in U$. Therefore, the main idea of L* is to learn the equivalence classes, i.e., two prefix aren't in the same class if and only if there is a distinguishing suffix u.

Let $U$ be an unknown regular language over some alphabet $\Sigma$. L* will produce a DFA $M$ such that $M$ is a minimal deterministic automata corresponding to $U$ and $L(M) = U$. In order to learn $U$, L* needs to interact with a *Minimally Adequate Teacher*, from now on called Teacher. The Teacher must be able to correctly answer two types of questions from L*. The first type is a membership query, consisting of a string $\sigma \in \Sigma^*$ (i.e., "is $\sigma \in U$?"); the answer is true if $\sigma \in U$, and false otherwise. The second type of these questions is a conjecture, i.e., a candidate DFA $M$ whose language the algorithm believes to be identical to $U$ ("is $L(M) = U$?"). The answer is true if $L(M) = U$. Otherwise the Teacher returns a counterexample, which is a string $\sigma$ in the symmetric difference of $L(M)$ and $U$. The interaction between L* Learning and the Teacher in a general view is illustrated in Figure 3.1.

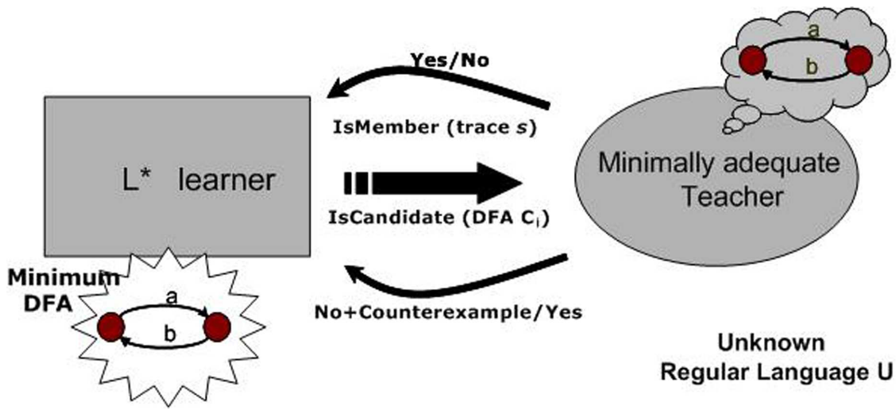



Figure 3.1: The interaction between L* Learner and the Teacher.

At a higher level, L* maintains a table $T$ that records whether string $s$ in $\Sigma^*$ belong to $U$. It does this by making membership queries to the Teacher to update the table. At various stages L* decides to make a conjecture. It uses the table $T$ to build a candidate DFA $M_i$ and asks the Teacher whether the conjecture is correct. If the Teacher replies true, the algorithm terminates. Otherwise, L* uses the counterexample returned by the Teacher to maintain the table with string $s$ that witness differences between $L(M_i)$ and $U$.

For more details, L* builds an observation table $(S, E, T)$, where:

- $S \in \Sigma^*$ is a set of prefixes. It presents equivalence classes or states.
- $E \in \Sigma^*$ is a set of suffixes. It presents the distinguishing.
- $T$: $(S \cup S.\Sigma).E \mapsto \{true, false\}$ where, the operator "." means that given two sets of event sequences $P$ and $Q$, $P.Q = \{pq \mid p \in P, q \in Q\}$, where $pq$ presents the concatenation of the event sequences $p$ and $q$. With a string $s$ in $\Sigma^*$, $T(s)$ = true means $s \in U$, otherwise $s \notin U$.

An observation table $(S, E, T)$ is closed if $\forall s \in S$, $\forall a \in \Sigma$, $\exists s' \in S$, $\forall e \in E$: $T(sae) = T(s'e)$. In this case, $s'$ presents the next state from $s$ after seeing $a$, $sa$ is undistinguishable form $s'$ by any of suffixes. Intuitively, the observation table $(S, E, T)$ is closed means that every row $sa$ of $S.\Sigma$ has a matching row $s'$ in $S$.

The detailed information of the L* algorithm step by step is presented in Algorithm 1, line numbers refer to L*'s illustration. Initially, L* sets $S$ and $E$ to $\{\lambda\}$ (line 1), where $\lambda$ presents the empty string. Subsequently, it updates the function $T$ by making membership queries so that it has a mapping for every string in $(S \cup S.\Sigma).E$ (line 3). It then checks whether the observation table $(S, E, T)$ is closed (line 4). If the observation table $(S, E, T)$ is not closed, then $sa$ is added to $S$, where $s \in S$ and $a \in \Sigma$ are the elements for which there is no $s' \in S$ (line 5). Because $sa$ has been added to $S$, $T$ must be updated again by making membership queries (line 6). Line 5 and line 6 are repeated until the table $(S, E, T)$ is closed.

When the observation table $(S, E, T)$ is closed, a candidate DFA $M = \langle Q, \alpha M, \delta, q_0, F\rangle$ is constructed (line 8) from the closed table $(S, E, T)$, where:

- $Q = S$.
- Alphabet $\alpha M = \Sigma$, where $\Sigma$ is the alphabet of the unknown language $U$.
- The transition $\delta$ is defined as $\delta(s, a) = s'$ where $\forall e \in E : T(sae) = T(s'e)$
- initial state $q_0 = \lambda$.
- $F = \{s \in S \mid T(s) = true\}$.

The candidate DFA $M$ is presented as a conjecture to the Teacher (line 9). If the Teacher replies true (i.e., $L(M) = U$) (line 10), L* returns $M$ as correct (line 11), otherwise it receives an counterexample $cex \in \Sigma^*$ from the Teacher.

The counterexample $cex$ is analyzed by L* to find a suffix $e$ of $cex$ that witnesses a difference between $L(M)$ and $U$. After that, $e$ must be added to $E$ (line 13). It will cause the next conjectured automaton to reflect this difference. When $e$ has been added to $E$, L* iterates the entire process by looping around to line 3.

**Algorithm 1** The L* learning algorithm.

**Input:** $U, \Sigma$: an unknown regular language $U$ over some alphabet $\Sigma$
**Output:** $M$: a DFA $M$ such that $M$ is a minimal deterministic automata corresponding to $U$ and $L(M) = U$

1: Initially, $S = \{\lambda\}, E = \{\lambda\}$
2: **loop**
3:    update $T$ using *membership queries*
4:    **while** $(S, E, T)$ is *not closed* **do**
5:       add $sa$ to $S$ to make $S$ closed, where $s \in S$ and $a \in \Sigma$
6:       update $T$ using *membership queries*
7:    **end while**
8:    construct a *candidate* DFA $M$ from the closed $(S, E, T)$
9:    present an *equivalence query*: $L(M) = U$?
10:    **if** $M$ is correct **then**
11:       **return** $M$
12:    **else**
13:       add $e \in \Sigma^*$ that witnesses the *counterexample cex* to $E$
14:    **end if**
15: **end loop**

For example, Figure 3.2 presents a closed observation table and its candidate DFA constructed from this table. It is very easy to check this table is closed. Intuitively, every row $sa$ of $S.\Sigma$ has a matching row $s'$ in $S$. In order to avoid misunderstanding in the figure, we modify state's name of the DFA, i.e, $\lambda$ changes into $q_0$, $a$ changes into $q_1$. From this closed table, L* constructs the candidate DFA $M$, where $\alpha M = \Sigma = \{a, b\}$, $Q = \{q_0, q_1\}$, the initial state is $q_0$, $\delta = \{(q_0, a, q_1), (q_0, b, q_1), (q_1, a, q_1), (q_1, b, q_0)\}$, and $F = \{q_1\}$. From the DFA $M$, we can get a safety LTS simply by removing non-accepting state $q_0$ and all its ingoing transitions.

Each candidate DFA $M_i$ produced by L* is smallest. It means that any DFA consistent with the observation table $(S, E, T)$ has at least as many states as $M_i$. Let $M_1, M_2, \ldots, M_n$ are candidate DFAs produced by L* step by step, it is very easy to check that $|M_1| \leq |M_2| \leq \ldots \leq |M_n|$, where $|M_i|$ denotes number of states of the DFA $M_i$. L* is guaranteed

| | $T$ | $\lambda$ | a | ab | |
|---|---|---|---|---|---|
| $S$ | $\lambda$ | false | true | false | $[q_0]$ |
| | a | true | true | false | $[q_1]$ |
| $S \cdot \Sigma$ | b | true | true | false | $[q_1]$ |
| | aa | true | true | false | $[q_1]$ |
| | ab | false | true | false | $[q_0]$ |

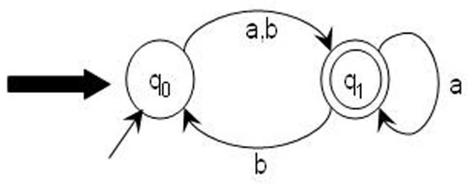



Figure 3.2: An illustration of a closed observation table $(S, E, T)$ and its candidate DFA.

to terminate with a minimal automaton $M$ for the unknown language $U$. Moreover, for each closed observation table $(S, E, T)$, the candidate DFA $M$ that L* constructs is smallest [29], in the sense that any other DFA consistent with the function $T$ has at least as many states as $M$. The conjectures made by L* strictly increase in size; each conjecture is smaller than the next one, and all incorrect conjectures are smaller than $M$. Therefore, if $M$ has n states, L* makes at most n-1 incorrect conjectures. The number of membership queries made by L* is $\mathcal{O}(kn^2 + n \log m)$ [9], where $k$ is the size of alphabet of $U$, $n$ is the number of states in the minimal DFA for $U$, and $m$ is the length of the longest counterexample returned when a conjecture is made.

### 3.2.2 L*-Based Assumption Generation

The assume-guarantee paradigm is a powerful *divide-and-conquer* mechanism for decomposing a verification process of a CBS into subtasks about the individual components. Consider a simple case where a system is made up of two components including a framework $M_1$ and an extension $M_2$. The extension $M_2$ is plugged into the framework $M_1$ via the parallel composition operator defined in Chapter 2 (i.e., synchronizing the common actions and interleaving the remaining actions). Figure 3.3 shows a general view of assume-guarantee verification. The goal is to verify that whether this system satisfies a property $p$ *without composing* $M_1$ with $M_2$. For this purpose, an assumption $A(p)$ is generated [10] by applying the L* learning algorithm such that $A(p)$ is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$ (i.e., $\langle A(p) \rangle\ M_1\ \langle p \rangle$ and $\langle \text{true} \rangle\ M_2\ \langle A(p) \rangle$ both hold). From these compositional rules, this system satisfies $p$. Unfortunately, it is often difficult to find such an assumption. Formally, given two component models and a required property represented by LTSs $M_1$ , $M_2$ and $p$, the main goal in this problem is to find a LTS $A(p)$ such that $L(A(p)\|M_1)\uparrow\alpha p \subseteq L(p)$ and $L(M_2)\uparrow\alpha A(p) \subseteq L(A(p))$.

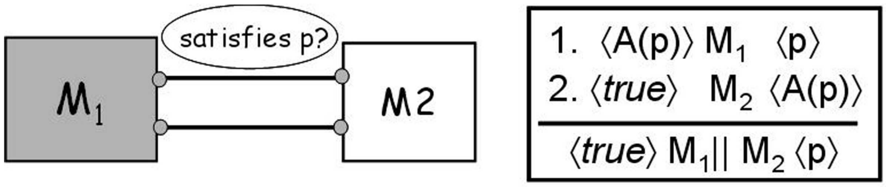



Figure 3.3: A general view of assume-guarantee verification.

Recently, there are two proposed methods for generating such assumptions automatically. The first one is an algorithmic, non-incremental, generation of assumptions proposed in [16]. It finds the weakest assumption $A_W$ by taking the complement of paths in the product automata leading to error states. The weakest assumption $A_W$ describes exactly those traces over $\Sigma = (\alpha M_1 \cup \alpha p) \cap \alpha M_2$ which do not lead to the error state $\pi$ in $M_1\|p_{err}$. For all component model $M_2'$, $M_1\|M_2' \models p$ if and only if $M_2' \models A_W$. The drawback in this method is that if the computation runs out of memory (i.e., if the state space of the component model is too large), then no assumption will be obtained as a result. The advantage of this method is that it does not require knowledge of the environment. We would like to find an assumption $A(p)$ that is stronger than $A_W$ because $A_W$ is the weakest assumption. This is major goal of the second method proposed in [10] about assumption generation using the L* learning algorithm. It is an incremental method, based on counterexamples and learning. Instead of finding $A_W$, the method uses the L* learning algorithms to learn $A_W$. The advantage of this method is an any time method, which means that it produces a finite sequence of approximations to an assumption that can be used to obtain conclusive results in assume-guarantee reasoning. If it runs out of memory, intermediate assumptions can still be useful. However, this method requires knowledge of the environment and is quite difficult to understand.

We explain details of the second proposed method as follows. In order to obtain appropriate assumptions, the method applies the compositional rules in an iterative fashion illustrated in Figure 3.5. At each iteration $i$, a candidate assumption $A_i$ is produced based on some knowledge about the system and on the results of the previous iteration. The two steps of the compositional rules are then applied. Step 1 checks whether $M_1$ satisfies $p$ in environments that guarantee $A_i$ by computing the formula $\langle A_i \rangle\ M_1\ \langle p \rangle$. If the result is false, it means that this candidate assumption is *too weak* (i.e., $A_i$ does not restrict the environment enough for $p$ to be satisfied). Thus, the candidate assumption $A_i$ must be strengthened, which corresponds to removing behaviors from it, with the help of the counterexample *cex* produced by this step. In the context of the next candidate assumption $A_{i+1}$, component $M_1$ should at least not exhibit the violating behavior reflected by this counterexample. Otherwise, the result is true, it means that $A_i$ is strong enough for $M_1$ to satisfy the property $p$. The step 2 is then applied to check that if component model $M_2$ satisfies $A_i$ by computing the formula $\langle \text{true} \rangle$ $M_2$ $\langle A_i \rangle$. If this step returns true, the property $p$ holds in the compositional system $M_1 \| M_2$ (i.e., the system $M_1 \| M_2 \models p$ is verified) and the algorithm terminates. Otherwise, this step returns false, further analysis is required to identify whether $p$ is indeed violated in $M_1 \| M_2$ or the candidate assumption $A_i$ is too strong to be satisfied by $M_2$. Such analysis is based on the counterexample $cex$ returned by this step. It must checks that whether the counterexample $cex$ belong to the unknown language $U = L(A_W)$ (i.e., whether $cex \in L(A_W)$?). For this purpose, this analysis checks whether $p$ is violated by $M_1$ in the context of the counterexample $cex$ by checking the formula $[cex] \| M_1 \not\models p$, where $[cex]$ is a LTS defined as follows:

Let LTS $[cex] = \langle Q, \alpha[cex], \delta, q^0 \rangle$ and the counterexample $cex = a_1 a_2 \ldots a_k$. The LTS $[cex]$ is created from the counterexample $cex$ as follows:

- $Q = \{q_0, q_1, \ldots, q_k\}$,
- $\alpha[cex] = \{a_1, a_2, \ldots, a_k\}$,
- $\delta = \{(q_{i-1}, a_i, q_i) \mid 1 \leq i \leq k\}$, and
- $q^0 = q_0$.

Figure 3.4 illustrates the LTS $[cex]$ is created from the counterexample $cex$.

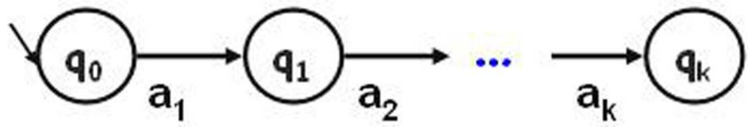



Figure 3.4: The LTS $[cex]$ is created from the counterexample $cex$.

If the property $p$ does not hold in the compositional system $[cex] \| M_1$ (i.e., $[cex] \| M_1 \not\models p$), it means that the compositional system $M_1 \| M_2$ does not satisfy the property $p$ (i.e., $M_1 \| M_2 \not\models p$). Otherwise, $A_i$ is too strong to be satisfied by $M_2$. The candidate assumption $A_i$ therefore must be weakened (i.e., behaviors must be added) in the iteration $i+1$. The result of such weakening will be that at least the behavior that the counterexample $cex$ represents will be allowed by candidate assumption $A_{i+1}$. New candidate assumption may of course be too weak, and therefore the entire process must be repeated.

An important question in this method is that how the module L* Learning works. The same question is that how to generate a candidate assumption $A_i$ at each iteration

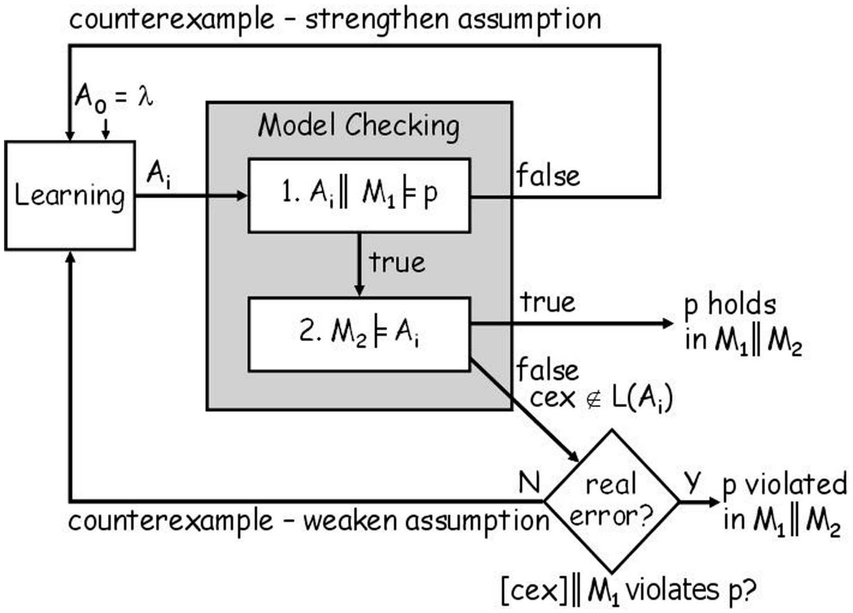



Figure 3.5: The L*-based assumption generation framework.

$i$ in the framework illustrated in Figure 3.5. In the assume-guarantee method proposed in [10], L* learns the language of the weakest assumption $A_W$. This means that L* learns the unknown language $U = L(A_W)$ over the alphabet $\Sigma = \alpha A_W = (\alpha M_1 \cup \alpha p) \cap \alpha M_2$. The method uses candidates produced by L* learning as candidate assumptions $A_i$ for the assume-guarantee rules (compositional rules). In order to produce each candidate assumption $A_i$, L* first produces a candidate DFA $M_i$ based on the closed observation table $S, E, T$, it then translates the candidate DFA $M_i$ into a safety LTS as candidate assumptions $A_i$ by applying the definition presented in Chapter 2. In order to learn $A_W$, we need to provide a Teacher that is able to answer the two different kinds of questions that L* asks. The first type is a membership query, consisting of a string $\sigma \in \Sigma^*$; the answer is true if $\sigma \in U$, and false otherwise. The second type of question is a conjecture, i.e., a candidate DFA $M_i$ whose language the algorithm believes to be identical to $U$. The answer is true if $L(M_i) = U$. Otherwise the Teacher returns a counterexample, which is a string $\sigma$ in the symmetric difference of $L(M_i)$ and $U$. This approach uses model checking to implement such a Teacher.

For the first type of questions, in order to answer a membership query for string $\sigma = a_1a_2 \ldots a_n$ whether in $\Sigma^* = L(A_W)$, the Teacher simulates the query on the composition $M_1 \| p_{err}$. For the string $\sigma$, the Teacher first builds a safety LTS $[\sigma] = \langle Q, \alpha[\sigma], \delta, q^0 \rangle$, where $Q = \{q_0, q_1, \ldots, q_n\}$, $\alpha[\sigma] = \Sigma$, $\delta = \{(q_{i-1}, a_i, q_i) \mid 1 \leq i \leq n\}$, and $q^0 = q_0$. The Teacher then checks the formula $\langle [\sigma] \rangle\ M_1\ \langle p \rangle$ by computing the compositional system $[\sigma] \| M_1 \| p_{err}$. If the state error $\pi$ is unreachable in this compositional system (the formula returns *true*), it means that $\sigma \in L(A_W)$. In this case, the Teacher returns *true* because $M_1$ does not violate the property $p$ in the context of $\sigma$. Otherwise, the answer to the membership query is $false$.

For the second type of questions, with each DFA $M_i$ produced by L* from the observation table $S, E, T$ at each iteration $i$, the Teacher must checks that whether the DFA $M_i$ is a candidate DFA for the iteration $i$ (i.e., whether $L(M_i) = L(A_W)$?) For this purpose, the Teacher first translates the DFA $M_i$ into a safety LTS $A_i$. It then uses the safety LTS $A_i$ as candidate assumption for the compositional rules. The Teacher applies two steps of the compositional rules and the counterexample analysis to answer conjectures as follows:

- Step 1 illustrated in Figure 3.5 first is applied, the Teacher checks the formula $\langle A_i \rangle$ $M_1$ $\langle p \rangle$ by computing the compositional system $A_i \| M_1 \| p_{err}$. If the state error $\pi$ is reachable in this composition system, it means that this formula does not hold. The Teacher then returns false and a counterexample *cex*. The Teacher informs L* that its conjecture $A_i$ is not correct and provides $cex{\uparrow}\Sigma$ to witness this fact. Otherwise, this formula holds, the Teacher forwards $A_i$ to Step 2.

- Step 2 is applied by checking the formula $\langle \text{true} \rangle$ $M_2$ $\langle A_i \rangle$ illustrated in Figure 3.5. If this formula holds, the Teacher returns true. Our framework then terminates the verification because, according to the compositional rule, the property $p$ has been proved on the compositional system $M_1 \| M_2$. Otherwise, this step returns a counterexample *cex*. The Teacher then performs some analysis to determine whether $p$ is indeed violated in $M_1 \| M_2$ or the candidate assumption $A_i$ is too strong to be satisfied by $M_2$.

- Counterexample analysis is performed by the Teacher in a way similar to that used for answering membership queries. Let *cex* be the counterexample returned by the Step 2. The Teacher first creates a safety LTS $[cex \uparrow \Sigma]$ from the counterexample *cex* illustrated in Figure 3.4. The Teacher then checks the formula $\langle [cex \uparrow \Sigma] \rangle$ $M_1$ $\langle p \rangle$ by computing the compositional system $[cex \uparrow \Sigma] \| M_1 \| p_{err}$. If the state error $\pi$ is unreachable, then the compositional system $M_1 \| M_2$ does not satisfy the property $p$ (i.e., $M_1 \| M_2 \not\models p$). Otherwise, $A_i$ is too strong for $M_2$ to satisfy in the context of *cex*. The $cex{\uparrow}\Sigma$ is returned as a counterexample for conjecture $A_i$.

### 3.2.3 An Example

An illustrative CBS which contains of the framework $M_1$ and the extension $M_2$ presented in Figure 3.6. In the CBS, the LTS of the framework $M_1$ as the *Input* LTS, and the LTS of $M_2$ as the *Output* LTS. The initial state of the *Input* LTS in this example is the state

0. The initial state of the *Output* LTS is the state $a$. The extension $M_2$ is plugged into the framework $M_1$ via the parallel composition operator (i.e., synchronizing the common actions and interleaving the remaining actions). This system means that the *Input* LTS receives an input when the action *in* occurs, and then sends it to the *Output* LTS with action *send*. After some data is sent to it, the *Output* LTS produces output using the action *out* and acknowledges that it has finished, by using the action *ack*. At this point, both LTSs return to their initial states so the process can be repeated. The required property $p$ means that the *in* action has to occur before the *out* action.

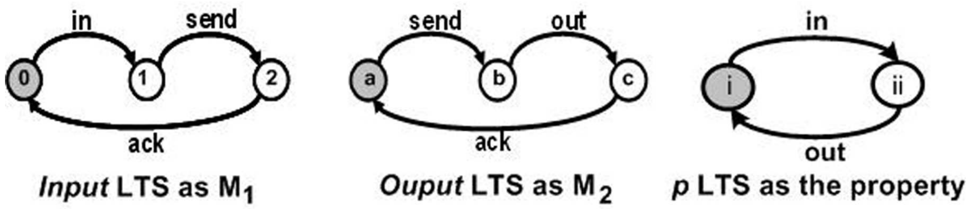



Figure 3.6: Components and order property of the illustration system.

In order to generate an assumption $A(p)$ between the framework $M_1$ and the extension $M_2$ that is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$, L* learns the weakest assumption $A_W$. This means that L* learns the unknown language $U = L(A_W)$ over the alphabet $\Sigma = \alpha A_W = (\alpha M_1 \cup \alpha p) \cap \alpha M_2 = \{send, out, ack\}$.

Initially, L* sets the observation table $S, E, T$ to the empty observation table illustrated in Figure 3.7 by setting $S$ and $E$ to $\{\lambda\}$, where $\lambda$ presents the empty string. The observation table $S, E, T$ is updated by making membership queries to the Teacher, i.e., $\lambda \in L(A_W)$?, $ack \in L(A_W)$?, $out \in L(A_W)$?, and $send \in L(A_W)$?.

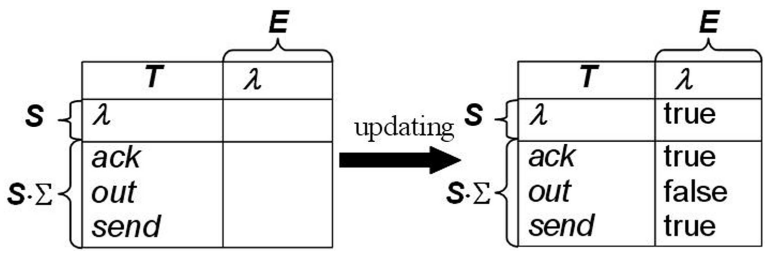



Figure 3.7: The empty observation table and its updated table.

The updated table presented in Figure 3.7 is not closed because the row *out* in $S.\Sigma$ has no matching row in $S$. In order to make this table to be closed, *out* is added to $S$. The observation table $S, E, T$ after adding *out* to $S$ illustrated in Figure 3.8. The observation table is updated again by making membership queries to the Teacher, i.e., $out\ ack \in L(A_W)$?, $out\ out \in L(A_W)$?, and $out\ send \in L(A_W)$?.

| | T | λ |
|---|---|---|
| S | λ | true |
| | out | |
| S·Σ | ack | true |
| | out | false |
| | send | true |
| | out ack | |
| | out out | |
| | out send | |

updating →

| | T | λ |
|---|---|---|
| S | λ | true |
| | out | false |
| S·Σ | ack | true |
| | out | false |
| | send | true |
| | out ack | false |
| | out out | false |
| | out send | false |


Figure 3.8: The table after adding *out* to $S$, its updated table, and the candidate assumption $A_1$.

The Teacher uses the safety LTS $A_1$ as a candidate assumption for the compositional rules. The Teacher applies two steps of the compositional rules and counterexample analysis to answer conjectures from L* Learner.

The step 1 first is applied to check the formula $\langle A_1 \rangle$ $Input$ $\langle p \rangle$ by computing the compositional system $A_1 \| Input \| p_{err}$. It is easy to check that the error state $\pi$ is reachable in this compositional system, so the Teacher then returns $false$ and a counterexample *cex* = *in send ack in*. The Teacher informs L* Learner that its conjecture $A_1$ is not correct and provides $cex{\uparrow}\Sigma$ = *send ack* to witness this fact.

The counterexample $cex{\uparrow}\Sigma$ = *send ack* is analyzed by L* to find a suffix $e$ of *cex* that witnesses a difference between $L(A_1)$ and $U = L(A_W)$. In this case, L* analyzes and sets $e$ to *ack*. In order to generate a next candidate assumption, the closed table $S, E, T$ presented in Figure 3.8 is updated by adding the suffix $e$ = *ack* to $E$. The observation table $S, E, T$ after adding *ack* to $E$ illustrated in Figure 3.9. This table continuously is updated by making membership queries to the Teacher, i.e., *ack* $\in L(A_W)$?, *out ack* $\in L(A_W)$?, *ack ack* $\in L(A_W)$?, *out ack* $\in L(A_W)$?, *send ack* $\in L(A_W)$?, *out ack ack* $\in L(A_W)$?, *out out ack* $\in L(A_W)$?, and *out send ack* $\in L(A_W)$?.

The updated table $S, E, T$ presented in Figure 3.9 is not closed because the row *send* in $S.\Sigma$ has no matching row in $S$. In order to make this table to be closed, *send* is added to $S$. The observation table $S, E, T$ after adding *send* to $S$ illustrated in Figure 3.10. This table is updated again by making membership queries to the Teacher.

The updated table $S, E, T$ presented in Figure 3.10 is closed. A candidate DFA $A_2$ is constructed from this closed observation table shown in Figure 3.10. The Teacher then uses the safety LTS $A_2$ as a candidate assumption for the compositional rules. The Teacher applies two steps of the compositional rules and counterexample analysis to answer conjectures from L* Learner.

| | $T$ | $\lambda$ | *ack* |
|---|---|---|---|
| $S$ | $\lambda$ | true | |
| | *out* | false | |
| $S{\cdot}\Sigma$ | *ack* | true | |
| | *out* | false | |
| | *send* | true | |
| | *out ack* | false | |
| | *out out* | false | |
| | *out send* | false | |

updating $\longrightarrow$

| | $T$ | $\lambda$ | *ack* |
|---|---|---|---|
| $S$ | $\lambda$ | true | true |
| | *out* | false | false |
| $S{\cdot}\Sigma$ | *ack* | true | true |
| | *out* | false | false |
| | *send* | true | false |
| | *out ack* | false | false |
| | *out out* | false | false |
| | *out send* | false | false |

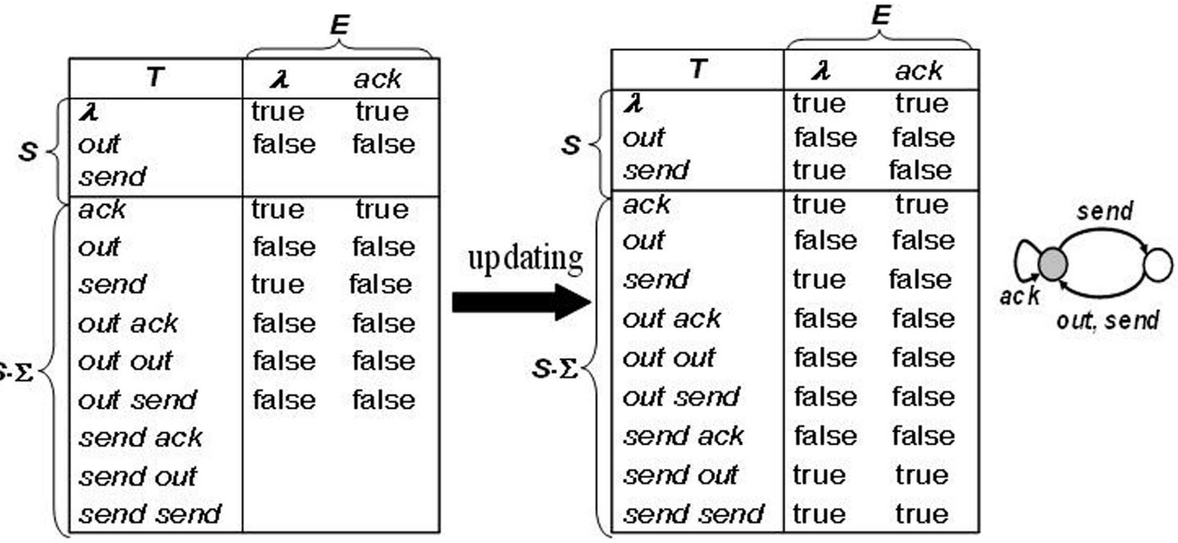
Figure 3.9: The observation table after adding *ack* to $S$ and its updated table.

The step 1 first is applied to check the formula $\langle A_2 \rangle$ *Input* $\langle p \rangle$ by computing the compositional system $A_2 \| Input \| p_{err}$. It is easy to check that the error state $\pi$ is unreachable in this compositional system, so the Teacher then returns *true*. This means that the formula $\langle A_2 \rangle$ *Input* $\langle p \rangle$ holds. The Teacher forwards $A_2$ to the step 2.

The step 2 is applied by checking the formula $\langle true \rangle$ *Output* $\langle A_2 \rangle$. In order to check this formula, the Teacher computing the compositional system $Output \| A_{2_{err}}$. The error state $\pi$ is unreachable in this compositional system, so the Teacher then returns *true*. This means that the required property $p$ holds in the CBS $Input \| Output$ (i.e., $Input \| Output \models p$). The L* learning algorithms terminates and returns the assumption $A(p) = A_2$.

## 3.3 Minimized Assumption Generation Method

The assume-generation verification proposed in [10] is a powerful method for checking component-based software by decomposing a verification target about a component-based software into parts about the individual components. In this method, assumptions which are seem as environments of the components are generated. The number of states of the assumptions should be minimized because this number influences on the computational cost of model checking. However, the assumptions generated by this method are not minimal. Figure 3.11 is a counterexample to prove this fact. In this counterexample, given two component models $M_1$ (Input), $M_2$ (Output), and a required property $p$, the method proposed in [10] generates the assumption $A(p)$. However, there is a smaller assumption with a smaller size and a smaller number of transitions. The reason why this method does not generate a minimal assumption is presented as follows. The L* used in this method learns the language of the weakest assumption $A_W$ over the alphabet $\Sigma =$



| T | $\lambda$ | ack |
|---|---|---|
| $\lambda$ | true | true |
| out | false | false |
| send | | |
| ack | true | true |
| out | false | false |
| send | true | false |
| out ack | false | false |
| out out | false | false |
| out send | false | false |
| send ack | | |
| send out | | |
| send send | | |

| T | $\lambda$ | ack |
|---|---|---|
| $\lambda$ | true | true |
| out | false | false |
| send | true | false |
| ack | true | true |
| out | false | false |
| send | true | false |
| out ack | false | false |
| out out | false | false |
| out send | false | false |
| send ack | false | false |
| send out | true | true |
| send send | true | true |



Figure 3.10: The table after adding *send* to $S$, its updated table, and the candidate assumption $A_2$.

$(\alpha M_1 \cup \alpha p) \cap \alpha M_2$ and produces a DFA that accepts it. In order to learn this language, L* builds an observation table $(S, E, T)$ where $S$ and $E$ are a set of prefixes and suffixes respectively, both over $\Sigma^*$. $T$ is a function which maps $(S \cup S.\Sigma).E$ to $\{true, false\}$, where the operator "." is defined as follows. Given two sets of event sequences $P$ and $Q$, $P.Q = \{pq \mid p \in P, q \in Q\}$, where $pq$ presents the concatenation of the event sequences $p$ and $q$. The technique for answering membership queries used in this method means that for any string $s \in (S \cup S.\Sigma).E$, $T(s) = true$ if $s \in L(A_W)$, and $false$ otherwise. In the counterexample showed in Figure 3.11, if $s \in L(A_W)$ but $s \notin L(A(p))$, then $T(s)$ is set to $true$ (in this case, $T(s)$ should to be $false$). For this reason, the assumption $A(p)$ generated by this method contains some strings/traces which do not belong to the language of the assumption be learned.

This section proposes a method for generating minimal assumptions for assume-guarantee verification of component-based software. We also define a new technique for answering membership queries to dealing with the above issue. The minimal assumption is generated by combining the L* learning algorithm and the breadth-first search strategy. We ensure that the assumptions generated by this method are minimal (see Theorem 2).

### 3.3.1 An Improved Technique for Answering Membership Queries

As mentioned above, in order to learn the language of the assumption, the L* learning algorithm used in [10] builds an observation table $(S, E, T)$ where $T$ is a function which maps $(S \cup S.\Sigma).E$ to $\{true, false\}$. For any string $s \in (S \cup S.\Sigma).E$, $T(s) = true$ if $s \in L(A_W)$, and $false$ otherwise. In the case where $s \in L(A_W)$, we can not ensure that

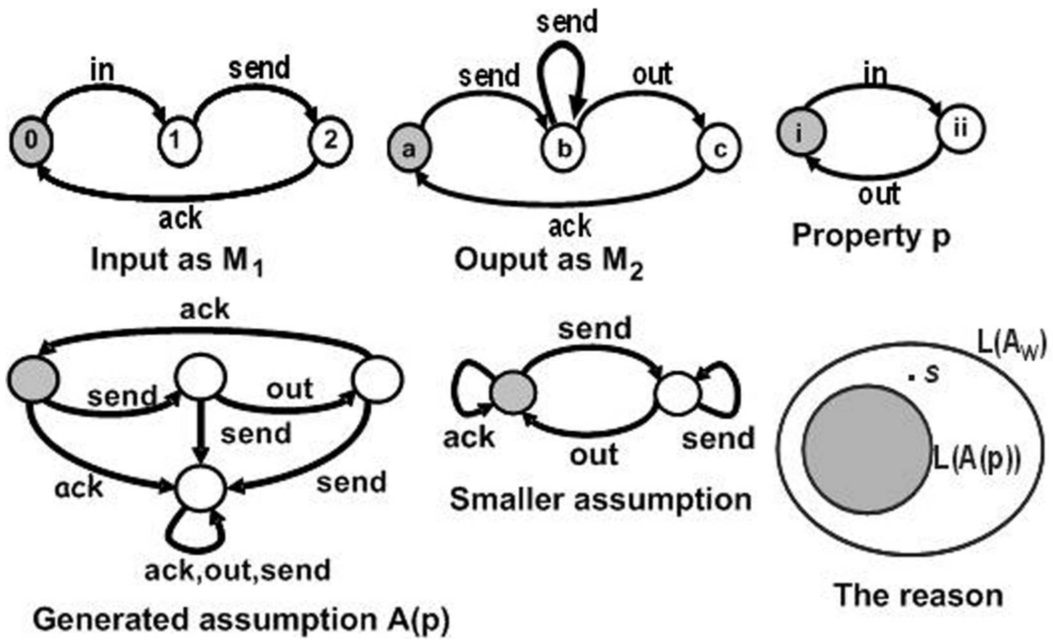



Figure 3.11: A counterexample and the reason to show that the assumptions generated in [10] are not minimal.

whether $s$ belongs to the language to be learned or not (i.e., whether $s \in L(A(p))$?). If $s \notin L(A(p))$ then $T(s)$ should be $false$. However, the work in [10] sets $T(s)$ to $true$ in this case. For this reason, the generated assumptions are not minimal in this work. In order to solve this issue, we use a new value called "?" to represent the value of $T(s)$ in such cases. We define an improved technique for answering membership queries as follows. To generate a minimal assumption, the L* learning algorithm used in our work builds an observation table $(S, E, T)$, where $S$ and $E$ are a set of prefixes and suffixes respectively, both over $\Sigma^*$. $T$ is a function which maps $(S \cup S.\Sigma).E$ to $\{true, false,$ "?"$\}$, where "?" can be seen as a "don't know" value. The "don't know" value means that for each string $s \in (S \cup S.\Sigma).E$, even if $s \in L(A_W)$, we do not know whether $s$ belongs to the language of the assumption be learned or not. The technique for answering membership queries used in our method means that for any string $s \in (S \cup S.\Sigma).E$, if $s$ is the empty string then $T(s) = true$, else $T(s) = false$ if $s \notin L(A_W)$, and "?" otherwise.

### 3.3.2 Algorithm for Minimal Assumption Generation

Finding an assumption where it has a minimal size that satisfies the compositional rules thus is considered as a search problem in a search space of observation tables. We use the breadth-first search strategy because this strategy ensures that the generated assumption is minimal (Theorem 2). In the following more detailed presentation of the proposed algorithm for generating the minimal assumption, line numbers refer to the algorithm's illustration presented in Algorithm 2. In this algorithm, we use a queue data structure which contains the generated observation tables with the *first-in first-out* order. These observation tables are used for generating the candidate assumptions. Initially, the algorithm sets the queue $q$ to the empty queue (line 1). We then put the initial observation table $OT_0 = (S_0, E_0, T_0)$ into the queue $q$ as the root of the search space of observation tables, where $S_0 = E_0 = \{\lambda\}$ ($\lambda$ represents the empty string) (line 2). Subsequently, the algorithm gets a table $OT_i$ from the top of the queue $q$ (line 4). If $OT_i$ contains the "don't know" value "?" (line 5), we obtain all instances of $OT_i$ by replacing all "?" entries in $OT_i$ with both *true* and $false$ (line 6). For example, the initial observation table of the illustrative system presented in Figure 3.11 and one of its instance obtained by replacing all "?" entries with *true* value are showed in Figure 3.12. The obtained instances then are put into the queue $q$ (line 7). Otherwise, the table $OT_i$ does not contain the "?" value (line 9). In this case, if $OT_i$ is not closed (line 10), an updated table $OT$ is obtained by calling the procedure named *make_closed*($OT_i$) (line 11). $OT$ then is put into $q$ (line 12). In the case where the table $OT_i$ is closed (line 13), a candidate assumption $A_i$ is generated from $OT_i$ (line 14). The candidate assumption $A_i$ is used to check that whether it satisfies the two steps of the compositional rules. The step 1 is applied by calling the procedure named $Step1(A_i)$ to check that whether $M_1$ satisfies $p$ in environments that guarantee $A_i$ by computing the formula $\langle A_i \rangle\ M_1\ \langle p \rangle$. If $Step1(A_i)$ fails with a counterexample $cex$ (line 15), $A_i$ is *too weak* for $M_1$ to satisfy $p$. Thus, the candidate assumption $A_i$ must be strengthened by adding a suffix $e$ of $cex$ that witnesses a difference between $L(A_i)$ and and the language of the assumption be learned to $E_i$ of the table $OT_i$ (line 16). After that, an updated table $OT$ is obtained by calling the procedure named *update*($OT_i$) (line 17). $OT$ then is put into $q$ (line 18). Otherwise, $Step1(A_i)$ return *true* (line 19). This means that $A_i$ is strong enough for $M_1$ to satisfy the property $p$. The step 2 is then applied by calling the procedure named $Step2(A_i)$ to check that if $M_2$ satisfies $A_i$ by computing the formula $\langle \text{true} \rangle\ M_2\ \langle A_i \rangle$. If $Step2(A_i)$ fails with a counterexample $cex$ (line 20), further analysis is required to identify whether $p$ is indeed violated in $M_1 \| M_2$ or $A_i$ is too strong to be satisfied by $M_2$. Such analysis is based on the counterexample $cex$. If $cex$ witnesses the violation of $p$ in the system $M_1 \| M_2$ (line 21), the algorithm terminates and returns $cex$ (line 22). Otherwise, $A_i$ is too strong to be satisfied by $M_2$ (line 23). The candidate assumption $A_i$ therefore must be weakened by adding a suffix $e$ of $cex$ to $E_i$ of the table $OT_i$ (line 24). After that, an updated table $OT$ is obtained by calling the procedure named *update*($OT_i$) (line 25). $OT$ then is put into $q$ (line 26). Otherwise, $Step2(A_i)$ return *true* (line 28). This means that the property $p$ holds in the compositional system $M_1 \| M_2$. The algorithm terminates and returns $A_i$ as the minimal assumption (line 29). The algorithm iterates the entire process by looping from line 3 to line 34 until the queue $q$ is empty or a minimal assumption is generated.

**Algorithm 2** Minimized assumption generation.

**Input:** $M_1, M_2, p$: two component models $M_1$ and $M_2$, and a required property $p$
**Output:** $A_m(p)$ or $cex$: an assumption $A_m(p)$ with a smallest size if $M_1 \| M_2$ satisfies $p$, and a counterexample $cex$ otherwise

Initially, $q = empty$ $\{q$ is an empty queue$\}$
$q.put(OT_0)$ $\{OT_0 = (S_0, E_0, T_0), S_0 = E_0 = \{\lambda\}$, where $\lambda$ is the empty string$\}$
**while** $q \neq empty$ **do**
  $OT_i = q.get()$ {getting $OT_i$ from the top of $q$}
  **if** $OT_i$ contains "?" value **then**
    **for** each instance $OT$ of $OT_i$ **do**
      $q.put(OT)$ {putting $OT$ into $q$}
    **end for**
  **else**
    **if** $OT_i$ is not closed **then**
      $OT = make\_closed(OT_i)$
      $q.put(OT)$
    **else**
      construct a candidate DFA $A_i$ from the closed $OT_i$
      **if** $Step1(A_i)$ fails with $cex$ **then**
        add the suffix $e$ of the counterexample $cex$ to $E_i$
        $OT = update(OT_i)$
        $q.put(OT)$
      **else**
        **if** $Step2(A_i)$ fails with $cex$ **then**
          **if** $cex$ witnesses violation of $p$ **then**
            **return** $cex$
          **else**
            add the suffix $e$ of the counterexample $cex$ to $E_i$
            $OT = update(OT_i)$
            $q.put(OT)$
          **end if**
        **else**
          **return** $A_i$
        **end if**
      **end if**
    **end if**
  **end if**
**end while**

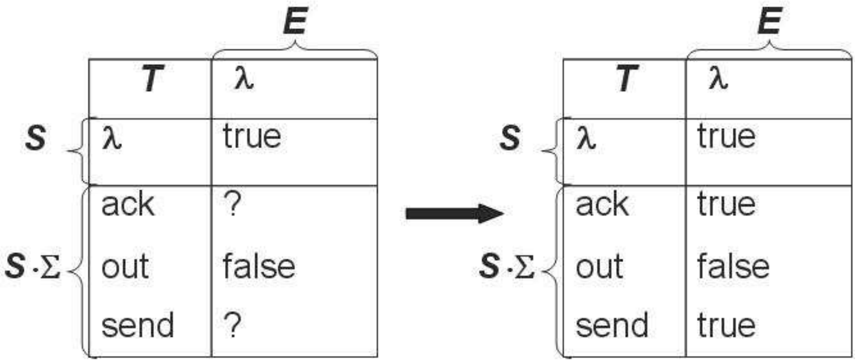



Figure 3.12: The initial observation table and one of its instances.

### 3.3.3 Characteristics of the Search Space

The search space of observation tables used in the proposed method exactly contains the generated observation tables which are used to generate the candidate assumptions. This search space is seen as a search tree where its root is the initial observation table $OT_0$. We can conveniently define the size of an observation table $OT = (S, E, T)$ as $|S|$, denoted $|OT|$. We use $A_{ij}$ to denote the $j$th candidate assumption generated from the $j$th observation table (denoted $OT_{ij}$) at the depth $i$ of the search tree. From the way to build the search tree presented in Algorithm 2, we have a theorem as follows.

**Theorem 1** *Let $A_{ij}$ and $A_{kl}$ be two candidate assumptions generated at the depth $i$ and $k$ respectively. $|A_{ij}| < |A_{kl}|$ implies that $i < k$.*

**Proof** The observation tables at the depth *i+1* are generated from the observation tables at the depth $i$ exactly in one of the following cases:

1. There is at least a table $OT_{ij}$ of the tables at the depth $i$ which contains the "?" value. In this case, the instances of this table are the tables at the depth *i+1*. These tables have the same size with the table $OT_{ij}$.

2. There is at least a table $OT_{ij}$ of the tables at the depth $i$ which is not closed. An updated table $OT_{(i+1)k}$ at the depth *i+1* is obtained from this table by adding a new element to $S_{ij}$. This mean that $|OT_{ij}| < |OT_{(i+1)k}|$.

3. Finally, there is at least a table $OT_{ij}$ of the tables at the depth $i$ which is not an actual assumption. In this case, an updated table $OT_{(i+1)k}$ at the depth *i+1* is obtained from this table by adding a suffix $e$ of the given counterexample $cex$ to $E_{ij}$. This mean that $|OT_{ij}| = |OT_{(i+1)k}|$.

These facts imply that if the size of the candidate generated from a table at the depth $i$ less than the size of the candidate generated from a table at the depth $k$, then $i < k$. ∎

### 3.3.4 Termination and Correctness

The termination and correctness of the proposed algorithm for the minimized assumption generation showed in Algorithm 2 are proved by the following theorem.

**Theorem 2** *Given two component models $M_1$ and $M_2$, and a property $p$, the proposed algorithm for the minimized assumption generation presented in Algorithm 2 terminates and returns true and an assumption $A_m(p)$ with a minimal size such that it is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$, if the compositional system $M_1 \| M_2$ satisfies $p$, and $false$ otherwise.*

**Proof** At any iteration $i$, the proposed algorithm returns an actual assumption $A_m(p) = A_i$ or a counterexample *cex* (i.e., $M_1 \| M_2 \not\models p$) and terminates or continues by providing a counterexample or continues to update the current observation table (if this table contains "?" or it is not closed). Because the proposed algorithm is based on the L* learning algorithm, by the correctness of L* [9, 17], we ensure that if the L* learning algorithm keeps receiving counterexamples, in the worst case, the algorithm will eventually produce the weakest assumption $A_W$ and terminates, by the definition of $A_W$ [16]. This means that the search space exactly contains the observation table $OT_W$ which is used to generate $A_W$. In the worst case, the proposed algorithm reaches to $OT_W$ and terminates.

With regard to the correctness, the proposed algorithm uses two steps of the compositional rules (i.e., $\langle A_i \rangle\ M_1\ \langle p \rangle$ and $\langle \text{true} \rangle\ M_2\ \langle A_i \rangle$) to answer the question of whether the candidate assumption $A_i$ produced by the algorithm is an actual assumption or not. It only returns *true* and a minimal assumption $A_m(p) = A_i$ when both steps return *true*, and therefore its correctness is guaranteed by the compositional rules. The proposed algorithm returns a real error (a counterexample *cex*) when it detects a trace $\sigma$ of $M_2$ which violates the property $p$ when simulated on $M_1$. In this case, it implies that $M_1 \| M_2'$ violates $p$. The remaining problem is to prove that the assumption $A_m(p)$ generated by the proposed algorithm is minimal. Suppose that there is exists an assumption $A$ such that $|A| < |A_m(p)|$. By using Theorem 1 for this fact, we can imply that the depth of the table used to generated $A$ less than the depth of the table used to generated $A_m(p)$. This means that the table used to generated $A$ has been visited by our algorithm. In this case, the algorithm generated $A$ as a candidate assumption and $A$ was not an actual assumption. These facts imply that such assumption $A$ does not exist. ∎

## 3.4 Reducing the Search Space

In the algorithm for minimal assumption generation shown in Algorithm 2, the queue has to hold a exponentially growing the number of the observation tables. This makes our method unpractical for the large-scale systems because it consumed too much memory. For the large-scale systems, the computational cost for generating the minimal assumption is very expensive. This section presents three solutions to deal with this issue. The first solution is to apply the depth-first search strategy in the search space of the observation tables to obtain the assumptions. In the second one, we apply the iterative deepening depth-first search strategy to combines depth-first search's space-efficiency and breadth-first search's completeness. Finally, we reuse the previous results (previous observation tables) to reduce the search space of the observation tables.

### 3.4.1 Depth-First Search

In order to reduce the memory cost for generating the minimal assumption of the proposed algorithm, we replace the breadth-first search with the depth-first search (DFS) as an improved algorithm for generating an assumption which satisfies the compositional rules presented in Algorithm 3. In this algorithm, we use a stack data structure which contains the generated observation tables with the *last-in first-out* order.

### 3.4.2 Iterative Deepening Depth-First Search

Although the memory complexity of depth-first search is much lower than breadth-first search, the time complexities of both strategies are the same. This means that depth-first search can not reduce the computational cost for generating assumptions of the proposed algorithm. An idea to reduce the computational cost for generating assumptions is to use the iterative-deepening depth first search (IDDFS). IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness. It is a state space search strategy in which a depth-limited search (DLS) is run repeatedly, increasing the depth limit with each iteration until it reaches the actual assumption or a counterexample to show that the CBS violates the property, the depth of the shallowest goal state. On each iteration, IDDFS visits the observation tables in the search tree in the same order as depth-first search, but the cumulative order in which observation tables are first visited, assuming no pruning, is effectively breadth-first. However, this strategy still has the same time complexity as breadth-first search. Only the path from the root of the search tree (i.e., the initial observation table $OT_0$) to the current instance has to be kept in memory. This path is represented by a stack data structure $s$. Algorithm 4 presents the algorithm named $IDDFS$ for generating the assumption by using iterative-deepening depth first search. Algorithm 5 shows the algorithm named $DLS$ for applying depth-limited search. $DLS(d)$ returns an assumption $A(p)$ with a smaller size (than the size of the assumption generated in [10]) if $M_1 \| M_2$ satisfies $p$. It return a counterexample $cex$ if $M_1 \| M_2$ violates $p$. Otherwise, it returns a value named $notfound$ to show that $DLS(d)$ can not find an

**Algorithm 3** Assumption generation algorithm by using DFS.

**Input:** $M_1, M_2, p$: two component models $M_1$ and $M_2$, and a required property $p$
**Output:** $A(p)$ or $cex$: an assumption $A(p)$ with a smaller size (than the size of the assumption generated in [10]) if $M_1 \| M_2$ satisfies $p$, and a counterexample $cex$ otherwise

```
Initially, s = empty {s is an empty stack}
s.push(OT_0) {putting OT_0 into the top of s, where OT_0 = (S_0, E_0, T_0), S_0 = E_0 = {λ},
    and λ is the empty string}
while s ≠ empty do
  OT_i = s.pop() {getting OT_i from the top of s}
  if OT_i contains "?" value then
    for each instance OT of OT_i do
      s.push(OT) {putting OT into the top of s}
    end for
  else
    if OT_i is not closed then
      OT = make_closed(OT_i)
      s.push(OT) {putting OT into the top of s}
    else
      construct a candidate DFA A_i from the closed OT_i
      if Step1(A_i) fails with cex then
        add the suffix e of the counterexample cex to E_i
        OT = update(OT_i)
        s.push(OT) {putting OT into the top of s}
      else
        if Step2(A_i) fails with cex then
          if cex witnesses violation of p then
            return cex
          else
            add the suffix e of the counterexample cex to E_i
            OT = update(OT_i)
            s.push(OT) {putting OT into the top of s}
          end if
        else
          return A_i
        end if
      end if
    end if
  end if
end while
```

assumption with the depth limit $d$ in the search space. The algorithm $IDDFS$ calls $DLS$ with increasing depth limits.

**Algorithm 4** $IDDFS(M_1, M_2, p, max)$

**Input:** $M_1, M_2, p, max$: two component models $M_1$ and $M_2$, a required property $p$, and the maximum depth $max$

**Output:** $A(p)$ or $cex$: an assumption $A(p)$ with a smaller size (than the size of the assumption generated in [10]) if $M_1 \| M_2$ satisfies $p$, and a counterexample $cex$ otherwise

```
for i = 0 to max do
  DLS(i) {Applying depth-limited search with depth limit i}
  if DLS(i) returns an assumption A(p) then
    return A(p)
  else
    if DLS(i) returns a counterexample cex then
      return cex
    end if
  end if
end for
```

### 3.4.3 Reusing the Previous Verification Result

Another idea to reduce the search space of the observation tables of the proposed method presented in Section 3.3 is to reuse the observation table of the current assumption as previous verification result in order to generate the minimal assumption of CBS in the context of the component evolution.

Consider a simple case where a CBS is made up of two component models including a framework $M_1$ and an extension $M_2$. It is known that the compositional system $M_1 \| M_2$ satisfies the property $p$. During the life cycle of this CBS, the component model $M_2$ is evolved to a new component model $M_2'$ by adding some new behaviors to $M_2$. The evolved compositional system $M_1 \| M_2'$ must be rechecked that whether it satisfies the property $p$. For this purpose, the proposed method in this section only checks the evolved component model $M_2'$ satisfying assumption $A_m(p)$, where $A_m(p)$ is a minimal assumption between two components $M_1$ and $M_2$ that is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$. The minimal assumption $A_m(p)$ is generated by using the proposed method for minimal assumption generation presented in Section 3.3. If $M_2'$ satisfies $A_m(p)$, the evolved compositional system $M_1 \| M_2'$ satisfies the property $p$. Otherwise, if $A_m(p)$ is too strong to be satisfied by $M_2'$, a new minimal assumption $A_{mnew}(p)$ between the framework $M_1$ and the evolve model $M_2'$ is regenerated. The proposed method regenerates the new minimal assumption $A_{mnew}(p)$ by applying the proposed method presented in Section 3.3 with the initial observation table as the observation table of $A_m(p)$. By

**Algorithm 5** *DLS*($d$)

**Input:** $d$: depth limit $d$ for depth-limited search
**Output:** $A(p)$ or *cex* or *notfound*

```
Initially, s = empty {s is an empty stack}
s.push(OT_0) {putting OT_0 into the top of s, where OT_0 = (S_0, E_0, T_0), S_0 = E_0 = {λ},
and λ is the empty string}
depth(OT_0) = 0
while s ≠ empty do
   OT_i = s.pop() {getting OT_i from the top of s}
   if depth(OT_i) ≤ d then
      if OT_i contains "?" value then
         for each instance OT of OT_i do
            s.push(OT) {putting OT into the top of s}
            depth(OT) = depth(OT_i) + 1
         end for
      else
         if OT_i is not closed then
            OT = make_closed(OT_i)
            s.push(OT) {putting OT into the top of s}
            depth(OT) = depth(OT_i) + 1
         else
            construct a candidate DFA A_i from the closed OT_i
            if Step1(A_i) fails with cex then
               add the suffix e of the counterexample cex to E_i
               OT = update(OT_i)
               s.push(OT) {putting OT into the top of s}
               depth(OT) = depth(OT_i) + 1
            else
               if Step2(A_i) fails with cex then
                  if cex witnesses violation of p then
                     return cex
                  else
                     add the suffix e of the counterexample cex to E_i
                     OT = update(OT_i)
                     s.push(OT) {putting OT into the top of s}
                     depth(OT) = depth(OT_i) + 1
                  end if
               else
                  return A_i
               end if
            end if
         end if
      end if
   else
      return notfound
   end if
end while
```

staring from the observation table of $A_m(p)$, we can reduce several observation tables of the search space which is used to regenerate the new minimal assumption $A_{mnew}(p)$ of the evolved CBS. Moreover, we know that $L(A_m(p)) \subset L(A_{mnew}(p))$ because $A_m(p)$ is too strong to be satisfied by $M'_2$ and the component evolution means only adding some new behaviors. Thus, we improve the technique for answering membership queries to reduce the number of the instances of each table which contains the "?" entries. At any step $i$ of the learning process, if the current candidate assumption $A_i$ is too strong for $M'_2$ to be satisfied, then $L(A_i)$ exactly is a subset of the language of the assumption be learned. For every $s \in (S \cup S.\Sigma).E$, if $s \in L(A_i)$ (this implies $s \in L(A_W)$), instead of setting $T(s)$ to "?", we set $T(s)$ to $true$. We can reduce several number of the "?" entries by reusing such candidate assumptions. Details of this method will be presented in Section 4.3 of Chapter 4.

## 3.5 Experiment and Evaluation

In order to evaluate the effectiveness of the proposed method, we have implemented the assumption generation method proposed in [10] (called AG tool) and the proposed minimized assumption generation method (called MAG tool) in the Objective Caml (OCaml) [30]. OCaml is a powerful functional programming language that supports numerous architectures for high performance, a bytecode compiler for increased portability, and an interactive loop for experimentation and rapid development [30]. Details of the introduction to functional programming and OCaml can be found in [18, 19] and in [26, 30, 31, 33] respectively. Although the AG tool for L*-based assumption generation method proposed in [10] have been implemented and presented in [35], this tool is not available. This means that there is not any tool which supports the assume-guarantee verification and assumption generation. Thus, in order to compare the effectiveness of both methods, we also have to implement the AG tool.

Figure 3.13 shows the architecture of the implemented AG tool and an example which illustrates how to use the tool. Inputs of this tool are two component model $M_1$ and $M_2$, and a required property $p$ where $M_1$, $M_2$, and $p$ are represented by LTSs. This tool returns an assumption $A$ satisfying the compositional rules if the CBS $M_1 \| M_2$ satisfies $p$, and a counterexample $cex$ to show that $M_1 \| M_2$ violates $p$ otherwise. For example, given two component models $M_1$ as the LTS $Input$ and $M_2$ as the LTS $Output$, and a property as the LTS $p$. The AG tool returns an assumption as LTS $A$ shown in Figure 3.13. With regard to the correctness of our implementation about the AG tool, checking the correctness of the tool is very difficult. The correctness of the AG tool implementation means that we have to check that whether the assumptions generated by this tool are the actual assumptions by checking that each generated assumption $A$ satisfies the compositional rules (i.e., checking that if $\langle A \rangle$ $M_1$ $\langle p \rangle$ and $\langle true \rangle$ $M_2$ $\langle A \rangle$ both hold). For this purpose, we use the tool for verifying concurrent systems called LTSA [13] presented in Section 2.3 of Chapter 2 to check correctness of the generated assumption $A$ by checking the compositional systems $A \| M_1 \| p_{err}$ and $M_2 \| A_{err}$ in the LTSA tool. If both formulas hold, the correctness of $A$ generated by our tool is proven. Figure 3.14 presents an example for checking the correctness of the assumption $A$ generated by the AG tool in the LTSA tool.

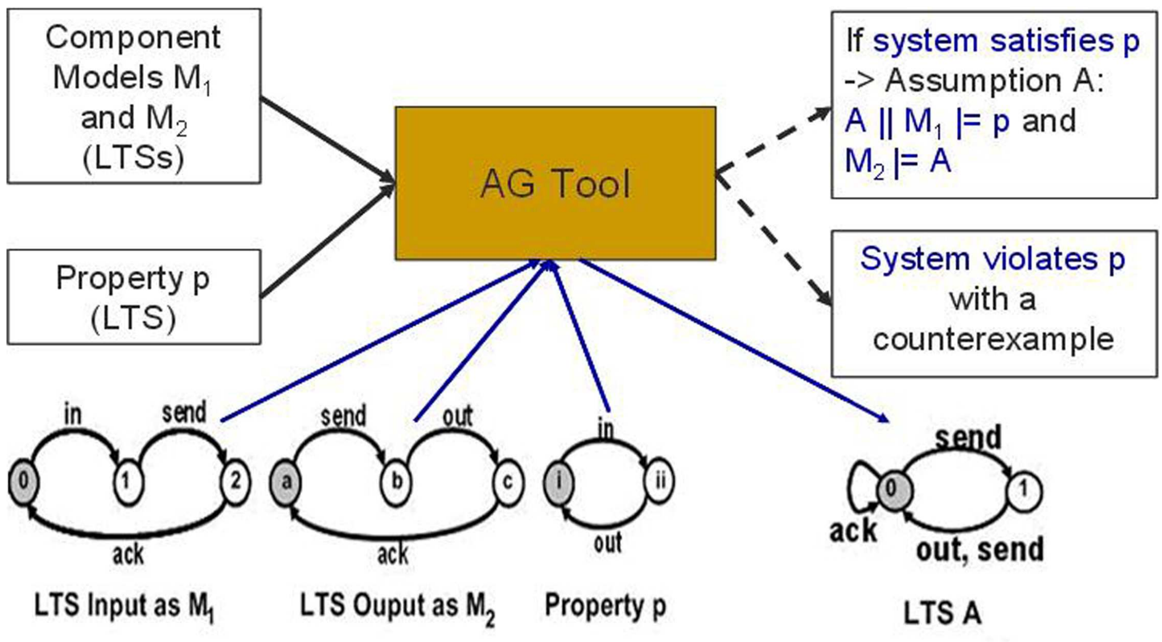



Figure 3.13: The architecture of the AG tool and an example.

With regard to the MAG tool which supports the proposed minimized assumption generation method, Figure 3.15 shows the architecture of the implemented MAG tool and an example which illustrates how to use the tool. Inputs of this tool are two component model $M_1$ and $M_2$, a required property $p$, and an option $o$ where $M_1$, $M_2$, and $p$ are represented by LTSs, and the option $o$ means that we can apply breadth-first search, depth-first search, and iterative-deepening depth first search depending on the given value of $o$. This tool returns a minimal assumption $A_m$ satisfying the compositional rules if the CBS $M_1 \| M_2$ satisfies $p$, and a counterexample $cex$ to show that $M_1 \| M_2$ violates $p$ otherwise. For example, given two component models $M_1$ as the LTS $Input$ and $M_2$ as the LTS $Output$, a property as the LTS $p$, and an option $o$ as empty (breadth-first search). The AG tool returns a minimal assumption as LTS $A_m$ shown in Figure 3.15. The correctness of $A_m$ also is checked by using the LTSA tool presented in Figure 3.16.

Some small component-based software are applied by both method for generating assumptions to compare the effectiveness of the methods. The size, the number of transitions, and the generating time of the generated assumptions are evaluated in this experiment. We also evaluate the rechecking time for each system by reusing the generated

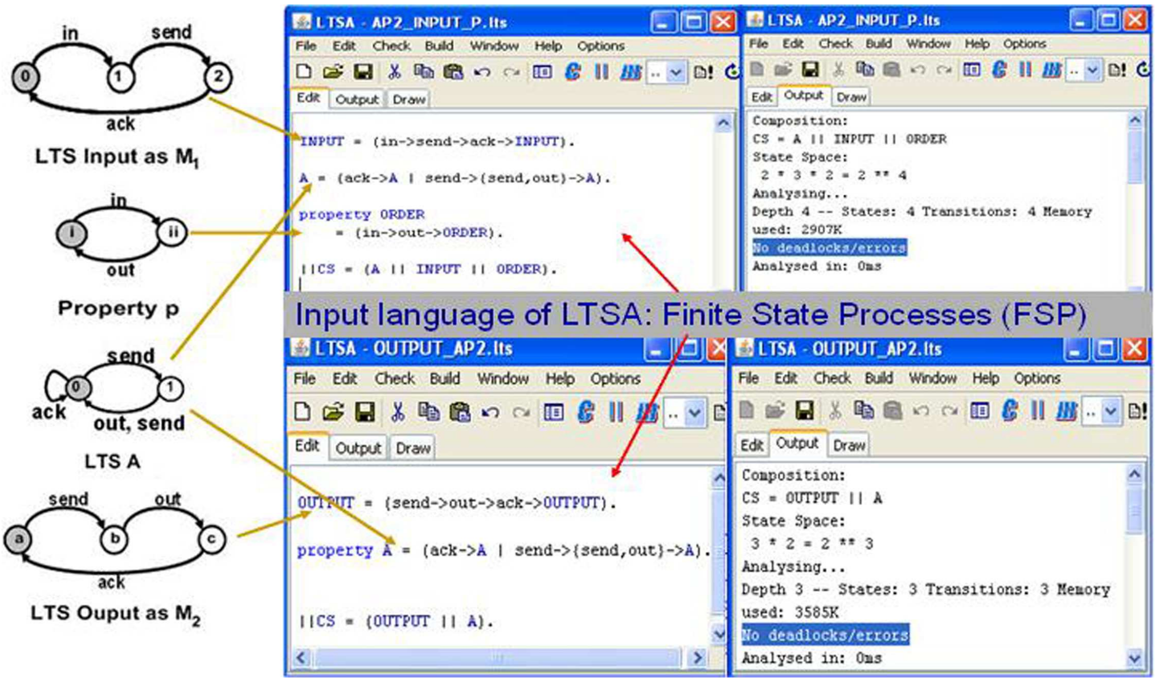



Figure 3.14: An example for checking the correctness of the AG tool via LTSA.

assumptions for checking the compositional rules. Table 3.5 shows experimental results for this purpose. In the results, the system size is the product of the sizes of the software components and the size of the required property for each CBS. Our obtained experimental results imply that the generated minimal assumptions have smaller sizes and number of transitions than the generated ones by the method proposed in [10]. These minimal assumptions are effective for rechecking the systems with a lower cost. However, our method has a higher cost for generating the assumption.

The implemented tools and the illustrative systems which are used in our experimental results can be found at the site [32].

Table 3.1: Experimental results

| System | Sys. size | The current AG Method | | | | Minimized AG Method | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | \|A\| | Trans. of A | Generating Time (ms) | Rechecking Time (ms) | \|A\| | Trans. of A | Generating Time (ms) | Rechecking Time (ms) |
| Simple Channel | 18 | 2 | 4 | 93 | 7.8 | 2 | 3 | 94 | 7.6 |
| Modified Channel | 18 | 4 | 9 | 97 | 9.5 | 2 | 4 | 102 | 6.3 |
| Two Channels | 75 | 3 | 12 | 94 | 37.5 | 3 | 6 | 107 | 23.5 |

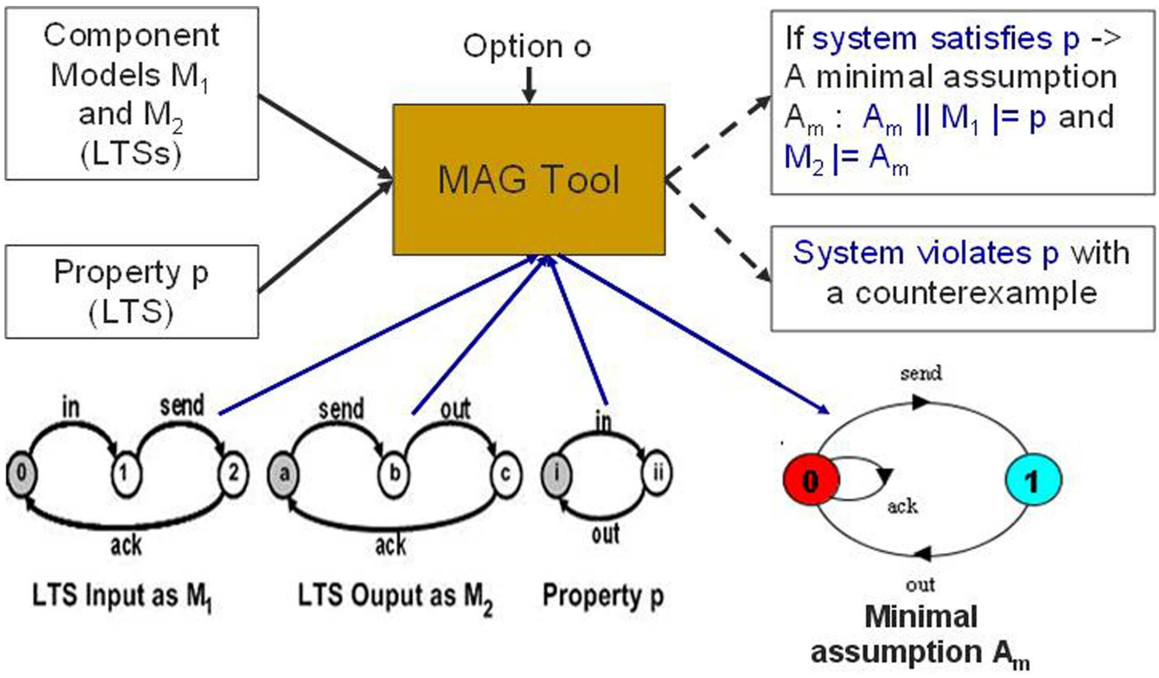



Figure 3.15: The architecture of the MAG tool and an example.

## 3.6 Discussion

With regard to the importance of the minimal assumptions, obtaining smaller assumptions is interesting for several advantages as follows:

- Modular verification of CBS is done by model checking the parallel compositional rules which has the assumption as one of its components. The computational cost of this checking is influenced by the size of the assumption. This means that the cost of verification of CBS is reduced with a smaller assumption which has a smaller size and smaller number of transitions.

- When a component is evolved after adapting some refinements in the context of the software evolution, the whole evolved CBS of many existing components and the evolved component is required to be rechecked [27]. In this case, we can reduce the cost of rechecking the evolved CBS by reusing the smaller assumption.

- Finally, a smaller assumption means less complex behavior so this assumption is easier for a human to understand. This is interesting for checking the large-scale systems.

Consider the practical usefulness of the proposed, the experimental results show that the difference between the generating time in our method and the current method is not so much because the systems used in our experiment are small. In fact, the method proposed in [10] always generates the assumptions at a lower generating time. If we are

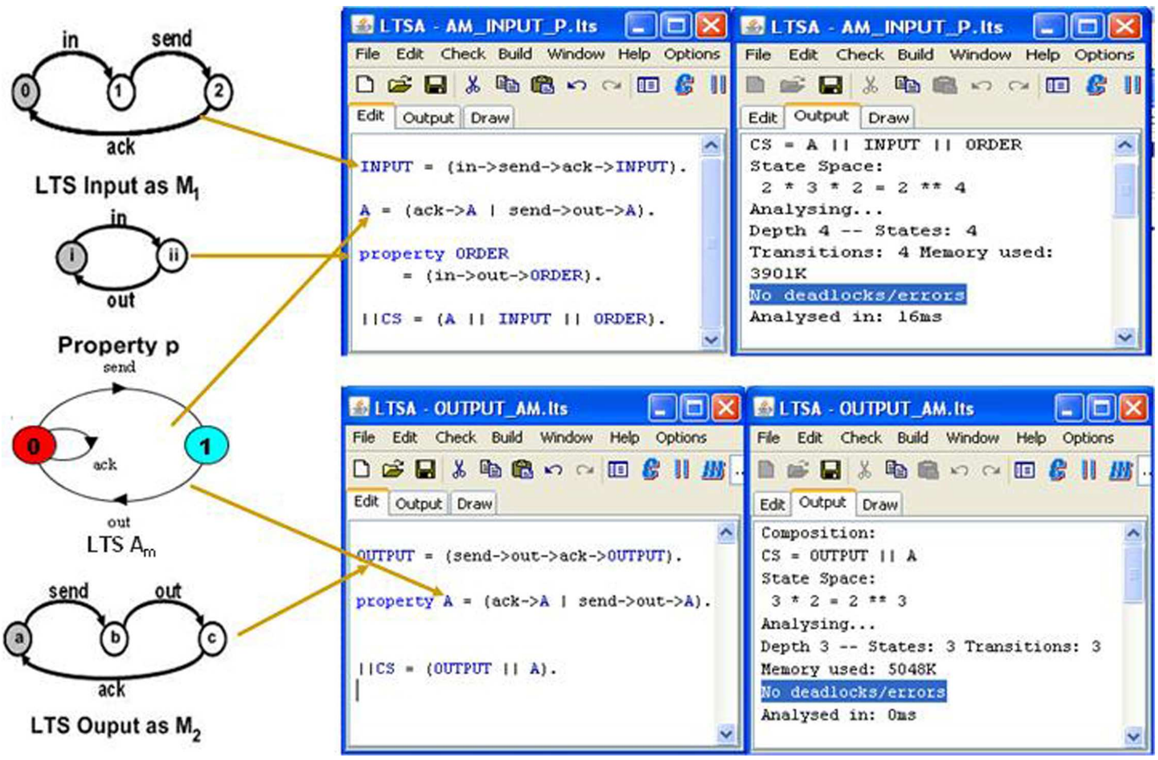



Figure 3.16: An example for checking the correctness of the MAG tool via LTSA.

not interesting in the above advantages, the method proposed in [10] is better than our method for generating assumptions. Otherwise, the generated assumptions are used for rechecking the CBS or are reused for regenerating the new assumptions for rechecking the evolved CBS [27]. In this case, the minimal assumptions generated by our method are useful. However, the breadth-first-search which is used in our work, may be not practical because it consumed too much memory. For the larger-scale systems, the computational cost for generating the minimal assumption is very expensive. An idea to solve this issue is using the iterative-deepening depth first search strategy. The search strategy combines the space efficiency of the depth-first search with the optimality of breadth-first search. It proceeds by running a depth-limited depth-first search repeatedly, each time increasing the depth limit by one. The assumptions generated by using this search strategy are smaller than the assumption generated in [10] but they may be not minimal. Another problem in the proposed method is that the queue has to hold a exponentially growing the number of the observation tables. This makes our method unpractical for the large-scale systems. In order to reduce the search space of the observation tables, we improve the technique for answering membership queries to reduce the number of the instances of each table which contains the "?" entries. At any step $i$ of the learning process, if the current candidate assumption $A_i$ is too strong for $M_2$ to be satisfied, then $L(A_i)$ exactly is a subset of the language of the assumption be learned. For every $s \in (S \cup S.\Sigma).E$, if $s \in L(A_W)$ and $s \in L(A_i)$, instead of setting $T(s)$ to "?", we should set $T(s)$ to *true*.

We can reduce several number of the “?” entries by reusing such candidate assumptions. Another approach for dealing with the problem is to reuse the current assumption of the old CBS for regenerating a new minimal assumption of the evolved CBS in the context of the component evolution presented in Section 4.3 of Chapter 4.

## 3.7 Related Work

There are many works that have been recently proposed in assume-guarantee verification of component-based systems, by several authors. Focusing only on the most recent and closest ones we can refer to [20, 10, 16, 25], to [24], and [22, 27, 28].

D. Giannakopoulou et al. proposes an algorithm for automatically generating the weakest possible assumption for a component to satisfy a required property [16]. Although the motivation of this work is different, the ability to generate the weakest assumption can be used for assume-guarantee verification of component-based software. Based on this work, the work proposed in [10] presents a framework to generate a stronger assumption incrementally and may terminate before the weakest assumption is computed. The key idea of the framework is to generate assumptions as environment for components to satisfy the property. The assumptions are then discharged by the rest of the CBS. However, this framework focuses only on generating the assumptions. The number of states of the generated assumptions is not mentioned in this work. Thus, the assumptions generated by this work are not minimal. This work has been extended in [20, 25] for modular verification of component-based systems at the source code level. Our work improve these works to generate the minimal assumptions in order to reduce the computational cost for rechecking the CBS.

An approach about optimized L*-based assume-guarantee reasoning was proposed by Chaki et al. [24]. The work suggests three optimizations to the L*-based automated assume-guarantee reasoning algorithm for the compositional verification of concurrent systems. The purposes of this work is to reduce the number of the membership queries and the number of the candidate assumptions which are used for generating the assumption, and to minimize the alphabet used by the assumption. However, the core of this approach is the framework proposed in [10]. Thus, the assumptions generated by this work are not minimal. Our work and this work share the motivation for optimizing the framework presented in [10] but we focus on generating the minimal assumptions.

Finally, several works for assume-guarantee verification of evolving software were suggested in [22, 27]. The work in [22] focuses on component substitutability directly from the verification point of view. The purpose of this work is to provide an effective verification procedure that decides whether a component can be replaced with a new one without violation. The work improve the L* algorithm to an improved version called the dynamic L* algorithm by reusing the previous assumptions. However, this work assumes the availability and correctness of models that describe the behaviors of the software components. The works proposed in [27] were suggested to dealing with this issue by providing a method for updating the inaccurate models of the evolved component. These updated models then are used to verify the evolved CBS by apply the improved L* algorithm. Even these works improve the L* algorithm to optimize it, the core of these works is the framework proposed in [10]. As a result, the assumptions generated by these works are not minimal. On the contrary, we focus on generating the minimal assumptions. The minimal assumptions generated by our work may be useful for these works to recheck the evolved at much lower computational costs.

## 3.8 Summary

We have presented a method for generating minimal assumptions for assume-guarantee verification of component-based software. The key idea of this method is finding the minimal assumptions in the search space of the candidate assumptions. These assumptions are strong enough for the components to satisfy a property and weak enough to be satisfied by the rest of the component-based software. In this method, we have improved the technique for answering membership queries of the Teacher which helps the L* to to correctly answer the membership query questions by using the “don’t know” value. By using this technique, the proposed method guarantees that every trace which belongs to the language of the generated assumption exactly belongs to the language be learned. The search space of observation tables used in the proposed method exactly contains the generated observation tables which are used to generate the candidate assumptions. This search space is seen as a search tree where its root is the initial observation table. Finding an assumption with a minimal size such that it satisfies the compositional rules thus is considered a search problem in this search tree. We apply the breadth-first search strategy because this strategy ensures that the generated assumptions are minimal (see Theorem 2). The minimal assumptions generated by the proposed method can be used to recheck the whole component-based software at a lower computational cost. However, the search space explosion for finding the minimal assumption can be occurred when applying the method for the large-scale systems. We have improved the method by using depth-first search, iterative-deepening depth first in order to reduce the search space and the cost for generating the minimal assumptions. Reusing the current assumption of the old CBS for regenerating a new minimal assumption of the evolved CBS in the context of the component evolution also is a promising solution for the issue. We have implemented tools for the assumption generation method proposed in [10] and our minimized assumption generation method. This implementation is used to verify some small component-based software to show the effectiveness of the proposed method.

# Chapter 4

# An Effective Framework for Assume-Guarantee Verification of Evolving Component-Based Software

This chapter proposes an effective framework for assume-guarantee verification of component-based software in the context of the component evolution at the design level. In this framework, if a component model is changed after adapting some refinements, the whole component-based software (CBS) of many existing component models and the evolved component model is not required to be rechecked. The method only checks whether the evolve model satisfies the assumption of the old system. If it is, the evolved CBS still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, a new assumption is regenerated. We propose two methods for the new assumption regeneration: assumption regeneration and minimized assumption regeneration. The methods reuse the current assumption as the previous verification result to regenerate the new assumption at much lower computational cost. An implementation and experimental results are presented.

## 4.1 Introduction

Component-based development has been recognized as one of the most important technical initiatives in software engineering. However, one of the key issues of component-based software (CBS) is to ensure that those separately specified and implemented components do not conflict with each other when composed - the *component consistency* issue. Currently, the popular solution to dealing with this issue is the verification of CBS via model checking [1, 2]. Nonetheless, a major problem of model checking is the *state space explosion*. The motivation in this chapter is to combine the best advantages of the two, i.e., model checking and component-based development to solve this issue in the context of the component evolution at the design time.

An assume-guarantee verification method proposed in [7, 8, 10, 16] has been recognized as a promising approach to dealing with the *state space explosion problem* in model checking by decomposing a verification target about component-based software into components so that we can model check each of them separately. The key idea of this method is to generate assumptions as environments needed for components to satisfy a property. These assumptions are then discharged by the rest of the system. However, assume-guarantee verification in the method is rather closed for the static systems. This means that this method is not prepared for future changes. During the software life cycle, adding or removing behaviors of an existing component seems to be an unavoidable task (the component evolution). Unfortunately, the consequence of these tasks is the whole CBS of many existing components and the evolved component are required to rechecked. In order to recheck the evolve CBS, the current method proposed in [7, 8, 10, 16] performs the assume-guarantee verification on the whole evolved CBS as a new system from scratch. In this case, rechecking on the whole evolved CBS is unnecessary because the changes often focus on a few existing components. It should be better to focus only on the evolved component models and try to reuse the previous verification results to verify the evolved CBS. Moreover, as mentioned in Chapter 3, the assumptions generated by the described method are not minimal. In the context of the rechecking the evolved CBS, the number of states of the generated assumptions should be minimized because the computational cost for rechecking the CBS via model checking is influenced by that number.

This chapter proposes an effective framework to recheck the evolved CBS via the assume-guarantee verification in the context of the component evolution. In the framework, the component evolution means *adding only some new behaviors to the old component without losing the old behaviors*. When a component model is evolved after adapting some refinements, the whole component-based software of many existing component models and the evolved component model is not required to be rechecked. The framework only checks whether the evolve model satisfies the assumption of the old system. If it is, the evolved component-based software still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, a new assumption is regenerated. We propose two methods for new assumption regeneration: assumption regeneration and minimized assumption regeneration. The methods reuse the current assumption as the previous verification result to regenerate the new assumption at much lower computational cost. In order to evaluate the effectiveness of our method, we have implemented a tool for the proposed assumption regeneration method. Some small concurrent systems are evolved by adding some new behaviors to these systems respectively. These evolved systems then are rechecked by applying the proposed method and the assumption generation method proposed in [10] to compare the effectiveness of both methods. The obtained experimental results showed in Section 4.4 presents the effectiveness of the proposed method. In some successful cases, our method can recheck the evolved systems in the fastest way without regenerating the new assumptions (e.g., I/O ver.3). In the other evolved systems, our method can reduce a large number of required membership queries and generated candidate assumptions which are needed to regenerate the new assumptions.

The rest of the chapter is organized as follows. We first describes an effective framework for modular verification of component-based software in the context of the component evolution in Section 4.2. An improvement of the framework for reducing the number of candidate queries also is presented in this section. Section 4.3 is about a minimized assumption regeneration method for modular verification of evolving component-based software. Section 4.4 presents the implemented tools and experimental result to show the effectiveness of the proposed framework. We discuss the advantages and disadvantages of the proposed framework in Section 4.5. Section 4.6 presents related works. Finally, we summarize the chapter in Section 4.7.

## 4.2 An Effective Framework for Assume-Guarantee Verification of Evolving CBS

This section proposes an effective framework for modular verification of component-based software in the context of the component evolution at the design time. In this framework, the component evolution means *adding only some new behaviors to the old component without losing the old behaviors*. In the proposed framework, if a component model is evolved to a new model by adding some new behaviors, the whole evolved CBS of many existing component models and the evolved model are not required to be rechecked. We only focus on the evolved model to recheck the evolved CBS. The framework only checks whether the evolve model satisfies the current assumption of the old CBS. If so, the evolved CBS still satisfies the required property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, a new assumption is regenerated by a new assumption regeneration method. The method reuses the old assumption to reduce a large number of required membership queries and generated candidate assumptions which are needed to regenerate the new assumption. With this approach, we have a faster assume-guarantee method to recheck the evolved CBS.

### 4.2.1 A Framework for Assume-Guarantee Verification of Evolving CBS

Currently, there are many approaches proposed in modular verification of CBS [20, 10, 16, 36, 37, 38, 39]. In these approaches, modular verification is rather closed for the static systems. It is not prepared for future changes. However, evolving of existing components of component-based software seems to be an unavoidable task during the software life cycle. Unfortunately, the consequence of the tasks is the whole evolved software must be rechecked. In order to recheck the evolved CBS, we can apply one of the recently approaches which have been proposed in modular verification and verify the whole evolved CBS as a new system from scratch. In this case, rechecking of the whole evolved CBS is unnecessary because the changes often focus on a few existing components. It should be better to focus only on the evolved component models and try to reuse the previous verification results to verify the evolved CBS.

The main goal in this section is to find a faster method for rechecking the evolved component-based software in the context of the component evolution. The motivation in this method is shown by a simple CBS presented in Figure 4.1. Suppose that there is a simple component-based software which contains a base component model $M_1$ as a fixed framework, and a model $M_2$ as an extension. The extension $M_2$ is plugged into the framework $M_1$ via the parallel composition operator (synchronizing the common actions and interleaving the remaining actions). This kind of CBS only allows us to evolve the behavior of the extension component in the context of the component evolution and it is popular in practice. We know that the compositional CBS $M_1 \| M_2$ satisfies a property $p$ (i.e., $M_1 \| M_2 \models p$). $M_2$ is then evolved to a new component model $M_2'$ by adding some new behaviors to the old component model $M_2$. The major goal of the proposed method is to verify whether the evolved compositional CBS $M_1 \| M_2'$ satisfies *p without rechecking* it from scratch. The method reuses the results of the previous verification (between $M_1$ and $M_2$) in order to have an incremental verification manner to verify the evolved CBS.

In a general view of the proposed approach, when we have verified the old system $M_1 \| M_2$ satisfying the property $p$, we have generated an assumption $A(p)$ that is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$ (i.e., $\langle A(p)\rangle\ M_1\ \langle p\rangle$ and $\langle true\rangle\ M_2\ \langle A(p)\rangle$ both hold) by applying the method presented in Section 3.2 of Chapter 3. Figure 4.2 presents the proposed framework for rechecking the evolved CBS. When the model $M_2$ of the extension component is evolved to a new model $M_2'$, in order to recheck the evolved CBS $M_1 \| M_2'$, the proposed method does not recheck on the whole evolved CBS containing of the framework $M_1$ and the evolved component model $M_2'$. It only checks the assume-guarantee formula $\langle true\rangle\ M_2'\ \langle A(p)\rangle$. If the formula $\langle true\rangle\ M_2'$ $\langle A(p)\rangle$ holds, the evolved CBS $M_1 \| M_2'$ satisfies the property $p$. This is the fastest way

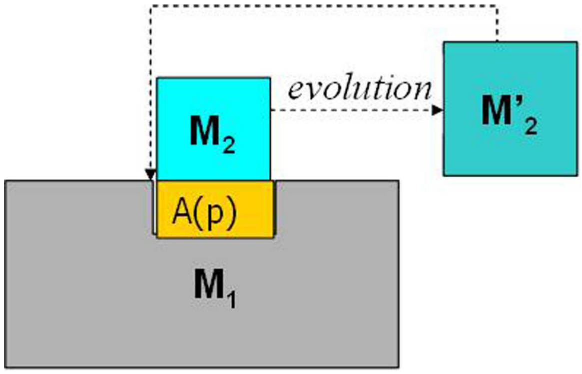



Figure 4.1: A simple case for evolving component-based software.

to recheck the evolved CBS because it rechecked the CBS without regenerating a new assumption. In this case, the assumption $A(p)$ will be seem as the previous verification result to recheck the new system with the future changes in an incremental manner. In practice, the difference between $M_2$ and $M_2'$ is often small thus probability for $M_2'$ satisfying $p$ is very high. Otherwise, the formula $\langle true \rangle\ M_2'\ \langle A(p) \rangle$ does not hold, it returns a counterexample $cex$ to witness this fact. The proposed framework then performs some analysis to determine whether $p$ is indeed violated in the evolve CBS $M_1 \| M_2'$ or if $A(p)$ is too strong to be satisfied by $M_2'$. The counterexample analysis is performed by the Teacher in a way similar to that used for answering membership queries. The Teacher first creates a safety LTS $[cex \uparrow \Sigma]$ from the counterexample $cex$. The Teacher then checks the formula $\langle [cex \uparrow \Sigma] \rangle\ M_1\ \langle p \rangle$ by computing the compositional system $[cex \uparrow \Sigma] \| M_1 \| p_{err}$. If the state error $\pi$ is unreachable in this system, the compositional system $M_1 \| M_2'$ violates the property $p$ (i.e., $M_1 \| M_2' \not\models p$). Otherwise, $A_i$ is too strong to be satisfied by $M_2'$ in the context of $cex$. The proposed framework regenerates a new assumption $A_{new}(p)$ between $M_1$ and $M_2'$ by reusing the entire $A(p)$. The new assumption regeneration method returns a new assumption $A_{new}(p)$ of the evolved CBS if $M_1 \| M_2'$ satisfies the property $p$, and a counterexample $cex$ otherwise.

The proposed framework can reduce a large number of the required membership queries and the generated candidate assumptions which are used to regenerate the new assumptions. In some successful cases where the current assumptions are actual assumptions of the evolved CBS, these CBS are verified in the fastest way without regenerating the new assumptions.

### 4.2.2 New Assumption Regeneration Method

In the above described proposed framework, the new assumption regeneration method is applied to regenerate a new assumption $A_{new}(p)$ when the current assumption $A(p)$

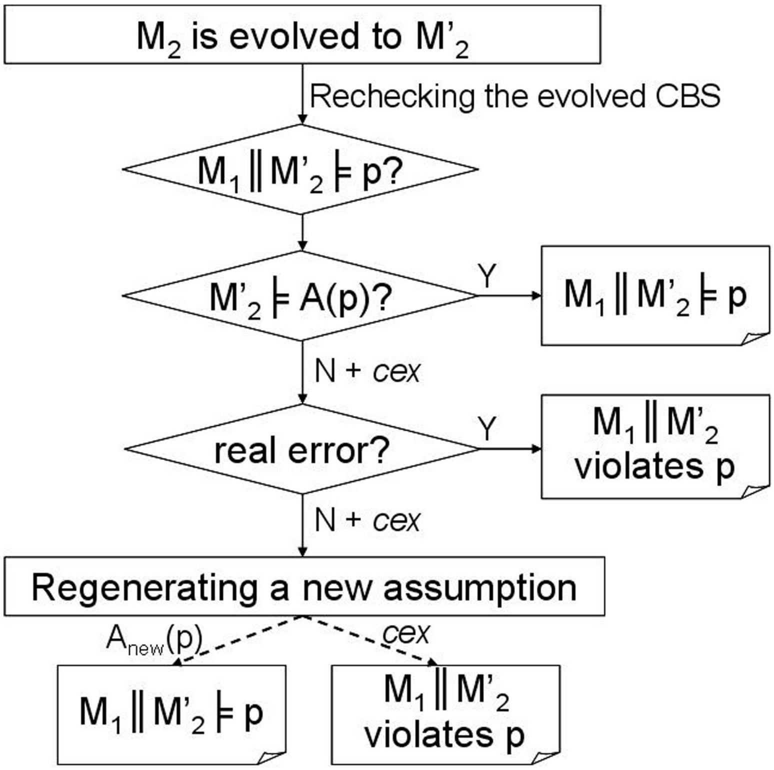



Figure 4.2: The proposed framework for modular verification of evolved CBS.

of the old CBS $M_1 \| M_2$ is too strong to be satisfied by $M_2'$. We propose an effective method for new assumption regeneration by reusing the entire current assumption $A(p)$ in order to reduce the number of the required membership queries and the generated candidate assumptions which are used to regenerate the new assumptions. The new assumption regeneration method returns a new assumption $A_{new}(p)$ if the evolved CBS $M_1 \| M_2'$ satisfies the property $p$, and a counterexample $cex$ otherwise.

Let $U$ be an unknown regular language over some alphabet $\Sigma$. The L* learning algorithm described in Section 3.2 of Chapter 3 learns $U$ and produces a DFA that accepts it. In order to learn $U$, L* builds an observation table $(S, E, T)$ where $S$ and $E$ are a set of prefixes and suffixes respectively, both over $\Sigma^*$. $T$ is a function which maps $(S \cup S.\Sigma).E$ to {true, false}, where the operator "." is defined as follows. Given two sets of event sequences $P$ and $Q$, $P.Q = \{pq \mid p \in P, q \in Q\}$, where $pq$ presents the concatenation of the event sequences $p$ and $q$. The previous method about assumption generation [10] presented in Section 3.2 of Chapter 3 regenerates the new assumption $A_{new}(p)$ also using the L* learning algorithm illustrated in Figure 3.5 of Chapter 3. At the initial step, this method sets the observation table $(S, E, T)$ to the empty observation table (i.e., L* sets $S$ and $E$ to $\{\lambda\}$, where $\lambda$ represents the empty string). Therefore, the initial assumption $A_0$ created from the empty observation table is the strongest assumption (i.e., $A_0 = \lambda$). By this way, the method proposed in [10] regenerates the new assumption $A_{new}(p)$ from scratch. On the contrary, our proposed method in this Section reuses the previous verification results (i.e., the previous assumptions) where possible. After generating the assumption $A(p)$ between $M_1$ and $M_2$, it keeps the current observation table $(S, E, T)$. This table is considered as the results of previous verification. In order to regenerate the new assumption $A_{new}(p)$ of the evolved CBS $M_1 \| M_2'$, the proposed method also uses the L* learning algorithm but with the initial observation table as $(S, E, T)$ which is used to create the current assumption $A(p)$, from now on called the old observation table $(S_{old}, E_{old}, T_{old})$. This means that the initial assumption in the proposed method is $A(p)$ (is not $\lambda$). Because the initial assumption $A(p)$ is weaker than $\lambda$ (see Theorem 3), the proposed method therefore can significantly reduce the number of steps involved in computing $A_{new}(p)$.

In the following more detailed presentation of the improved L*-based algorithm for regenerating $A_{new}(p)$, line numbers refer to the proposed algorithm's illustration in Algorithm 6. Initially, L* sets the initial observation table $(S, E, T)$ to the old observation table $(S_{old}, E_{old}, T_{old})$ (i.e., L* sets $S$ to $S_{old}$, $E$ to $E_{old}$, and $T$ to $T_{old}$) (line 1). When we check the formula $\langle true \rangle\ M_2'\ \langle A(p) \rangle$, the counterexample $cex{\uparrow}\Sigma$, returned by the Teacher which helps L* to answer that whether $A(p)$ is an assumption or not, is analyzed to find a suffix $e$ of $cex{\uparrow}\Sigma$ that witnesses this fact. After that, $e$ must be added to $E$ (line 2). Subsequently, L* updates the function $T$ by making membership queries so that it has a mapping for every string in $(S \cup S.\Sigma).E$ (line 4). The algorithm then checks whether the observation table $(S, E, T)$ is closed [17] (line 5). If the observation table $(S, E, T)$ is not closed, then $sa$ is added to $S$, where $s \in S$ and $a \in \Sigma$ are the elements for which there is no $s' \in S$ (line 6). $T$ must be updated by making membership queries (line 7) because $sa$ has been added to $S$. Line 6 and line 7 are repeated until the table $(S, E, T)$ is closed (line 8). When the observation table $(S, E, T)$ is closed, a candidate assumption DFA $M = \langle Q, \alpha M, \delta, q_0, F \rangle$ is constructed (line 9) from the closed table $(S, E, T)$ using the approach described in the definition about DFAs in Chapter 2. The candidate DFA $M$ is presented as a conjecture to the Teacher (line 10). If the Teacher replies $true$ (i.e., $L(M) = U$) (line 11), L* returns $M$ and terminates (line 12), otherwise L* receives a counterexample $cex \in \Sigma^*$ from the Teacher. The counterexample $cex$ is analyzed by L* to find a suffix $e$ of $cex$ that witnesses a difference between $L(M)$ and $U$. After that, $e$ must be added to $E$ (line 14). It will cause the next conjectured automaton to reflect this difference. When $e$ has been added to $E$, L* iterates the entire process by looping around to line 4.

In order to regenerate the new assumption $A_{new}(p)$ of the evolved CBS $M_1 \| M_2'$, the described improved L*-based algorithm learns the unknown language $U = L(A_W)$ over the alphabet $\Sigma = \alpha A_W = (\alpha M_1 \cup \alpha p) \cap \alpha M_2'$, where $L(A_W)$ is the language of the weakest assumption $A_W$ defined in [16]. Figure 4.3 presents an iterative framework to

**Algorithm 6** The improved L* learning algorithm for new assumption regeneration.

**Input:** $U, \Sigma, (S_{old}, E_{old}, T_{old})$: an unknown regular language $U$ over some alphabet $\Sigma$ and the current observation table $(S_{old}, E_{old}, T_{old})$
**Output:** $M$: a DFA $M$ such that $M$ is a minimal deterministic automata corresponding to $U$ and $L(M) = U$

Initially, $S = S_{old}, E = E_{old}, T = T_{old}$
add $e \in \Sigma^*$ that witnesses the *counterexample cex* to $E$
**loop**
  update $T$ using *membership queries*
  **while** $(S, E, T)$ is *not closed* **do**
    add $sa$ to $S$ to make $S$ closed, where $s \in S$ and $a \in \Sigma$
    update $T$ using *membership queries*
  **end while**
  construct a *candidate* DFA $M$ from the closed $(S, E, T)$
  present an *equivalence query*: $L(M) = U$?
  **if** $M$ is correct **then**
    **return** $M$
  **else**
    add $e \in \Sigma^*$ that witnesses the *counterexample cex* to $E$
  **end if**
**end loop**

illustrate the proposed improved L*-based algorithm for new assumption regeneration. At each iteration $i$, a candidate assumption $A_i$ is produced by the L* learning based on some knowledge about the system and on the results of the previous iteration. The two steps of the compositional rules are then applied. Step 1 checks whether $M_1$ satisfies $p$ in environments that guarantee $A_i$. If the result is $false$, it means that $A_i$ is *too weak* for $M_1$ to satisfy $p$. For this reason, $A_i$ must be strengthened with the help of the counterexample *cex* returned by this step. Otherwise, the result of this step is $true$, it means that $A_i$ is strong enough for $M_1$ to satisfy $p$. The step 2 is then applied to check that if the evolved component model $M_2'$ satisfies $A_i$. If this step returns $true$, the property $p$ holds in the evolved CBS $M_1 \| M_2'$ (i.e., $M_1 \| M_2' \models p$). In this case, the algorithm terminates and returns a new assumption $A_{new}(p) = A_i$. Otherwise, this step returns $false$ with a counterexample *cex* to witness this fact. We have to identify that whether $p$ is indeed violated in the evolved CBS $M_1 \| M_2'$ or $A_i$ is too strong to be satisfied by $M_2'$ by analyzing the counterexample *cex*. This analysis checks that whether $p$ is violated by $M_1$ in the context of the counterexample *cex* by checking the formula $[cex] \| M_1 \not\models p$, where $[cex]$ is a LTS defined in Section 2.1 of Chapter 2. If the property $p$ does not hold in the compositional system $[cex] \| M_1$, the evolved CBS $M_1 \| M_2'$ violates the property $p$. In this case, the algorithm terminates and returns the counterexample *cex* returned by this step. Otherwise, $A_i$ is too strong to be satisfied by $M_2'$. The candidate assumption $A_i$ therefore must be weakened in the iteration $i + 1$. The result of such weakening will be that at least the behavior that the counterexample *cex* represents will be allowed by candidate assumption $A_{i+1}$. New candidate assumption may be too weak, and therefore the entire process must be repeated.

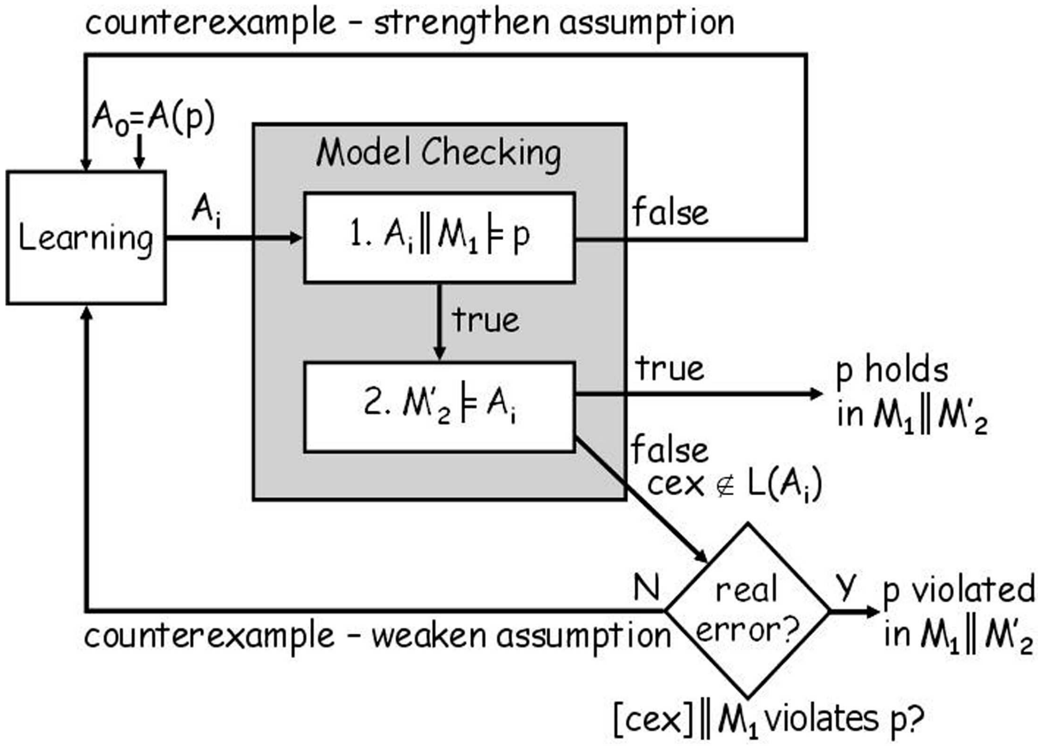



Figure 4.3: The iterative framework for the new assumption regeneration using the improved L* learning algorithm.

With regard to effectiveness of the proposed algorithm, Figure 4.4 intuitively describes the process for the new assumption regeneration by using the improved L* learning algorithm with the initial assumption $A(p)$. In this figure, $\lambda$ is the strongest assumption. It is the initial assumption in the L* used to generate the old assumption $A(p)$ of the old compositional CBS $M_1 \| M_2$ [10]. In the case where the evolved component model $M_2'$ does not satisfy the property $p$ (i.e., $M_2' \not\models p$), we use the old assumption $A(p)$ as the initial assumption for the improved L* to regenerate the new assumption $A_{new}(p)$ of the evolved compositional CBS $M_1 \| M_2'$. It is clear to show that the new assumption $A_{new}(p)$ is weaker than $A(p)$ because the old assumption $A(p)$ is too strong to be satisfied by $M_2'$ and $A_{new}(p)$ is strong enough to be satisfied by $M_2'$. Intuitively, by starting at the old assumption $A(p)$, the proposed method can reduce the large number of steps required in the assumption regeneration process.

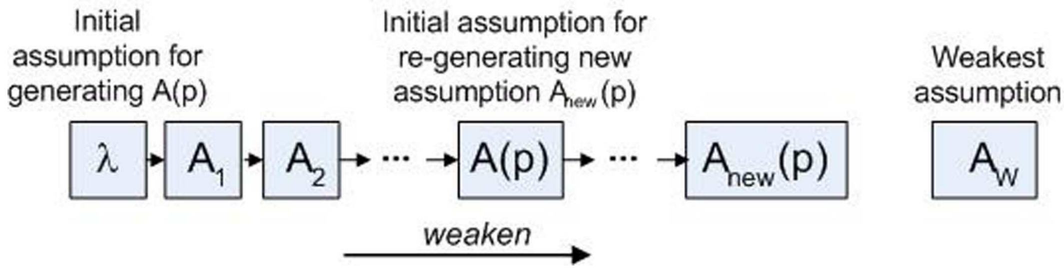



Figure 4.4: The process for new assumption regeneration using the improved L*.

### 4.2.3 Correctness and Termination

Correctness and termination of the proposed method for assumption regeneration is proved by the following theorem.

**Theorem 3** *Given an accurate component model $M_1$, an evolved component model $M_2'$ which is an evolution of a model $M_2$, a required property $p$, and an assumption $A(p)$ which is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$, the proposed method for the new assumption regeneration terminates and returns a new assumption $A_{new}(p)$ if the evolved compositional CBS $M_1 \| M_2'$ satisfies $p$, and false otherwise.*

**Proof** The proposed method uses two steps of the compositional rule (i.e., $\langle A(p)\rangle$ $M_1$ $\langle p\rangle$ and $\langle \text{true}\rangle$ $M_2'$ $\langle A(p)\rangle$) to answer the question of whether the candidate assumption $A_i$ produced by the L* Learner is an assumption or not. It only returns true and a new assumption $A_{new}(p) = A_i$ when both steps return true, and therefore its correctness is guaranteed by the compositional rule. The proposed method returns a real error when it detects a trace $\sigma$ of $M_2'$ which violates the property $p$ when simulated on $M_1$. In this case, it implies that $M_1 \| M_2'$ violates $p$. The remaining problem is to prove that whether we always achieve the new assumption $A_{new}(p)$ from the old assumption $A(p)$ if $M_1 \| M_2'$ satisfies $p$. In the case where the new assumption is regenerated, we know that $M_2' \not\models A(p)$ because $A(p)$ is too strong for $M_2'$ to satisfy. We also know that $M_2' \models A_{new}(p)$ because $A_{new}(p)$ satisfies the compositional rule. These observations imply that $A_{new}(p)$ is weaker than $A(p)$. By using the L* learning algorithm, we can obtain directly a weaker candidate assumption from a stronger one [10]. It means that we always achieve the new assumption $A_{new}(p)$ directly from $A(p)$. With regard to the termination of the proposed method, at any iteration, the algorithm returns true or false (i.e., $M_1 \| M_2' \not\models p$) and terminates or continues by providing a counterexample from the L* Learner. By the correctness of L* [9, 17], we ensure that if the L* learning algorithm keeps receiving counterexamples, in the worst case, the algorithm will eventually produce the weakest assumption $A_W$ and terminates, by the definition of $A_W$ [16]. ∎

### 4.2.4 Characteristics of the Proposed Method

In order to recheck the evolved CBS $M_1 \| M_2'$ with the current assumption $A(p)$, if $M_2'$ satisfies $A(p)$ then the evolved CBS is rechecked successful without regenerating a new assumption. This is the successful case to show the effectiveness of the proposed method. Otherwise, this method must regenerate a new assumption $A_{new}(p)$ by reusing the entire $A(p)$.

Let $n$ and $n'$ be sizes of the DFAs used to get $A(p)$ and $A_{new}(p)$ respectively. As for the assumption generation method proposed in [10], the assumption $A(p)$ is generated by making at most $n$ - 1 incorrect candidate assumptions, and the assumption $A_{new}(p)$ is generated by making at most $n'$ - 1 incorrect candidate assumptions. The numbers of required membership queries used to generate $A(p)$ and $A_{new}(p)$ by this method are $O(kn^2 + n \log m)$ and $O(kn'^2 + n' \log m)$ respectively [10], where $k$ is size of the alphabet $\Sigma$, and $m$ is the length of the longest counterexample.

With regard to the proposed method, the new assumption $A_{new}(p)$ is regenerated by making at most $n' - n$ incorrect candidate assumptions, and the number of required membership queries used to regenerate $A_{new}(p)$ is $O(k(n'^2 - n^2) + (n' - n) \log m)$. These observations imply that our method is more effective than the method proposed in [10] in the context of the rechecking of the evolving CBS.

### 4.2.5 Reducing the Number of Candidate Queries

Recall the iterative framework for the new assumption regeneration shown in Figure 4.3. Although the proposed framework reuses the old assumption $A(p)$ as an effective method to reduce the large number of the generated candidate assumptions which are needed to regenerate the new assumption $A_{new}(p)$, the core of this framework is based on the framework for L*-based assumption generation proposed in [10]. In the proposed framework, at each iteration $i$, a candidate assumption $A_i$ is produced. In order to check that whether $A_i$ is an actual assumption of the evolved CBS, a candidate query "is $L(A_i) = U$?" is sent to the Teacher. The two steps of the compositional rules are applied by the Teacher to answer the candidate query. However, There are some $A_i$ which are not assumption candidates without using candidate queries. To detect such $A_i$ as a way to reduce the number of the candidate queries which are sent to the Teacher is very important because that number primarily influences on the computational cost for assumption regeneration. In this section, we presents an improvement of the proposed method for the new assumption regeneration to reduce the computational cost of assumption regeneration.

For each candidate assumption $A_i$ produced by L* at iteration $i$, the Teacher checks that whether $A_i$ satisfies the compositional rules (i.e., $\langle A_i \rangle\ M_1\ \langle p \rangle$ and $\langle true \rangle\ M_2'\ \langle A_i \rangle$) of the evolved CBS in order to answer the candidate query about "is $L(A_i) = U$?". For this purpose, the Teacher first checks the step 1 (whether $M_1$ satisfies $p$ in environments that guarantee $A_i$) by computing the formula $\langle A_i \rangle\ M_1\ \langle p \rangle$. If the result is $false$ with a counterexample $cex$, it means that $A_i$ is *too weak* for $M_1$ to satisfy $p$ (i.e., $A_i$ does not restrict the environment enough for $p$ to be satisfied by $M_1$). Thus, $A_i$ must be strengthened which corresponds to removing behaviors from it with the help of the counterexample $cex$ produced by this step. In the context of the next candidate assumption $A_{i+1}$, component $M_1$ should at least not exhibit the violating behavior reflected by this counterexample. Formally, this step returns $cex$ such that $cex \in L(A_i)$ but $cex \notin U$, where $U$ is the language of the target assumption be learned. In this case, the strengthening $A_i$ means that the counterexample $cex$ should be removed from $L(A_i)$. Otherwise, the result of the step 1 is $true$, it means that $A_i$ is strong enough for $M_1$ to satisfy $p$. The step 2 is then applied to check that if the evolved component model $M'_2$ satisfies $A_i$ by computing the formula $\langle true \rangle\ M'_2\ \langle A_i \rangle$. If this step returns true, the property $p$ holds in the evolved CBS $M_1 \| M'_2$ ($M_1 \| M'_2 \models p$) and the proposed algorithm terminates. Otherwise, this step returns $false$ with a counterexample $cex$, further analysis is required to identify whether $p$ is indeed violated in $M_1 \| M'_2$ or $A_i$ is too strong to be satisfied by $M'_2$. If $A_i$ is too strong to be satisfied by $M'_2$, $A_i$ therefore must be weakened (i.e., behaviors must be added) in the iteration $i + 1$. The result of such weakening will be that at least the behavior that the counterexample $cex$ represents will be allowed by candidate assumption $A_{i+1}$. Formally, this step returns $cex$ such that $cex \in U$ but $cex \notin L(A_i)$. In this case, the weakening $A_i$ means that the counterexample $cex$ should be added to $L(A_i)$. For example, Figure 4.6 presents the meaning of the strengthening and weakening of the generated candidate assumptions for the new assumption regeneration of the evolved CBS shown in Figure 4.5.

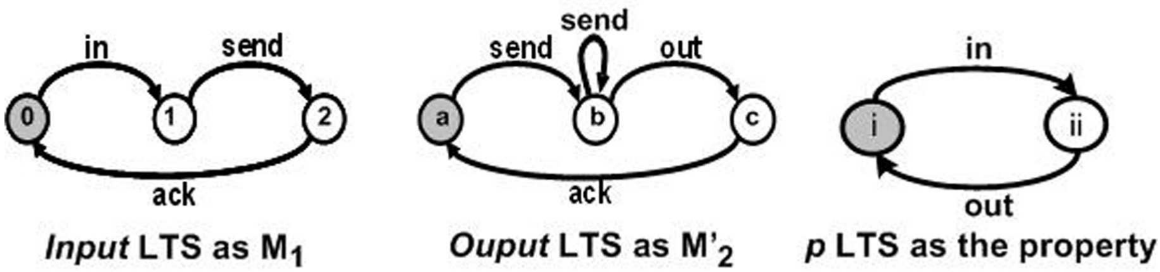



Figure 4.5: An evolved CBS including the model $M_1$, the evolved model $M'_2$, and the required property.

We explain an improvement of the proposed method for the new assumption regeneration by detecting the candidate assumptions which are not assumptions without using the candidate queries as follows.

Suppose that the formula $\langle A_i \rangle\ M_1\ \langle p \rangle$ returns $false$ with a counterexample $cex$ for the generated candidate assumption $A_i$ at the iteration $i$ of the proposed framework presented

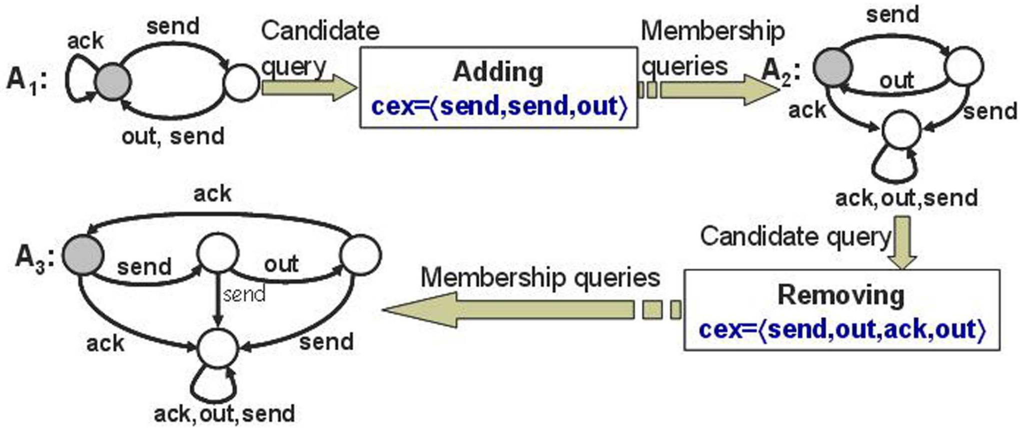



Figure 4.6: The meaning of the strengthening and weakening of the generated candidate assumptions.

in Figure 4.3. $A_i$ must be strengthened which corresponds to remove the counterexample $cex\uparrow\Sigma$ from $L(A_i)$. In this case, $cex$ is called a negative counterexample. For the purpose, the proposed algorithm adds a suffix $c$ of $cex\uparrow\Sigma$ to the set of suffixes $E$ of the current observation table $S, E, T$. This table may be updated to be closed by using the membership queries. The next candidate assumption $A_{i+1}$ is produced by the closed table. However, this refinement process does not guarantee the elimination of the counterexample $cex\uparrow\Sigma$ from $L(A_i)$. Thus, the counterexample $cex\uparrow\Sigma$ may still be accepted by the next candidate assumption $A_{i+1}$.

In a similar case where the formula $\langle true\rangle\ M_2'\ \langle A_i\rangle$ returns $false$ with a counterexample $cex$ for the generated candidate assumption $A_i$ at the iteration $i$. If $A_i$ is too strong to be satisfied by $M_2'$, $A_i$ therefore must be weakened which corresponds to add the counterexample $cex\uparrow\Sigma$ to $L(A_i)$. In this case, $cex$ is called a positive counterexample. For the goal, the proposed algorithm adds a suffix $c$ of $cex\uparrow\Sigma$ to the set of suffixes $E$ of the current observation table $S, E, T$. This table may be updated to be closed by using the membership queries. The next candidate assumption $A_{i+1}$ is produced by the closed table. However, this refinement process does not guarantee the addition of the counterexample $cex\uparrow\Sigma$ to $L(A_i)$. Thus, the counterexample $cex\uparrow\Sigma$ may still be rejected by the next candidate assumption $A_{i+1}$.

In order to detect such candidate assumptions, for every candidate $A_{i+1}$ obtained by refining on a negative counterexample, we check that whether $cex\uparrow\Sigma \in L(A_{i+1})$ before sending a candidate query "is $L(A_{i+1}) = U$?" to the Teacher. If $cex\uparrow\Sigma \in L(A_{i+1})$, $A_{i+1}$ is not an assumption without checking the compositional rules (without using the candidate query). In this case, we repeat the refinement process on $A_{i+1}$ using $cex$ instead of performing the candidate query "is $L(A_{i+1}) = U$?". Similarity with the positive counterexample, we check that whether the positive counterexample $cex\uparrow\Sigma \notin L(A_{i+1})$ before sending a candidate query "is $L(A_{i+1}) = U$?" to the Teacher. If so, $cex$ is reused to further refine $A_{i+1}$. This improvement can reduce the number of candidate queries which are needed for the new assumption regeneration presented in Figure 4.3.

### 4.2.6 Examples

Figure 4.7 describes an illustrative concurrent CBS which contains two component model $M_1$ and $M_2$. The model $M_1$ is plugged into the model $M_2$ via the parallel composition operator defined in Section 2.1 of Chapter 2. In this CBS, the LTS of $M_1$ is the $Input$ LTS, and the LTS of $M_2$ is the $Output$ LTS. This concurrent CBS is an extension of the CBS described in Figure 3.6 of Chapter 3. The CSB means that the sender ($Input$ LTS) can acquire messages via two different input actions $in1$ and $in2$, and then proceeds to send the message on one of two corresponding channels. The receiver ($Output$ LTS) acts analogously. A required property $p$ and the current assumption $A(p)$ of the CBS also described in this figure. The assumption $A(p)$ is generated by using the framework illustrated in Figure 3.5 in Chapter 3 that is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$.

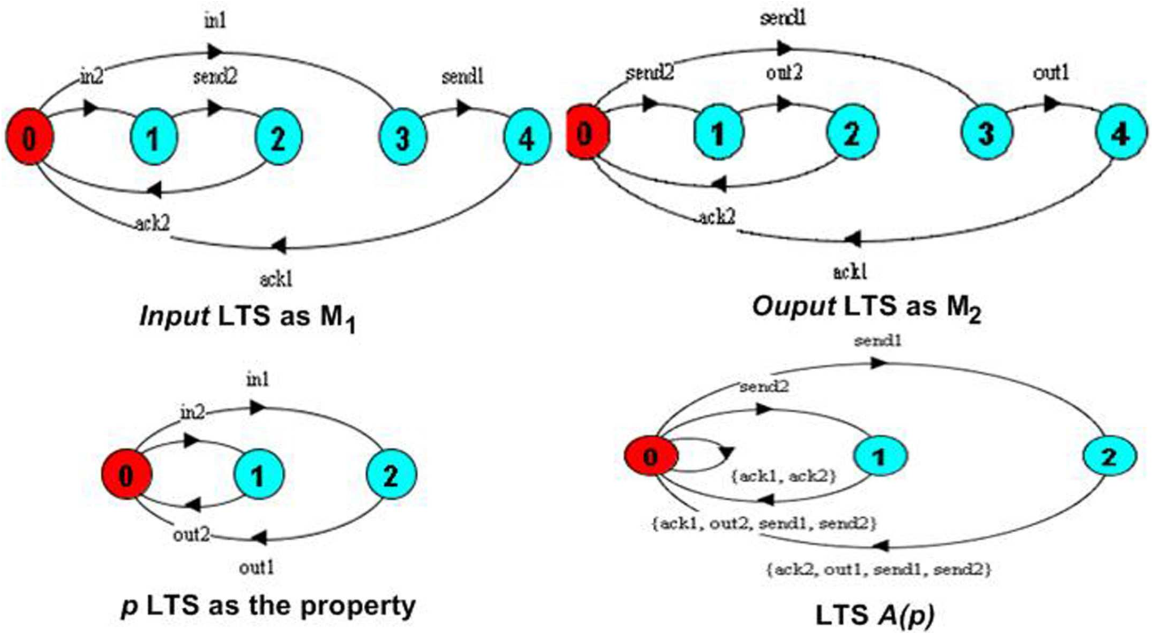



Figure 4.7: Component models, order property, and assumption $A(p)$ of the illustrative CBS.

The extension component model $M_2$ is then evolved to a evolved model $M_2'$ presented in Figure 4.8 by adding a new behavior which allows multiple $send1$ actions to occur before producing $out1$ action. In order to recheck the evolved CBS $M_1 \| M_2'$, the proposed framework only checks the formula $\langle true \rangle\ M_2'\ \langle A(p) \rangle$. This checking returns $false$ and the counterexample analysis implies that $A(p)$ is too strong to be satisfied by $M_2'$. A new assumption $A_{new}(p)$ for the evolved CBS $M_1 \| M_2'$ must be regenerated. For the purpose, the new assumption regeneration method reuses the old assumption $A(p)$ to regenerate the new assumption $A_{new}(p)$ shown in Figure 4.8. In the assumption generation method proposed in [10], for the same goal, the method has generated 6 candidate assumptions and 294 membership queries to generate $A_{new}(p)$. We regenerate $A_{new}(p)$ at much lower computational cost with 3 generated candidate assumptions and 210 required membership queries.

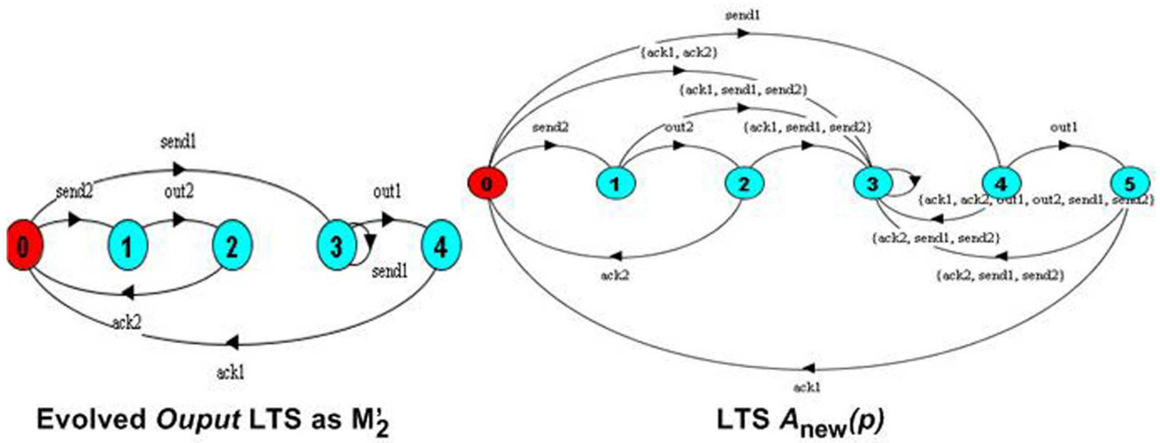



Figure 4.8: The evolved model $M_2'$ and the new assumption $A_{new}(p)$ of the evolved CBS.

Consider the next component evolution of the described evolved CBS where the evolved model $M_2'$ is evolved continuously to a evolved model $M_2''$ shown in Figure 4.9 by adding a new behavior which allows multiple *send*2 actions to occur before producing *out*2 action. The proposed framework then rechecks the evolved CBS $M_1 \| M_2''$ by checking the formula $\langle true \rangle\ M_2''\ \langle A_{new}(p) \rangle$. The result of this checking is *true*. This means that the evolved CBS $M_1 \| M_2''$ satisfies the property $p$ without regenerating a new assumption. This is a successful example to show the effectiveness of the proposed method. In such cases, our method can recheck the evolved systems in the fastest way.

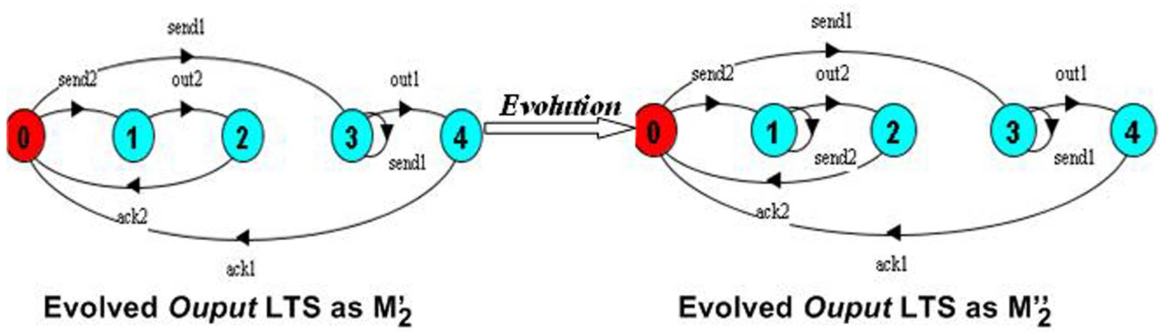



Figure 4.9: The evolved model $M_2''$ of the model $M_2'$.

## 4.3 A Minimized Assumption Regeneration Method for Assume-Guarantee Verification of Evolving CBS

This section proposes a minimized assumption regeneration method for modular verification of component-based software in the context of the component evolution. This method is an improvement of the minimized assumption generation method presented in Section 3.3 of Chapter 3. In this method, if the current assumption is too strong to be satisfied by the evolved component model, a new minimal assumption is regenerated. The method reuses the assumption in order to reduce the search space of the observation tables which are used for regenerating the new minimal assumption of the evolved CBS.

Although the proposed new assumption regeneration method presented in Section 4.2 is an effective approach to regenerate the new assumption at much lower computational cost, the core of this approach is the framework proposed in [10]. Thus, the new assumptions regenerated by the proposed method are not minimal. As mentioned in Section 3.6 of Chapter 3, obtaining minimal assumptions is interesting for several advantages. The key advantage is that the minimal assumptions can be used to recheck the whole CBS at much lower computational cost. However, in the proposed algorithm for minimal assumption generation presented in Algorithm 2 of Chapter 3, the queue has to hold a exponentially growing the number of the observation tables. This makes the method unpractical for the large-scale systems because it consumed too much memory. For the large-scale systems, the computational cost for regenerating the minimal assumption is very expensive. In the context of the component evolution, when the current assumption is too strong to be satisfied by the evolved component model presented in Section 4.2, we guarantee that the new minimal assumption can be obtained directly from the strong assumption.

Algorithm 7 presents the proposed algorithm for the new assumption regeneration by improving the algorithm for minimized assumption generation presented in Algorithm 2. Recall the proposed framework presented in Subsection 4.2.1, when the model $M_2$ of the extension component is evolved to a new model $M_2'$, in order to recheck the evolved CBS $M_1 \,\| M_2'$, the proposed framework does not recheck on the whole evolved CBS. It only checks whether the assume-guarantee formula $\langle true \rangle\ M_2'\ \langle A(p) \rangle$. If so, the evolved CBS $M_1 \,\| M_2'$ still satisfies the property $p$. Otherwise, the formula does not hold, it returns a counterexample $cex$ to witness this fact. The proposed framework then performs some analysis to determine whether $p$ is indeed violated in the evolve CBS $M_1 \| M_2'$ or if $A(p)$ is too strong to be satisfied by $M_2'$.

If $A_i$ is too strong to be satisfied by $M_2'$ in the context of $cex$, the minimized assumption regeneration method presented in Algorithm 7 is applied to regenerates a new minimal assumption $A_{nnew}(p)$ between $M_1$ and $M'_2$ by reusing the entire $A(p)$. The method returns a new minimal assumption $A_{mnew}(p)$ of the evolved CBS if $M_1 \| M'_2$ satisfies the property $p$, and a counterexample *cex* otherwise. In order to regenerate the new minimal assumption $A_{mnew}(p)$ of the evolved CBS, at the initial step, instead of putting the initial observation table $OT_0 = (S_0, E_0, T_0)$ into the empty queue $q$ as the root of the search space of observation tables, the method sets the initial observation table $OT_0$ to the observation table $OT_{old} = (S_{old}, E_{old}, T_{old})$ of the current assumption $A(p)$ (line 2). A suffix $e$ of *cex* to $E_0$ of the table $OT_0$ in order to weaken the assumption $A(p)$ because $A(p)$ is too strong to be satisfied by $M'_2$ (line 3). After that, the table $OT_0$ is updated by calling the procedure named $update(OT_i)$ (line 4). The remaining of the proposed algorithm is the same as the algorithm presented in Algorithm 2. By this approach, we can reduce the search space of the observation tables which are used for regenerating the new minimal assumption of the evolved CBS.

## 4.4 Experiment and Evaluation

In order to evaluate the effectiveness of the proposed framework for assume-guarantee verification of evolving CBS presented in Section 4.2.1, we have implemented the L*-based assumption generation method proposed in [10] (the AG tool shown in Section 3.5 of Chapter 3) and the proposed assumption regeneration method (called AR tool) in the Objective Caml (OCaml) functional programming language [30]. Figure 4.10 shows the architecture of the implemented AR tool and an example which illustrates how to use the tool. Inputs of this tool are a component model $M_1$ as framework, an evolved model $M'_2$ of the old model $M_2$, an assumption $A(p)$ of the old CBS $M_1 \| M_2$, and a required property $p$ where $M_1$, $M'_2$, $A(p)$, and $p$ are represented by LTSs. This tool returns *true* if the evolved model $M_2$ satisfies $A(p)$ (the evolved CBS $M_1 \| M'_2$ still satisfies $p$ without regenerating a new assumption), a new assumption $A_{new}$ satisfying the compositional rules if the CBS $M_1 \| M'_2$ satisfies $p$, and a counterexample *cex* to show that $M_1 \| M'_2$ violates $p$ otherwise. For example, given two component models $M_1$ as the LTS *Input* and $M'_2$ as the evolved LTS *Output*, a property as the LTS $p$, and an assumption as LTS $A$. The AR tool returns a new assumption as LTS $A_{new}$ shown in Figure 4.10. The correctness of $A_{new}$ also is checked by using the LTSA tool.

The concurrent system Input/Output (I/O ver.1) illustrated in Figure 4.5 and two evolved versions (I/O ver.2 and I/O ver.3) of this system have been verified by applying both methods. Because the computational cost for assumption generation is influenced by the number of the required membership queries and the number of the generated candidate assumptions, we only compute these measures for each evolved system to compare the effectiveness of the methods. Table 4.1 shows experimental results for this purpose.

**Algorithm 7** Minimized assumption regeneration.

**Input:** $M_1, M_2', p, OT_{old} = (S_{old}, E_{old}, T_{old})$: existing component model $M_1$, evolved model $M_2'$, a required property $p$, and the current observation table $OT_{old}$
**Output:** $A_m(p)$ or $cex$: an assumption $A_m(p)$ with a smallest size if $M_1 \| M_2$ satisfies $p$, and a counterexample $cex$ otherwise

```
Initially, q = empty {q is an empty queue}
OT_0 = OT_old {S_0 = S_old, E_0 = E_old, T_0 = T_old}
add the suffix e of the counterexample cex to E_0
OT_0 = update(OT_0)
q.put(OT_0)
while q ≠ empty do
  OT_i = q.get() {getting OT_i from the top of q}
  if OT_i contains "?" value then
    for each instance OT of OT_i do
      q.put(OT) {putting OT into q}
    end for
  else
    if OT_i is not closed then
      OT = make_closed(OT_i)
      q.put(OT)
    else
      construct a candidate DFA A_i from the closed OT_i
      if Step1(A_i) fails with cex then
        add the suffix e of the counterexample cex to E_i
        OT = update(OT_i)
        q.put(OT)
      else
        if Step2(A_i) fails with cex then
          if cex witnesses violation of p then
            return cex
          else
            add the suffix e of the counterexample cex to E_i
            OT = update(OT_i)
            q.put(OT)
          end if
        else
          return A_i
        end if
      end if
    end if
  end if
end while
```

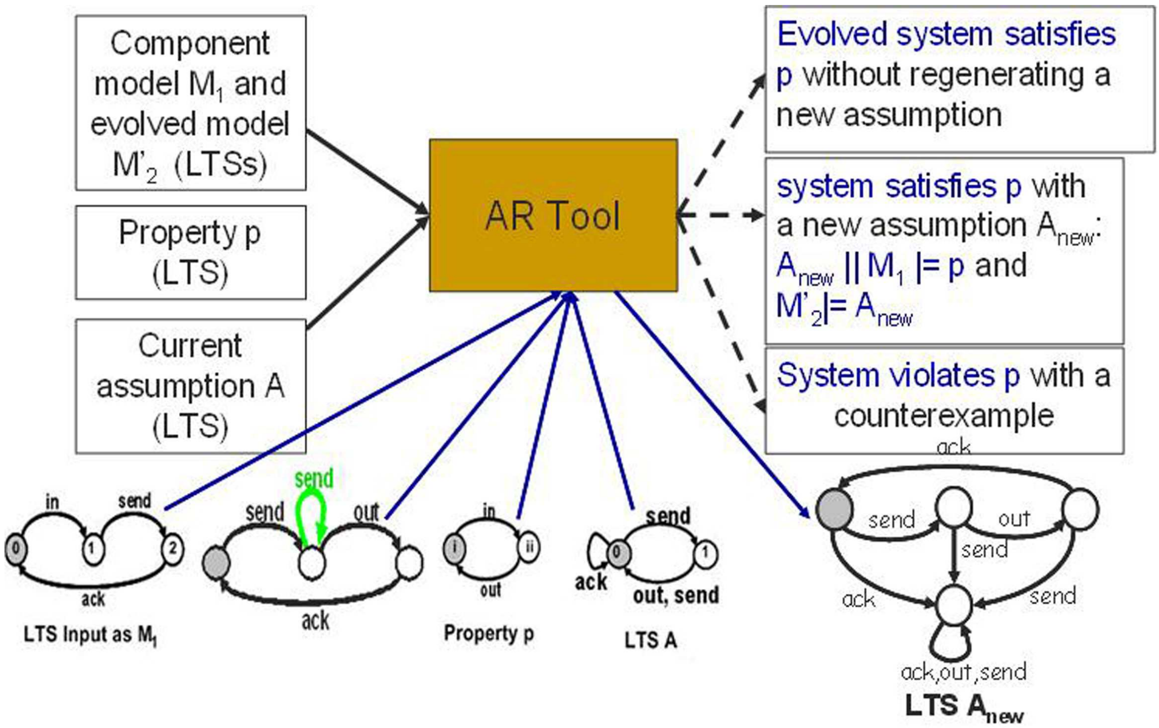



Figure 4.10: The architecture of the AR tool and an example.

In our method, for the first and second evolved systems (I/O ver.1 and I/O ver.2), the current assumptions are not actual assumptions of these systems. In this case, the new assumptions are regenerated with a smaller number of the required membership queries and the generated candidate assumptions which are needed to regenerate the new assumptions. In the third system (I/O ver.3), the current assumption is an actual assumption of this system so the system is verified without regenerating a new assumption. This is a successful example to show the effectiveness of the proposed method.

The implemented tool and the illustrative systems which are used in our experimental results contain at the site [44].

We also use the tool for verifying concurrent systems called LTSA [13] to check correctness of the new assumption $A_{new}(p)$ which is generated by our proposed method. For this purpose, we check that whether $A_{new}(p)$ satisfies the compositional rule (i.e.,

Table 4.1: Experimental results

| **System** | Sys. Size | **Ass. Generation** | | **Ass. Regenaration** | |
|---|---|---|---|---|---|
| | | Queries | Cadidates | Queries | Candidates |
| I/O ver.1 | 18 | 80 | 4 | 56 | 2 |
| I/O ver.2 | 75 | 294 | 6 | 210 | 3 |
| I/O ver.3 | 75 | 294 | 6 | 0 | 0 |

$\langle A_{new}(p) \rangle$ $M_1$ $\langle p \rangle$ and $\langle \text{true} \rangle$ $M_2'$ $\langle A_{new}(p) \rangle$ both hold) by checking the compositional systems $A_{new}(p) \| M_1 \| p_{err}$ and $M_2' \| A_{new}(p)_{err}$ in the LTSA tool. For each compositional system, the LTSA tool returns the same result as our verification result for each evolved system.

Our obtained experimental results imply that the proposed framework can reduce the number of steps required in the model update and the number of the membership queries and the candidate assumptions which are needed to regenerate the new assumptions. In some successful cases where the current assumptions are actual assumptions of the evolved CBS, these CBS are verified in the fastest way without regenerating the new assumptions. This means that the proposed framework can reduce the cost of the conformance testing and modular verification of the evolved CBS.

## 4.5 Discussion

As mentioned in the above section (Section 4.4), the proposed framework presented in Section 4.2.1 is effective for rechecking the evolved CBS in the context of the component evolution. In some successful cases where the current assumptions are actual assumptions of the evolved CBS, these CBS are verified in the fastest way without regenerating the new assumptions. In the others, the framework applies the proposed method for assumption regeneration. The method reuses the entire current assumption to reduce the large number of the membership queries and the candidate assumptions which are needed to regenerate the new assumptions. The experimental results shows that the proposed framework is practical for rechecking the evolved CBS. However, the core of the new assumption regeneration method is based on the framework proposed in [10]. Thus, the new assumptions regenerated by the proposed method are not minimal mentioned in Chapter 3. Moreover, the regenerated assumptions in the method should be minimized because of several advantages mentioned in Section 3.6 of Chapter 3. The minimized assumption regeneration method described in Section 4.3 has been proposed to solve the issue. The improved method for minimized assumption regeneration produces a new minimal assumption of the evolved CBS in the case where the current assumption is too strong. Nonetheless, the computational cost for regenerating the minimal assumption in the improved method is more expensive than the proposed method presented in Subsection 4.2.2.

Another limitation of the proposed framework is about the meaning of the component evolution concept. In the framework, we define a the component evolution concept means that adding only some new behaviors to the old component without losing the old behaviors. We think that adding behaviors to the old component is enough for software component evolution as mentioned in Section 2.4 of Chapter 2. With this approach, we have a simpler and faster assume-guarantee framework to recheck the new CBS by reusing the entire assumption of the old CBS. However, this concept does not hold in some cases where the component evolution means that adding some new behaviors to the old component and removing some old behaviors from the component. In the cases, we can not reuse the entire assumption of the old CBS directly because the component evolution may change the unknown language $U$ of the assumption be learned. A nice idea to dealing with this issue is to combine the proposed framework and the method for verification of evolving CBS via component substitutability analysis proposed in [22, 23]. When the unknown language $U$ is changed to $U'$ by the component evolution, the current assumption $A(p)$ of the old CBS may be invalidated for the new language $U'$. A validated assumption $A(p)$ for $U'$ means that for every trace $s$ of $A$, $s$ exactly belongs to $U'$. If $A(p)$ is invalidated, a validated candidate assumption $A'(p)$ is obtained by revalidating $A(p)$. If $A'(p)$ is too strong to be satisfied by the evolved model, the L* learning algorithm is applied to regenerate a new assumption with the initial assumption as $A'(p)$.

## 4.6 Related Work

Even though the proposed technique in this paper is based on component-based modular model checking, there is a fundamental difference between the conventional modular verification works [37, 38, 39] and our work. Modular verification in the previous works [37, 38, 39] is rather closed. It also is not prepared for future changes. If a component is added to the system, the whole system of many existing components and the new component are required to re-checked altogether. On the contrary, the proposed method verifies global system properties by checking components separately. In the simplest form, it checks whether a component $M_1$ guarantees a property $p$ when the external environment satisfies an assumption $A(p)$, and checks that the remaining components in the system ($M_1$'s environment) indeed satisfy $A(p)$.

The work introduced in this paper is similar to these works in [20, 10, 16, 36, 34]. However, our method differs these methods in [20, 10, 16, 36, 34] in some key points. Firstly, our work presents a faster assume-guarantee method to verify component-based systems in the context of component refinement. There is a strong relationship between two components $C_1$ and $C_2$, where $C_2$ is the refinement of $C_1$. For this reason, the proposed method is efficient to change. On the contrary, component refinement is not mentioned in these works. Secondly, in the proposed method, if the component $C_1$ is refined into a new component $C_2$ and if the formula $\langle \text{true} \rangle\ C_2\ \langle A(p) \rangle$ does not hold, the new assumption $A_{new}(p)$ is re-generated *without regenerating* on a faster way. These works in [20, 10, 16, 36, 34] are viewed from a static perspective, i.e., the component and the external environment do not evolve. If the component changes after adapting some refinements, the assumption-generating method is re-run on the whole refined system, i.e., the component model has to be re-constructed; and the assumption about the environment is then regenerated from that model.

Our work is close to the assume-guarantee verification of evolving software in [21]. However, the approach in [21] tries to reuse previous assumptions by changing from the L* learning algorithm to the dynamic L* algorithm. We share this purpose, but our work sets the previous assumption as initial assumption for the L* algorithm.

Finally, our work relates to many works have been proposed in model checking publish/subscribe systems [40, 41, 42, 43]. The paper in [43] based on the idea of providing a generic, parametric publish/subscribe model-checking framework is proposed. This framework allows for decomposing the problem in two parts: *(1)* a reusable model that captures run-time event management and dispatch, and *(2)* components that are specific to the application being modeled. This work has been extended in [40, 41, 42] by the different ways. In particular, [41] uses architectural patterns as an abstraction to carry on, and reuse, formal reasoning on systems whose configuration can dynamically change. [42] presents a compositional reasoning to verify middleware-based software architecture descriptions. [40] embeds the asynchronous communication mechanisms of publish-subscribe infrastructures within Bogor. Our work and [42] share the compositional reasoning approach but we focus on solving a sub-part in the framework that is assumption regeneration in the context of component refinement.

## 4.7 Summary

We have presented an effective framework for assume-guarantee verification of component-based software in the context of the component evolution at the system design level. The component evolution means that adding only some new behaviors to the old component without losing the old behaviors. In the proposed framework, if a component model is evolved to a new component model, the whole evolved CBS of many existing component models and the evolved model is not required to be rechecked. It only checks whether the evolved component model satisfies the assumption of the old system. If it is, the evolved CBS has been verified. Otherwise, if the assumption is too strong, the proposed method for the new assumption regeneration is required to regenerate a new assumption by reusing the entire old assumption as the previous verification result. Our work does not regenerate the new assumption from scratch. Therefore, the proposed new assumption regeneration method can cut down several steps for regeneration of the new assumption and it also opens for future changes. In order to improve the proposed method, we present a solution for reducing the number of candidate queries which are needed for regenerating the new assumption. We also propose a minimized assumption regeneration method for modular verification of component-based software in the context of the component evolution. This method is an improvement of the minimized assumption generation method presented in Section 3.3 of Chapter 3. We have implemented a tool for the assumption generation method proposed in [10] and our assumption regeneration method. This implementation is used to verify some small evolved concurrent systems to show the effectiveness of the proposed method.

# Chapter 5

# Modular Conformance Testing and Assume-Guarantee Verification for Evolving Component-Based Software

This chapter proposes a framework for modular conformance testing and modular verification of evolving component-based software at the source code level. This framework includes two stages: modular conformance testing for updating inaccurate models of the evolved components and modular verification for evolving component-based software. When a component is evolved after adapting some refinements, the proposed framework focuses on this component and its model in order to update the model and to recheck the whole evolved system. The framework also reuses the previous verification results and the previous models of the evolved components to reduce the number of steps required in the model update and modular verification processes.

## 5.1 Introduction

Component-based development has been recognized as one of the most important technical initiatives in software engineering. However, one of the key issues of component-based software (CBS) is to ensure that those separately specified and implemented components do not conflict with each other when composed - the *component consistency* issue. Currently, the popular solution to dealing with this issue is the verification of CBS via model checking [2]. Nonetheless, a major problem of model checking is the *state space explosion*. Moreover, the CBS verification is very difficult task due to the frequent lack of information about software components that may be provided by third parties without source codes and with incomplete documentations. Even if we have source code and complete documentation, it is very hard to understand them. The best way is to see the software component implementations as black boxes. In this case, obtaining accurate models which accurately describe behaviors of the software components under study is an interesting problem.

Model-based verification techniques generally assume that the ways to obtain a model which describes the behavior of software under study and its correctness are available. It means that this model is available and accurate. However, these assumptions may not always hold in practice due to the modelling errors, bug fixing, etc. Even if the assumptions hold, the model could be invalidated when software is evolved by adding and removing some behaviors because evolving of existing components seems to be an unavoidable task during the software life cycle. Unfortunately, the consequence of this tasks is the whole evolved software must be rechecked. This initiates the study of adaptive model checking (AMC) [12] which necessitates an iterative construction of a model for software by applying a learning algorithm called L* [9, 17]. Nonetheless, the model in AMC describes the behavior of the whole software. In order to recheck the evolved CBS, the *state space explosion* problem may occur when checking large-scale software. In this case, rechecking of the whole evolved CBS is unnecessary. It should be better to focus only on the evolved components and try to reuse previous verification results to verify the new system. Moreover, the AMC approach can not reuse the whole given model because it does not ensure the achievement of an updated model from the inaccurate model because the *software evolution* concept in AMC means adding some new behaviors and removing some existing behaviors. Furthermore, when system is changed, the model is required to update including comparisons of software with the new candidate model via the Vasilevskii-Chow (VC) algorithm [45, 47]. If the model is inaccurate then updating the whole model is not necessary (and very expensive) because the changes often focus on a few existing components with small changes.

This chapter proposes a framework for modular verification of component-based software to deal with the above issues in the context of the component evolution. We propose two methods for this framework: *modular conformance testing* (MCT) for updating inaccurate models of the evolved software components, and *modular verification* for evolving component-based software. In this framework, when a software component is evolved after adapting some refinements, instead of doing conformance testing on the whole system and its model, the proposed MCT only performs conformance testing to compare this component with its model. If the model of the evolved component is inaccurate then it is used as the initial model for the L* learning algorithm in order to update itself. Otherwise, the component and its model are in conformance. The proposed framework then applies the assume-guarantee method to verify the evolved CBS. In this case, the whole evolved CBS of many existing components and the evolved component is not required to be rechecked for its satisfaction of property $p$. The framework focuses only on the model of the evolved component to recheck the evolved software. With regard to effectiveness, the proposed framework can reduce the number of steps required in the model update and the number of the membership queries and the candidate assumptions which are needed to regenerate the new assumptions. In some successful cases where the current assumptions are actual assumptions of the evolved CBS, these CBS are verified in the fastest way without regenerating the new assumptions.

We explain an overview of the proposed framework as follows. Suppose that there is a simple component-based software which contains a base component $C_1$ as a fixed framework, and a component $C_2$ as an extension. The extension $C_2$ is plugged into the framework $C_1$ via some mechanisms. This kind of CBS only allows us to evolve the behavior of the extension component in the context of the component evolution and it is popular in practice. Let $M_1$ and $M_2$ be accurate models of $C_1$ and $C_2$ respectively. It is known that the compositional system $M_1 \| M_2$ satisfies the property $p$ (i.e., $C_1 \| C_2$ satisfies $p$). During the life cycle of this system, the extension $C_2$ is evolved to a new component $C_2'$ by adding some new behaviors to $C_2$. The proposed MCT only performs conformance testing to compare $C_2'$ with $M_2$. If they are not in conformance, $M_2$ is used as the initial model for the L* algorithm to obtain an accurate model $M_2'$ for the evolved component $C_2'$. The new compositional system $M_1 \| M_2'$ then must be rechecked for whether it satisfies the property $p$ or not. For this purpose, the proposed modular verification method only checks that the new model $M_2'$ satisfies an assumption $A(p)$, where $A(p)$ is an assumption between $M_1$ and $M_2$ that is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$. The assumption $A(p)$ is generated by using the L* learning algorithm. In this method, component models, properties, and assumptions are represented by Labeled Transition Systems (LTSs). If $M_2'$ satisfies $A(p)$, then the evolved CBS $C_1 \| C_2'$ satisfies the property $p$. Otherwise, this step returns a counterexample *cex* to witness this fact. The proposed method then performs some analysis to determine whether $p$ is indeed violated in the evolved system $M_1 \| M_2'$ or if $A(p)$ is too strong to be satisfied by $M_2'$. If the assumption $A(p)$ is too strong, a new assumption $A_{new}(p)$ between $M_1$ and $M_2'$ is regenerated. The proposed method regenerates the new assumption $A_{new}(p)$ *without rerunning* on the whole evolved system. We try to reuse the results of the previous verification (i.e., the generated assumptions) in order to reduce the number of steps of the new assumption regeneration process.

The rest of the chapter is organized as follows. We first describes a modular conformance testing method and its application for updating the inaccurate component models in Section 5.2. Section 5.3 is about an assumption regeneration method to recheck evolved CBS. Section 5.4 presents an integrated framework for modular verification of evolving CBS. An implementation and some experimental results also are showed in this section. Section 5.6 presents related works. Finally, we summarize the chapter in Section 5.7.

## 5.2 Modular Conformance Testing for Evolving Component

In this section, we propose a conformance testing method called *modular conformance testing* (MCT) to reduce the cost of the conformance testing process in the context of the software component evolution. In this method, when a software component is evolved after adding some new behaviors, instead of doing conformance testing on the whole system and its model [12], the proposed MCT only performs conformance testing to compare this component with its model. If the model of the evolved component is inaccurate then it is used as the initial model for the L* learning algorithm in order to update itself. Otherwise, the component and its model are in conformance.

Consider a simple case where a system contains two components $C_1$ and $C_2$. We can see these components as black boxes due to the frequent lack of information about software components that are provided by third parties without source codes and with incomplete documentations. For each component $C_i$ (i=1,2), we obtain an accurate model $M_i$ by applying the L* learning algorithm or some modelling techniques. The model $M_i$ is an accurate model of the component $C_i$ (i=1,2), denoted $M_i \models_T C_i$, if and only if the accurate model definition defined in Chapter 2 is satisfied. $C_2$ then is refined into a new component $C_2'$ by adding some behaviors to the old component $C_2$. In this case, the MCT is applied to check whether the current model $M_2$ is an accurate model of the evolved component $C_2'$. Let $C_2 = (\Sigma_{C_2}, T_{C_2})$ and $C_2' = (\Sigma_{C_2'}, T_{C_2'})$ where the strings in $T_{C_2}$ and $T_{C_2'}$ reflect the allowed executions of $C_2$ and $C_2'$ respectively.

In order to check conformance between $C_2'$ and its current model $M_2$, instead of doing conformance testing on all strings in $T_{C_2'}$, MCT only checks on all strings in $v \in (T_{C_2'} \setminus T_{C_2})$. MCT does not check the strings in $T_{C_2}$ due to $M_2 \models_T C_2$. This means that for every string/trace $v \in (T_{C_2'} \setminus T_{C_2})$ (after applying a **Reset**), if $v \in L(M_2)$ then $C_2'$ and $M_2$ are in conformance and MCT terminates. Otherwise, the inaccurate model $M_2$ must be updated by using the L* learning algorithm with the help of the counterexample $v$. The L* performs experiments on the evolved component $C_2'$ and produces a minimized finite automaton representing behavior of this component. This learning algorithm is an iterative process illustrated in Figure 5.1. At the initial step, we use the current model $M_2$ as the initial model. At each iteration $i$, a candidate model $M_{2i}$ is produced based on some knowledge about the component $C_2'$ and the results of the previous iteration. The MCT is then applied to check conformance between $C_2'$ and its candidate model $M_{2i}$. If they do conform, MCT terminates. Otherwise, a counterexample is provided by MCT to generate the next candidate model $M_{2(i+1)}$ and the entire process must be repeated. The MCT's performance always terminates because $(T_{C_2'} \setminus T_{C_2})$ is a finite set of the test strings.

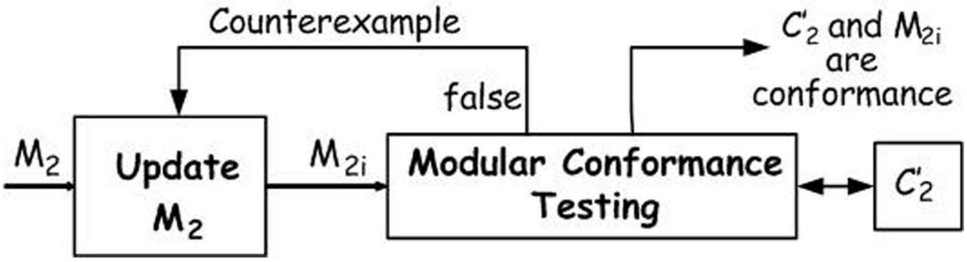



Figure 5.1: The modular conformance testing framework.

**Correctness of MCT.** We use $M_1 \| M_2 \models_T C_1 \| C_2$ to denote that the compositional model $M_1 \| M_2$ is an accurate model of the compositional system $C_1 \| C_2$. Correctness of the *modular conformance testing* is guaranteed by the following theorem.

**Theorem 4** *Given two software components $C_1 = (\Sigma_{C_1}, T_{C_1})$ and $C_2 = (\Sigma_{C_2}, T_{C_2})$. If $M_1$ and $M_2$ are accurate models of $C_1$ and $C_2$ respectively (i.e., $M_1 \models_{\mathrm{T}} C_1$ and $M_2 \models_{\mathrm{T}} C_2$) then the compositional model $M_1 \| M_2$ is an accurate model of the compositional system $C_1 \| C_2$ (i.e., $M_1 \| M_2 \models_{\mathrm{T}} C_1 \| C_2$).*

**Proof** For every trace $v \in (\Sigma_{C_1} \cup \Sigma_{C_2})^*$, if $v$ is a successful experiment of $C_1 \| C_2$ then $v \uparrow \Sigma_{C_1}$ is a successful experiment of $C_1$ and $v \uparrow \Sigma_{C_2}$ is a successful experiment of $C_2$. Because of $M_1 \models_T C_1$ and $M_2 \models_T C_2$, by checking the accurate model definition defined in Chapter 2, it follows that $v \uparrow \Sigma_{C_1} \in L(M_1)$ and $v \uparrow \Sigma_{C_2} \in L(M_2)$ . This means that $v$ is a trace of the compositional model $M_1 \| M_2$ (i.e., $v \in L(M_1 \| M_2)$). ∎

## 5.3 Modular Verification of Evolving CBS

Suppose that there are two components including a fixed based architecture $C_1$ as a framework and an extension $C_2$. Let $M_1$ and $M_2$ be accurate models of $C_1$ and $C_2$ respectively. We know that the property $p$ holds in the compositional system $M_1 \| M_2$. $C_2$ is then evolved to a new component $C_2'$ by adding some new behaviors to the old component $C_2$. Let $M_2'$ be the updated model of the $M_2$, obtained by applying the proposed MCT (i.e., $M_2' \models_T C_2'$). In order to recheck the evolved compositional system $M_1 \| M_2'$, the proposed method only checks the formula $\langle \text{true} \rangle\ M_2'\ \langle A(p) \rangle$, where $A(p)$ is an assumption between $M_1$ and $M_2$ that is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$. If this formula holds, the evolved system satisfies the property $p$. Otherwise, if the assumption $A(p)$ is too strong, a new assumption $A_{new}(p)$ is regenerated by applying the new assumption regeneration method proposed in Chapter 4 with the initial assumption $A(p)$.

## 5.4 Framework for MCT and Modular Verification of Evolving CBS

This section proposes an integrated framework for modular verification of component-based software in the context of the component evolution. Some small concurrent systems are evolved and applied by our implemented tool to show the effectiveness of the proposed framework.

### 5.4.1 Proposed Framework

We integrate the proposed modular conformance testing and the modular verification into a framework for verifying component-based software in the context of component evolution. This framework is illustrated in Figure 5.2. It consists of the following steps.

1. Updating the inaccurate model $M_2$ of the evolved component $C_2'$ by using the modular conformance testing method with the initial model $M_2$. This step returns the updated model $M_2'$ of $C_2'$.

2. Checking whether the evolved system $M_1 \| M_2'$ satisfies the property $p$ by applying the modular verification method. This step only focuses on checking the updated model $M_2'$ of the evolved component $C_2'$. If $M_2'$ satisfies the assumption $A(p)$, the evolved compositional system $M_1 \| M_2'$ satisfies $p$. Otherwise, it returns a counterexample *cex*.

3. Further analysis is required to identify that whether $p$ is indeed violated in $M_1 \| M_2'$ or $A(p)$ is too strong to be satisfied by $M_2'$. Such analysis is based on the counterexample *cex* returned by the step 2. This step must check that the counterexample *cex* belongs to the unknown language $U = L(A_W)$. If it does not, the property $p$ does not hold in the system $M_1 \| M_2'$. Otherwise, $A(p)$ is too strong.

4. The assumption regeneration method is applied to generate a new assumption $A_{new}(p)$ with the help of the counterexample *cex* returned by the step 2. The generated assumption $A_{new}(p)$ is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2'$.

Although the framework illustrated in Figure 5.2 considers the simple case where the software only consists of two components $C_1$ and $C_2$, we can generalize it for a larger system containing n-components $C_1, C_2, ..., C_n$ (n$\geq$2). The framework for the larger system consists of the similarly steps as described above because we only focus on the evolved component. However, the assume-guarantee verification for larger systems will be more expensive than simple cases.

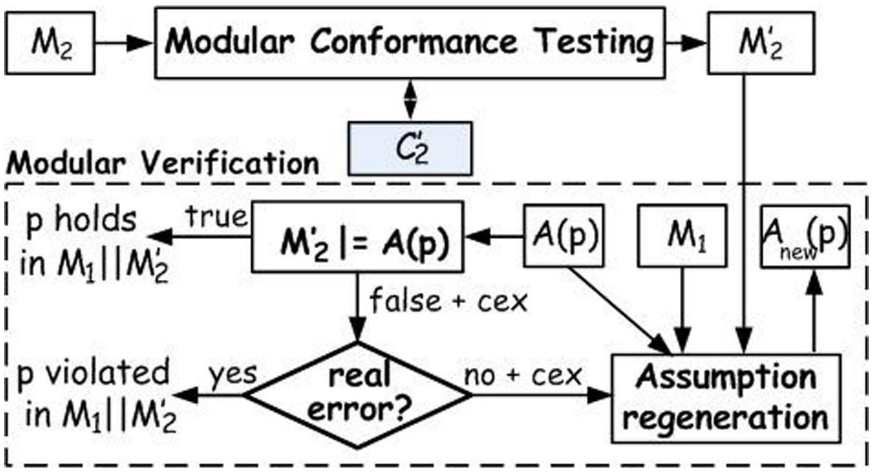



Figure 5.2: The proposed framework for MCT and modular verification of evolving CBS.

### 5.4.2 An Example

Figure 5.3 describes an illustrative concurrent system which contains the accurate model $M_1$ of a base component $C_1$ and the accurate model $M_2$ of a extension component $C_2$ . The model $M_1$ is plugged into the model $M_2$ via the parallel composition operator defined in Section 2. In this system, the LTS of $M_1$ is the Input LTS, and the LTS of $M_2$ is the Output LTS. This concurrent system means that the *Input* LTS receives an input when the action *in* occurs, and then sends it to the Output LTS with action *send*. After some data is sent to it, the Output LTS produces output using the action *out* and acknowledges that it has finished, by using the action *ack*. At this point, both LTSs return to their initial states so the process can be repeated. The property $p$ means that the *in* action has to occur before the *out* action. The assumption $A(p)$ is generated by using the framework illustrated in Figure 3.5 in Chapter 3 that is strong enough for $M_1$ to satisfy $p$ but weak enough to be discharged by $M_2$.

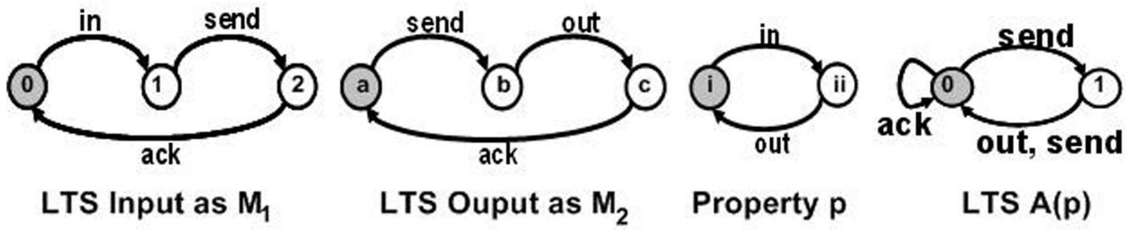



Figure 5.3: Component models, order property and assumption $A(p)$ of the illustrative system.

The extension component $C_2$ of the model $M_2$ is then evolved to a new component $C'_2$ by adding a new behavior which allows multiple *send* actions to occur before producing *out*. In this case, the current model $M_2$ is inaccurate. For example, the string ⟨*send*, *send*, *out*⟩ is a successful experiment on $C'_2$ but it is not a trace of $M_2$. The proposed

MCT is applied to update $M_2$. The updated model $M'_2$ produced by MCT is illustrated in Figure 5.4. To recheck the new compositional system $M_1 \| M'_2$, the proposed method only checks the formula $\langle \text{true} \rangle\ M'_2\ \langle A(p) \rangle$. In this case, this formula does not hold and a counterexample *cex* = *send send out* is returned to witness this fact. The method then performs some analysis to determine that whether the evolved system violates the property $p$ or $A(p)$ is too strong. The result is that $A(p)$ is too strong to be satisfied by $M'_2$. A new assumption $A_{new}(p)$ must be regenerated. For this purpose, the method proposed in this chapter reuses the old assumption $A(p)$ as the initial assumption and applies the improved L* learning algorithm showed in Algorithm 3 to regenerate the new assumption $A_{new}(p)$ illustrated in Figure 5.4.

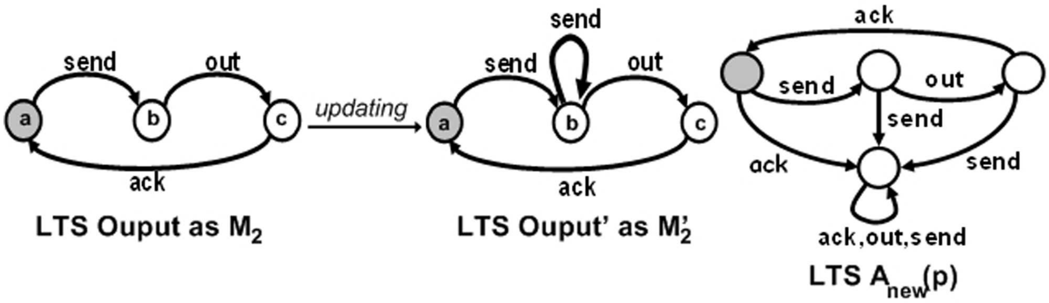



Figure 5.4: The updated model and the regenerated assumption $A_{new}(p)$.

## 5.5 Discussion

## 5.6 Related Work

There are many works that have been recently proposed in verification of evolving software, by several authors. Focusing only on the most recent and closest ones we can refer to [46, 12], to [21, 22, 23], to [28], to [10, 16], and [11, 15].

Peled et al. proposes an approach called black box checking [46] as a way to directly verify a system when its model is not given but a way of conducting experiments is provided. A related idea, called adaptive model checking (AMC) (AMC) [12], allows using an inaccurate and updated model to do the verification, while refining it during verification process. Our work combines the idea of AMC and modular model checking in order to deal with the *state space explosion* problem and to reduce the expensiveness of the conformance testing process.

An approach for verification of evolving software was suggested by Chaki et al. [21, 22, 23]. This work focusses on component substitutability directly from the verification point of view. The purpose of this work is to provide an effective verification procedure that decides whether a component can be replaced with a new one without violation. The approach also reuses previous assumptions by using the dynamic L* algorithm. We share the motivation with this work, but the concept about *component evolution* in our work means adding only some new behaviors to the old component. In our opinion, adding is enough for software evolution. By this definition, our verification method is simpler than the method proposed in [21, 22, 23]. Moreover, this work uses abstraction technique to obtain a new model of the upgraded component. Regenerating the new model is not necessary because the component changes are often small. Our work reuses the previous model to update the new one by applying the L* learning algorithm.

The assume-guarantee verification for component-based systems was proposed in [10, 16]. However, these works assume that the models which describe the behaviors of the software components are available and accurate. Furthermore, our approach differs from this approach in some key points. Firstly, our work presents an assume-guarantee method to verify component-based systems in the context of the component evolution. There is a strong relationship between two models $M_2$ and $M_2'$, where $M_2'$ is the evolution of $M_2$. For this reason our proposed approach is efficient. On the contrary, the component evolution is not mentioned in these works. Secondly, in our work, if the component $C_2$ is refined into a new component $C_2'$ and if the formula $\langle \text{true} \rangle\ M_2'\ \langle A(p) \rangle$ does not hold, the new assumption $A_{new}(p)$ is regenerated by starting from $A(p)$. These works in [10, 16] are viewed from a static perspective, i.e., the component and the external environment do not evolve. If the component changes after adapting some refinements, the assumption-generating method is rerun on the whole refined system, i.e., the component model has to be reconstructed; and the assumption about the environment is then regenerated from that model.

Finally, the different approaches about assume-guarantee verification methods for component-based systems were proposed in [28, 11, 15]. These papers assume the availability and correctness of models that describe the behaviors of the software components. Furthermore, our work differs in the concept of software component evolution. In our framework, component evolution means only adding some behaviors to the old component whereas the concept in [11, 15] means adding (or plugging) a new component (extension) to the base component via compatible interface states. These works also assume the availability and correctness of the new model of the evolved component. In practice, checking correctness of the new model and updating the inaccurate model are the difficult tasks.

## 5.7 Summary

We have presented a framework for modular verification of component-based software in the context of component evolution. In this framework, if a component is evolved to a new component by only adding some new behaviors, the whole evolved system of many existing components and the evolved component are not required to be rechecked. We only focus on the evolved component and its model to recheck the evolved CBS. We think that adding behaviors to the old component is enough for software component evolution. With this approach, we have a simpler assume-guarantee method to recheck the new CBS. Moreover, when a component is evolved, its model may be inaccurate. We propose the *modular conformance testing* method to check conformance between this model and the actual evolved component. If they do not conform, this model is updated by using the L* learning algorithm with the initial model as itself. In our work, the models describe the behaviors of the corresponding software components. Therefore, the proposed framework can deal with the *state space explosion* problem in model checking and reduce the cost of the conformance testing when checking large-scale software. We also have implemented a tool for the assumption generation method proposed in [10] and our assumption regeneration method. This implementation is used to verify some small evolved concurrent systems to show the effectiveness of the proposed method.

# Chapter 6

# Conclusion

## 6.1 Summary of the Dissertation

The research in this dissertation focuses on assume-guarantee verification of evolving component-based software in the context of the component evolution at the software design level and implementation level. In the research, the component evolution means *adding only some new behaviors to the old component without losing the old behaviors*. We think that adding behaviors to the old component is enough for software component evolution. With this approach, we have a simpler assume-guarantee method to recheck the evolved component-based software. The key idea of our research is to reuse the previous verification results and the previous models of the evolved components in order to reduce the number of steps required in the model update and modular verification processes.

The first and the second chapters of the dissertation are about the context and the background of this research. The main contributions of the research are in Chapters 3, 4, and 5.

In Chapter 3, we propose a method for generating minimal assumptions for the assume-guarantee verification of component-based software. The method is an improvement of the described L*-based assumption generation method. The key idea of the proposed method is finding the minimal assumptions in the search spaces of the candidate assumptions. These assumptions are seen as the environments needed for the components to satisfy a property and for the rest of the system to be satisfied. The minimal assumptions generated by the proposed method can be used to recheck the whole system at much lower computational cost. We also present some improvements of the method in order to reduce the computational cost for generating the minimal assumptions. We have implemented a tool for generating the minimal assumptions. Experimental results are also presented and discussed.

Chapter 4 proposes an effective framework for assume-guarantee verification of component-based software in the context of the component evolution at the design level. In this framework, if a component model is changed after adapting some refinements, the whole component-based software (CBS) of many existing component models and the evolved component model is not required to be rechecked. The method only checks whether the evolve model satisfies the assumption of the old system. If it is, the evolved CBS still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, a new assumption is regenerated. We propose two methods for the new assumption regeneration: assumption regeneration and minimized assumption regeneration. The methods reuse the current assumption as the previous verification result to regenerate the new assumption at much lower computational cost. An implementation and experimental results are presented.

Chapter 5 proposes a framework for modular verification of evolving component-based software at the source code level. This framework includes two stages: modular conformance testing for updating inaccurate models of the evolved components and modular verification for evolving component-based software. When a component is evolved after adapting some refinements, the proposed framework focuses on this component and its model in order to update the model and to recheck the whole evolved system. The framework also reuses the previous verification results and the previous models of the evolved components to reduce the number of steps required in the model update and modular verification processes.

## 6.2 Future Directions

In this dissertation, we focus on the simple component-based software where the software only consists of two components. Therefore, one of our future works is to generalize the proposed method in Chapter 3, the proposed effective framework in Chapter 4, and the proposed framework in Chapter 5 for the larger component-based software, where CBS contain more than two components. We are also improving the method and the frameworks, and applying some CBS, where their sizes are larger than the sizes of the CBS which are used in our experiments in order to show their practical usefulness.

In Chapter 3 of the dissertation, we have proposed a method for generating minimal assumptions. However, the computational cost for generating the minimal assumptions is very expensive. Although we have presented some improvements of the method in order to reduce the computational cost, the effectiveness of the improvements should be evaluated by applying some larger illustrative system in our experiment.

As mentioned in Chapter 4, in the case where a new assumption is required to regenerate for rechecking the evolved CBS, we can apply one of the two proposed methods: assumption regeneration and minimized assumption regeneration. In the former, its core is based on the framework proposed in [10]. Thus it should be improved to obtain a more effective method for assumption regeneration by reducing the number of the membership queries and the candidate assumptions which are needed to regenerate the new assumptions. One of solutions we intend to use is applying the approach about optimized L*-based assume-guarantee reasoning proposed by Chaki et al. [24]. The core of the later is based on the method proposed in Chapter 3. Although we reuse the current assumption as an approach for reducing the search space of of the observation tables, the computational cost for regenerating the minimal assumptions still is expensive. We are investigating to apply the improvements presented in Chapter 3 for this method in order to reduce the computational cost. We also are investigating to implement a tool supporting for the minimized assumption regeneration method.

In Chapter 5, we have proposed a method for modular conformance testing for updating inaccurate models of the evolved components. However, we have not implemented the tool yet. The most difficult part in building this tool is checking the conformance between a component and its model via the Vasilevskii-Chow (VC) algorithm [45, 47].

# Bibliography

[1] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, P. McKenzie: "Systems and Software Verification: Model-Checking Techniques and Tools", Springer-Verlag (2001).

[2] E. M. Clarke, O. Grumberg, and D. Peled: "Model Checking", The MIT Press (1999).

[3] G. T. Heineman and W. T. Councill, Eds: "Component-Based Software Engineering: Putting the Pieces Together", Addison-Wesley Longman Publishing Co., Inc. (2001).

[4] J. Hopkins: "Component Primer", Communications of the ACM, ACM Press, vol. 43, no. 10, pp. 27–30 (2000).

[5] H. Hungar, O. Niese, and B. Steffen: "Domain-Specific Optimizations in Automata Learning", Proc. of the 15th Computer Aided Verification (CAV), LNCS 2725, Springer Verlag, pp.315–327 (July 2003).

[6] C. Szyperski: "Component Software: Beyond Object-Oriented Programming", ACM Press/Addison-Wesley Publishing Co. (1998).

[7] C. B. Jones: "Tentative Steps Toward a Development Method for Interfering Programs", ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 5, no. 4, pp. 596–619 (Oct. 1983).

[8] A. Pnueli: "In Transition from Global to Modular Temporal Reasoning about Programs", In Logics and Models of Concurrent Systems, K. R.Apt, Ed. Nato Asi Series F: Computer And Systems Sciences, Springer-Verlag New York, vol. 13, pp. 123–144 (1985).

[9] D. Angluin: "Learning Regular Sets from Queries and Counterexamples", Information and Computation, vol. 75, no. 2, pp. 87-106 (Nov. 1987).

[10] J.M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu: "Learning Assumptions for Compositional Verification", Proc. of 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 331–346 (Apr. 2003).

[11] K. Fisler and S. Krishnamurthi: "Modular Verification of Collaboration-Based Software Designs", Proc. of 8th European Software Engineering Conference, Austria, pp. 152–163 (Sept. 2001).

[12] A. Groce, D. Peled, and M. Yannakakis: "Adaptive Model Checking", Logic Journal of the IGPL, vol. 14, no. 5, pp. 729–744 (Oct. 2006).

[13] J. Magee and J. Kramer: "Concurrency: State Models & Java Programs", John Wiley & Sons (1999).

[14] N. T. Thang and T. Katayama: "Open Incremental Model Checking," Proc. of 3rd Microsoft Research – Specification and Verification of Component-Based Systems Workshop (SAVCBS), ACM FSE, pp. 122–125 (Nov. 2004).

[15] N. T. Thang and T. Katayama: "Specification and Verification of Inter-Component Constraints in CTL", Proc. of 4th Microsoft Research – Specification and Verification of Component-Based Systems Workshop (SAVCBS), Portugal, pp. 122–125 (Sept. 2005).

[16] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer: "Assumption Generation for Software Component Verification", Proc. of 17th IEEE Int. Conf. on Automated Software Engineering, Edinburgh, UK, pp. 3–12 (Sept. 2002).

[17] R. L. Rivest and R. E. Schapire: "Inference of Finite Automata using Homing Sequences", Information and Computation, vol. 103, no. 2, pp. 299-347 (Apr. 1993).

[18] R. Bird and P. Wadler: "Introduction to Functional Programming", Prentice Hall International Series in Computer Science (1988).

[19] R. Bird: "Introduction to Functional Programming using Haskell", Prentice Hall (1988).

[20] C. Blundell, D. Giannakopoulou, and C. S. Pasareanu: "Assume-Guarantee Testing", Proc. of 4th Microsoft Research – Specification and Verification of Component-Based Systems Workshop (SAVCBS), pp. 7–14 (Sept. 2005).

[21] S. Chaki, E. Clarke, N. Sharygina, N. Sinha: "Verification of Evolving Software and Dynamic Component Substitutability Analysis", Technical Report: CMU/SEI-2005-TR-008, (2005).

[22] S. Chaki, E. Clarke, N. Sharygina, and N. Sinha: "Verification of Evolving Software", Proc. of 3rd Microsoft Research – Specification and Verification of Component-Based Systems Workshop (SAVCBS), pp. 55–61 (Nov. 2004).

[23] S. Chaki, E. Clarke, N. Sharygina, N. Sinha: “Verification of Evolving Software via Component Substitutability Analysis”, Formal Methods in System Design, vol. 32, no 3, pp. 235–266 (Jun. 2008).

[24] S. Chaki, O. Strichman: “Three Optimizations for Assume-Guarantee Reasoning with L*”, Formal Methods in System Design, vol. 32, no. 3, pp. 267–284 (Jun. 2008).

[25] S. Chaki, E. Clarke, D. Giannakopoulou, C. S. Pasareanu: “Abstraction and Assume-Guarantee Reasoning for Automated Software Verification”, Technical Report 05.02, Research Institute for Advanced Computer Science (RIACS), Mountain View, CA. (2004).

[26] J. Hickey: “Introduction to Objective Caml”, Cambridge University Press (2008).

[27] P. N. Hung and T. Katayama: “Modular Conformance Testing and Assume-Guarantee Verification for Evolving Component-Based Software”, Proc. of 15th Asia-Pacific Software Engineering Conf. (APSEC), IEEE Computer Society, pp. 479–486 (Dec. 2008).

[28] P. N. Hung, N. T. Thang, and T. Katayama: “An Assume-Guarantee Method for Modular Verification of Evolving Component-Based Software”, Proc. of 6th WADS in conjunction with 37th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), pp. 160–165 (Jun. 2007).

[29] A. Nerode: “Linear Automaton Transformations”, Proc. of the American Mathematical Society, no. 9, pp. 541–544 (1958).

[30] French National Institute for Research in Computer Science and Control (INRIA): “Objective Caml”, http://caml.inria.fr/ocaml/index.en.html (2004).

[31] OCaml Standard Library: http://caml.inria.fr/pub/docs/manual-ocaml/libref/

[32] A Minimized Assumption Generation Tool for Modular Verification of Component-Based Software, http://www.jaist.ac.jp/ s0620204/MAGTool/ (Apr. 2009).

[33] John H. .Reppy: “Concurrent Programming in ML”, Cambridge University Press (1999).

[34] M. Gheorghiu Bobaru, C. S. Pasareanu, and D. Giannakopoulou: “Automated Assume-Guarantee Reasoning by Abstraction Refinement”, Proc. of the 20th International Conference on Computer Aided Verification, Lecture Notes In Computer Science, Springer-Verlag, vol. 5123, pp. 135–148 (July 2008).

[35] D. Giannakopoulou, C. S. Pasareanu: "Learning-Based Assume-Guarantee Verification (Tool Paper)", Lecture Notes In Computer Science, Springer-Verlag, vol. 3639, pp. 282–287 (2005).

[36] D. Giannakopoulou, C. S. Pasareanu, J. Cobleigh: "Assume-Guarantee Verification of Source Code with Design-Level Assumptions", Proc. of the 26th International Conference on Software Engineering, pp. 211–220 (May 2004).

[37] O. Kupferman and M. Y. Vardi: "Modular Model Checking", In Revised Lectures from the International Symposium on Compositionality: The Significant Difference, vol. 1536 of Lecture Notes in Computer Science, Springer-Verlag, pp. 381–401 (Sept. 1997).

[38] K. Laster, O. Grumberg: "Modular Model Checking of Software", Proc. of the 4th Conference on Tools and Algorithms for the Constructions and Analysis of Systems (TACAS), pp. 20–35 (Apr. 1998).

[39] C. S. Pasareanu, M. B. Dwyer, and M. Huth: "Assume-Guarantee Model Checking of Software: A Comparative Case Study", Proc. of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, vol. 1680 of Lecture Notes of Computer Science, Springer-Verlag, pp. 168–183 (Sept. 1999).

[40] L. Baresi, C. Ghezzi, and L. Mottola,: "On Accurate Automatic Verification of Publish-Subscribe Architectures", Proc. of the 29th International Conference on Software Engineering, Minneapolis (MN, USA), pp. 199–208 (May 2007).

[41] M. Caporuscio, P. Inverardi, and P. Pelliccione: "Formal Analysis of Architectural Patterns", Proc. of 1st European Workshop on Software Architecture (EWSA), St. Andrews, Scotland, UK, pp. 10–24 (May 2004).

[42] M. Caporuscio, P. Inverardi, P. Pelliccione: "Compositional Verification of Middleware-Based Software Architecture Descriptions", Proc. of the 26th International Conference on Software Engineering, pp. 221–230 (May 2004).

[43] D. Garlan, S. Khersonsky, J. Kim: "Model Checking Publish-Subscribe Systems", Proc. of the 10th International SPIN Workshop on Model Checking of Software (SPIN'03), pp. 166–180, (May 2003).

[44] An assumption regeneration tool for modular verification of evolving component-based software, http://www.jaist.ac.jp/ s0620204/RegenerationTool/ (March 2009).

[45] T. S. Chow: "Testing Software Design Modeled by Finite-State Machines", IEEE Transactions on Software Engineering, vol. 4, no. 3, pp. 178–187 (May 1978).

[46] D. Peled, M. Vardi, and M. Yannakakis: "Black Box Checking", Proc. of FORTE/PSTV, Beijing, China, pp. 225–240 (Oct. 1999).

[47] M. P. Vasilevskii: "Failure Diagnosis of Automata", Kibernetika, no. 4, pp. 98–108 (Aug. 1973).

# Publications

[1] P. N. Hung, T. Aoki and T. Katayama: "Modular Conformance Testing and Assume-Guarantee Verification for Evolving Component-Based Software", Conditional accepted by IEICE Trans. Fundamentals, Special Issue on Theory of Concurrent Systems and Its Applications (2009).

[2] P. N. Hung, T. Aoki and T. Katayama: "A Minimized Assumption Generation Method for Component-Based Software Verification", submitted to the 6th International Colloquium on Theoretical Aspects of Computing, Kuala LumPur, Malaysia (Aug. 2009).

[3] P. N. Hung and T. Katayama: "Modular Conformance Testing and Assume-Guarantee Verification for Evolving Component-Based Software", In the 15th Asia-Pacific Software Engineering Conference (APSEC), IEEE Computer Society, pp. 479–486, Beijing, China (Dec. 2008).

[4] P. N. Hung, N. T. Thang and T. Katayama: "An Assume-Guarantee Method for Modular Verification of Evolving Component-Based Software", In Supplemental Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007) – Workshop on Architecting Dependable Systems (WADS 2007), pp. 160–165, Edinburgh, Scotland, UK (Jun. 2007).

[5] P. N. Hung and T. Katayama: "Adaptive Modular Model Checking for Evolving Component-Based Software", Japan-Vietnam Workshop on Software Engineering (JVSE) – Verifiable and Evolvable e-Society, pp. 29–36, Hanoi, Vietnam (Sept. 2007).

[6] P. N. Hung, N. T. Thang and T. Katayama: "An Assumption Regeneration Approach for Component-Based Software Verification", Japan-Vietnam Workshop on Software Engineering (JVSE) – Verifiable and Evolvable e-Society, Hanoi, Vietnam (Aug. 2006).