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: Professor Takuya Katayama
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 are performed with respect to its model which exactly describes the behavior of the system. Thus, 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. In addition, evolving of the existing components of CBS is a daily and unavoidable activity during the software life cycle. Therefore, even if the assumptions hold, the model could be invalidated when the software is evolved by adding or removing some behaviors. Unfortunately, the consequence of these tasks is the whole evolved software must be rechecked. The purpose of this research is to provide an effective approach for modular verification of evolving component-based software systematically 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 design

level. In this framework, if the model of a component is evolved, the whole component-based software of many models of the existing components and the evolved model of the evolved component is not required to be rechecked. The framework only checks whether the evolved model satisfies the assumption of the system before evolving. If it does, 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 assume-guarantee verification of component-based software in the context of component evolution at source code level. This framework includes two stages: modular conformance testing for updating inaccurate models of the evolved components and assume-guarantee verification for evolving component-based software. When a component is evolved, 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 assume-guarantee verification processes.

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

Acknowledgments

First and foremost, I would like to express my sincere gratitude and appreciation to my supervisor, Professor Takuya Katayama, for his constant encouragement and kind guidance during the whole period of my doctoral research and for giving the opportunity that allows me to study in his Laboratory. He introduced me to the fascinating field of modular verification of evolving software and gave me the opportunity to study on several interesting problems in this field.

I would like also to express my sincere thanks to my second supervisor, Associate Professor Toshiaki Aoki, for his encouragement and helpful discussions. Since my first presentation in Aoki Laboratory, I have received many valuable comments from him.

I wish to say grateful thanks to Professor Kokichi Futatsuki, my subtheme supervisor. Although the topic for the subtheme research is very different from that of the main theme, discussing with him helps me improve the way of undertaking the main theme.

I am specially grateful to Associate Professor Xavier Defago for his encouragement and advices not only for my research but also for my daily life.

I wish to continue my gratitude to Professor Tomoji Kishi, Professor Mizuhito Ogawa form JAIST, and Professor Shin Nakajima from National Institute of Informatics for gladly agreeing to serve as members of my dissertation committee and for giving valuable advices to improve my research.

I am deeply indebted to the 21^{st} Century CoE Program "Verifiable and Evolvable e-Society" at JAIST for granting funds for my research.

On this occasion, I wish to thank all members of Katayama Laboratory, specially Mr. Tanizaki Hiroaki and Mrs Miyuki Sakurai, for giving me a lot of kind helps in research and daily life since the fist day I came to Japan.

I would like to take this chance to thank all Vietnamese people in JAIST, specially Dr. Vo Dinh Hieu, for their helps in my daily life.

I wish to express my gratitude to Dr. Nguyen Viet Ha (Vice-Dean of Faculty of Information Technology, College of Technology, Vietnam National University), Associate Professor Do Duc Giao (Faculty of Information Technology, College of Technology, Vietnam National University), and Dr. Nguyen Truong Thang (Institute of Information Technology), for their encouragement, helps and suggestions.

I would like to thank to my parents, sisters, and brothers. They have been with me and have supported me since the day I was born.

Last but not the least, I express my deepest gratitude and great thanks go to my wife Dinh Thi Minh Huyen and two daughters Pham Thi Ngoc Ha and Pham Thi Nhat Ha for their love and encouragement, specially for their tolerance and understanding that without them my research would not be completed.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Component-based development has been recognized as a promising approach for building software systems 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 deal 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 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. The assume-guarantee verification proposed in [7, 8, 10, 16] is a powerful method to deal with the problem. This method is also fitted

to CBS. The method decomposes 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 fixed 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 a daily and unavoidable activity during the software life cycle. Moreover, software evolution often occurs at any time in any phases of the software development process. As a results, it is one of the major characteristics of software. Unfortunately, the consequence of the tasks is the whole evolved software must be rechecked. Furthermore, 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 consider the software component implementations as black boxes. In this case, obtaining accurate models which exactly describe behaviors of the software components under checking 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, 47]. In those works, a learning algorithm called L* [9, 17] 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. Nonetheless, 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 of large-scale software. Rechecking of 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 approach for modular verification of component-based software systematically 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, 40], 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. Furthermore, when a component is evolved 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 are of primary importance. Consequently, our first aim is to optimize the method proposed in [10] in order to generate minimal assumptions for the assume-guarantee verification of CBS.

Consider a popular architecture of CBS 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 a daily and unavoidable activity 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 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 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 the L* learning algorithm [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 of large-scale software systems. 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, AMC is an iterative process for verifying the system including learning the candidate models of the system and model checking each learned candidate model. In this process, many test strings are tested repeatedly via a conformance testing algorithm named Vasilevskii-Chow (VC) [46, 49] and model checking is applied many times as the number of the learned candidate model. As a result, this makes the computational cost for verifying the system to be high. In our opinion, the model learning and model checking should be separated into two independent processes. The model learning process first generates an accurate model of the system. The model is seen as the input of the model checking process for verifying the system. With this approach, the model checking is applied only once. Thus, the computational cost for verifying the system can be reduced. Furthermore, the AMC approach cannot reuse the whole given model because it does not ensure the achievement of an updated model from the inaccurate model because the concept of software evolution in AMC means adding some new behaviors and removing some existing behaviors. Moreover, when system is changed, the model is required to update including comparisons of software with the new candidate model via the VC algorithm. 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 systematically in order to solve the above issues in the context of the component evolution.

1.3 Contributions

This dissertation has three main contributions as follows.

The first contribution of the research is to propose a minimal assumption generation method for assume-guarantee verification of CBS. 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 that satisfies the compositional rules. 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. With regard to the effectiveness, the proposed method can generate the minimal assumptions which have the minimal sizes and smaller numbers of transitions than the assumptions generated by the L*-based assumption generation 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 second contribution is to propose an effective framework for assume-guarantee verification of CBS in the context of the component evolution at design level. In the framework, the component evolution means adding only some new behaviors to the design model of component without losing the old behaviors. When the design model of a component is evolved after adapting some refinements, the whole CBS of many models of the existing components and the evolved model of the evolved component is not required to be rechecked. The framework only checks whether the evolved model satisfies the assumption of the system before evolving. If it does, 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 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.

The third contribution of the research is to propose a framework for modular conformance testing and modular verification of CBS in the context of component evolution at source code level. This framework includes two stages: modular conformance testing (MCT) for updating inaccurate models of the evolved components and assume-guarantee verification for evolving CBS. 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. 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 evolved model M_2' satisfies the assumption $A(p)$ of the CBS before evolving. If it does, 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 an effective approach. The process for rechecking of the evolved CBS in this context is presented in Figure 1.1. 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.

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 some basic concepts for the research. This chapter includes the model and component specifications, safety property and satisfiability, assume-guarantee reasoning, minimal assumption, component evolution, and others related concepts.

Chapter 3 introduce the concept of deterministic finite state automata, the L*-based assumption generation method for assume-guarantee verification of CBS and the black-box checking strategy for verification a system which are improved in our research. A proposed method for new assumption regeneration in the context of the component evo-

lution is also presented.

Chapter 4 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.

In Chapter 5, we propose an effective framework for modular verification of evolving component-based software at 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 model of the evolved component. If the evolved model satisfies the current assumption of the CBS before evolving, 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 the component evolution, in Chapter 6, 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 assume-guarantee verification of evolving CBS at source code level.

Chapter 7 presents three typical CBS: automobile cruise control system, gas oven control system, and banking subsystem and experimental results obtained by applying the proposed approaches for these systems. These experiments are not only to show the practical usefulness of our proposed approaches but also to present how to generalize the proposed approaches for larger CBS (i.e., CBS containing more than two components).

Chapter 8 presents related works.

Finally, we conclude the research and present the future works in Chapter 9.

In this dissertation, Chapter 3, 4, 5 and 6 are the main works 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 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.

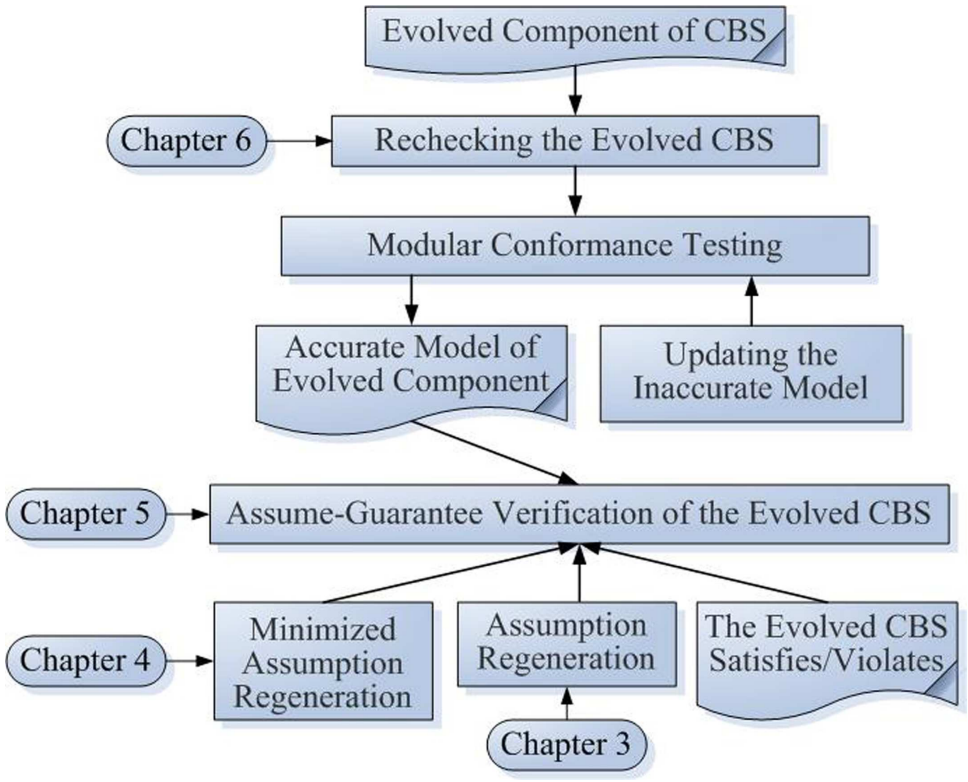


Figure 1.1: The relations between the main chapters of the dissertation and the process for modular verification of evolved CBS.

Otherwise, the component and its model are in conformance. The accurate models of the evolved CBS then are used to recheck the evolved CBS by applying the assume-guarantee verification framework. If the model of the evolved 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 models of the software components which describe the behaviors of communicating components are represented by the *Labeled Transition Systems* (LTSs). An 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 τ denote a local/internal action unobservable to a component's environment. We use π to denote a special error state, which models the fact that a safety violation has occurred in the compositional system. We require that the error state has no outgoing transition.

2.1.1 Labeled Transition Systems

Definition 2.1.1 *(LTSs). An 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.*

Definition 2.1.2 *(LTS size). Size of an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is the number of states of M, denoted $|M|$ (i.e., $|M| = |Q|$).*

Definition 2.1.3 *(deterministic and non-deterministic LTSs). An LTS* $M = \langle Q, \alpha M, \delta, q_0 \rangle$ *is* non-deterministic *if it contains* τ*-transition or if* $\exists (q, a, q'), (q, a, q'') \in \delta$ *such that* $q' \neq q''$*. Otherwise,* M *is* deterministic*.*

Note 2.1.1 *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* $\delta = \delta'$*. We use* $\prod$ *to denote the LTS* $\langle \{\pi\}, Act, \phi, \pi \rangle$*.*

2.1.2 Traces

A trace σ of an LTS M is a sequence of observable actions that M can perform starting at its initial state.

Definition 2.1.4 *(Trace). A trace* σ *of an LTS* $M = \langle Q, \alpha M, \delta, q_0 \rangle$ *is a finite sequence of actions* $a_1 a_2 ... a_n$*, where* $a_1 = q_0$ *and* $a_i \in \alpha M$ $(i = 1, ..., n)$*.*

Note 2.1.2 *For* $\Sigma \subseteq \mathcal{A}ct$*, we use* $\sigma{\uparrow}\Sigma$ *to denote the trace obtained by removing from* σ *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 an 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 models by synchronising the actions common to their alphabets and interleaving the remaining actions.

Definition 2.1.5 *(Parallel composition operator). The parallel composition between* $M_1 = \langle Q_1, \alpha M_1, \delta_1, q_0^1 \rangle$ *and* $M_2 = \langle Q_2, \alpha M_2, \delta_2, q_0^2 \rangle$*, 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 an LTS* $M = \langle Q, \alpha M, \delta, q_0 \rangle$ *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* δ *is given by the rules:*

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

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

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

Example 2.1.1 *When composing the two models represented by two LTSs Input and Output illustrated in Figure 2.1, the actions send and ack are synchronised while the others are 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.*

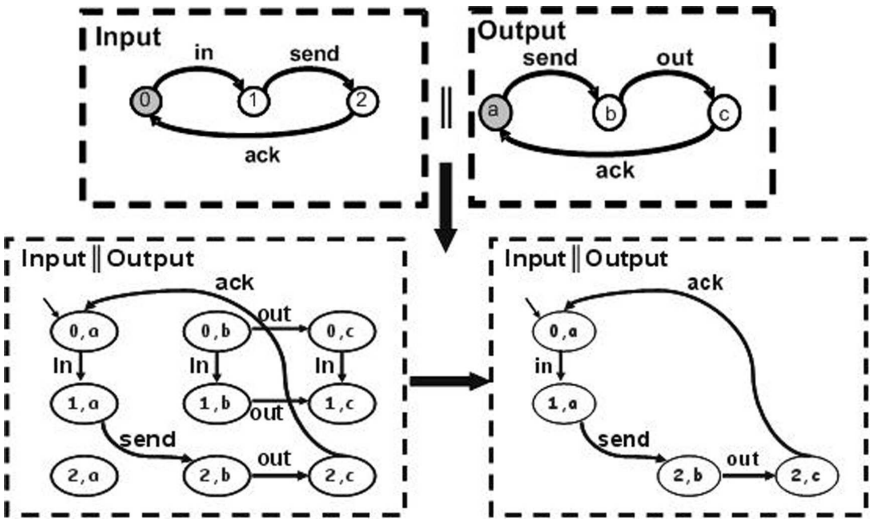


Figure 2.1: An illustration of parallel composition.

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

Definition 2.1.6 *(Safety LTS). A safety LTS is a deterministic LTS that contains no* π *states.*

Definition 2.1.7 *(Safety property.) A safety property asserts that nothing bad happens. The safety property* p *is specified as a safety LTS* $p = \langle Q, \alpha p, \delta, q_0 \rangle$ *whose language* $L(p)$ *defines the set of acceptable behaviors over* αp*.*

Definition 2.1.8 *(Satisfiability). An LTS* M *satisfies* p*, denoted as* $M \models p$*, if and only if* $\forall \sigma \in L(M)$*:* $(\sigma \uparrow \alpha p) \in L(p)$*.*

Note 2.1.3 *When checking of the LTS* M *which satisfies the property* p*, an* error LTS*, denoted* p_{err}*, is created which traps possible violations with the* π *state.* p_{err} *is defined as follows:*

Definition 2.1.9 *(Error LTS). 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* $\delta' = \delta \cup \{(q, a, \pi) \mid a \in \alpha p \text{ and } \nexists q' \in Q : (q, a, q') \in \delta\}$.

Remark 2.1.1 *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 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].*

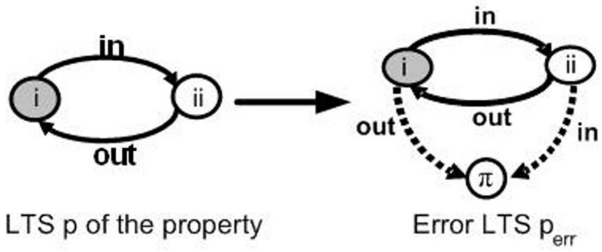


Figure 2.2: The LTS p of the property and the corresponding error LTS p_{err}.

Note 2.1.4 *In order 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* π *is reachable in the composition then M violates p. Otherwise, it satisfies.*

Example 2.1.2 *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* π *is not reachable in this composition. Thus, we conclude that the compositional system* $Input \| Output$ *satisfies the property p.*

2.1.5 Component

Although we consider a software component as a black-box, theoretically, a component can be represented by a finite state machine defined as follows.

Definition 2.1.10 *(Component). Let* $M = \langle Q_M, \Sigma, \delta_M, q_0^M \rangle$ *be a model which describes the behaviors of a component C. C is an unknown finite state machine* $\langle Q_C, \Sigma, \delta_C, q_0^C \rangle$ *where:*

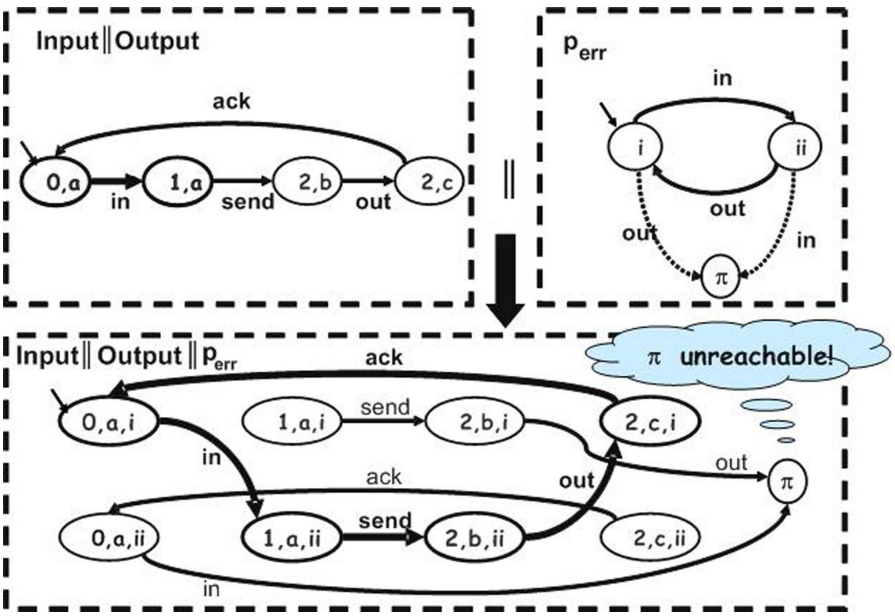


Figure 2.3: Computing the composition $Input \| Output \| p_{err}$.

- Q_C *is a non-empty set of states,*
- $\Sigma \subseteq \mathcal{A}ct$ *is a finite set of actions called the alphabet of* C *(the same alphabet with the model* M*),*
- $\delta \subseteq Q_C \times \Sigma \cup \{\tau\} \times Q_C$ *is a transition relation, and*
- $q_0^C \in Q_C$ *is the initial state.*

Remark 2.1.2 *The unknown finite state machine means that we do not know all its states and transitions because the component* C *is considered as a black-box. We can only perform experiments on the component* C*. With this fact, we view the software component* $C = (\Sigma, T)$ *as a (typically infinite) prefix closed set of strings* $T \subseteq \Sigma^*$ *over the finite alphabet of actions* Σ*.* 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* ϵ
- *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

Definition 2.1.11 *(Accurate model). Let M be a model which describes behavior of a software component C. The 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 Assume-Guarantee Verification

2.2.1 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 π is reachable in the composition. If so, then the formula is violated, otherwise it is satisfied.

Definition 2.2.1 *(Assumption). Given two models M_1 and M_2, and a required safety property p, $A(p)$ is an assumption if and only if it 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 $\langle true \rangle\ M_2\ \langle A(p) \rangle$, called assume-guarantee rule, both hold). Equivalently, $A(p)$ is an assumption if and only if $L(A(p) \| M_1) \uparrow \alpha p \subseteq L(p)$ and $L(M_2) \uparrow \alpha A(p) \subseteq L(A(p))$.*

Remark 2.2.1 *The assumption $A(p)$ is generated by applying the L^* learning algorithm such that $A(p)$ satisfies the assume-guarantee rule. From the rule, the system $M_1 \| M_2$ satisfies p without composing M_1 with M_2. The iterative fashion for generating $A(p)$ is illustrated in Figure 3.7. Details of this fashion will be presented in Section 3.2 of Chapter 3.*

2.2.2 Weakest Assumption

An assumption with which the assume-guarantee rule 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.

Definition 2.2.2 *(Weakest assumption). Weakest 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 π is not reachable in the compositional system $M_1 \| p_{err}$. The weakest assumption A_W means that for any environment component E, $M_1 \| E \models p$ if and only if $E \models A_W$.*

2.2.3 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 of minimal assumption is defined as follows.

Definition 2.2.3 *(Minimal assumption). Given two models M_1, M_2 and a property p, $A(p)$ is an assumption if and only if $A(p)$ satisfies the assume-guarantee rule.* An assumption $A(p)$ represented by an 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.

2.2.4 Labelled Transition Systems Analyzer

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 (LTSs) 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 whether a generated assumption $A(p)$ satisfies the assume-guarantee rule (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.

2.3 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 of *component evolution*: adding only some new behaviors to the old component without losing the old behaviors. Let $C_2 = (\Sigma_2, T_2)$ and $C_2' = (\Sigma_2', T_2')$ be two components as black boxes. C_2' is an evolution of C_2 if and only if $\Sigma_2 \subseteq \Sigma_2'$ and $T_2 \subseteq T_2'$. With regard to the formal definition about the evolution inside of the component, even for this work where we consider the software components as black boxes, theoretically, we can represents components as LTSs or finite state machines. 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' inside of the components as follows.

Definition 2.3.1 *(Component evolution). 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* $L(C_2) \subseteq L(C_2')$.

Remark 2.3.1 *During the software life-cycle, suppose that we have a mechanism to manage versions of each software component. For example, we consider the old versions of the current evolved component C_2'. Currently, we have many versions of this component during the software development process, i.e., C_{20}, C_{21}, C_{22}, ..., C_2, C_2' shown in Figure 2.4, where a new version is produced by adding some new behaviors to the old version. Suppose that we do not allow removing the behaviors of the initial version C_{20}. At the current version C_2', if we allow to remove some old behaviors of the component, suppose that the version management mechanism is good enough, the new version produced by removing the old behaviors is exactly one of the old versions (i.e., one of C_{20}, C_{21}, C_{22}, ..., C_2). 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.*

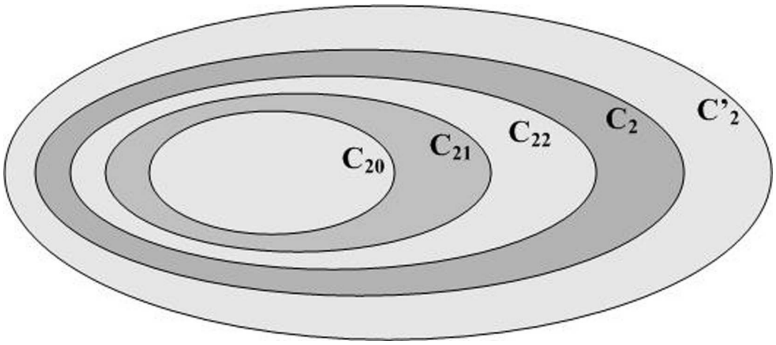


Figure 2.4: Version management mechanism for C_2 by only adding new behaviors.

Example 2.3.1 *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 2.5. 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 model M_2' of the component C_2' is created by adding the transition (b,send,b) into the model M_2. It means that M_2' allows multiple send actions to occur before producing output.*

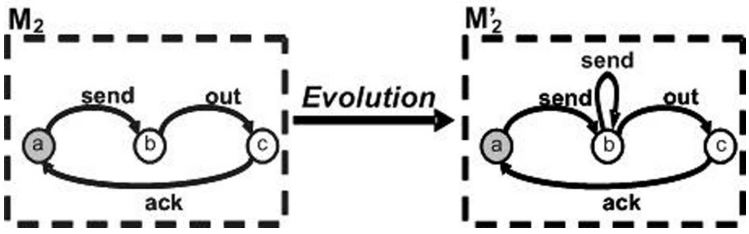


Figure 2.5: An illustration of the component evolution concept.

Chapter 3

L* Learning Algorithm and Black-box Checking

This chapter presents the L* learning algorithm and its applications for assume-guarantee verification of component-based software (CBS) and black-box checking of a system, and proposes an effective method for new assumption regeneration in the context of the component evolution.

3.1 Deterministic Finite State Automata

In this dissertation, we use a learning algorithm called L* [9, 17] to generate an assumption from two models. A framework for assumption generation will be described in Section 3.2.2. In the framework presented in Figure 3.7, 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 the component M_1 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 DFA is defined as follows:

Definition 3.1.1 *(DFA). A DFA M is a five tuple $\langle Q, \alpha M, \delta, q_0, F\rangle$ where:*

- *Q, αM, δ, q_0 are defined as for deterministic LTSs, and*
- *$F \subseteq Q$ is a set of accepting states.*

Note 3.1.1 *For a DFA M and a string σ, we use $\delta(q, \sigma)$ to denote the state that M will be in after reading σ starting at state q. A string σ 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 $L(M) = \{\sigma \mid \delta(q_0, \sigma) \in F\}$.*

Example 3.1.1 *Figure 3.1 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* $aaaab \notin L(M)$.

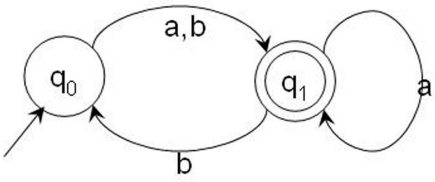


Figure 3.1: An illustration of DFA.

Definition 3.1.2 *(Prefix-closed DFA). A DFA M is* prefix-closed *if $L(M)$ is prefix-closed, i.e., for every $\sigma \in L(M)$, every prefix of σ is also in $L(M)$.*

Remark 3.1.1 *The DFAs returned by the learning algorithm in the proposed approach are* complete, minimal, and prefix-closed. *These DFAs therefore contain a single non-accepting state* nas. *In order to obtain 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:*

Definition 3.1.3 *(DFA to LTS). Let a DFA* $M = \langle Q \cup \{nas\}, \alpha M, \delta, q_0, Q\rangle$, *the safety LTS* $A = \langle Q, \alpha M, \delta \cap (Q \times \alpha M \times Q), q_0\rangle$.

Example 3.1.2 *Figure 3.2 describes an illustrative example to transform a DFA M into a safety LTS A.*

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

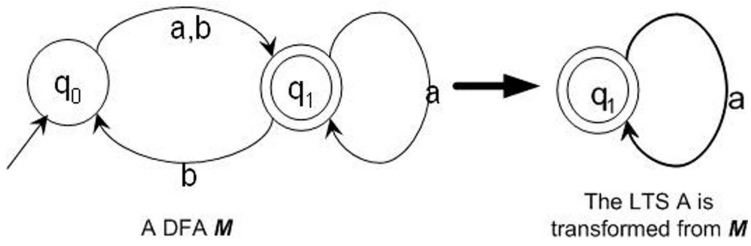


Figure 3.2: An illustration of getting the safety LTS A from the DFA M.

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 Σ. 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 σ 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 3.3.

At a higher level, L* maintains a table T that records whether string s in Σ^* 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) defined as follows:

Definition 3.2.1 *(Observation table).* (S, E, T) *is an observation table built by L*, where:*

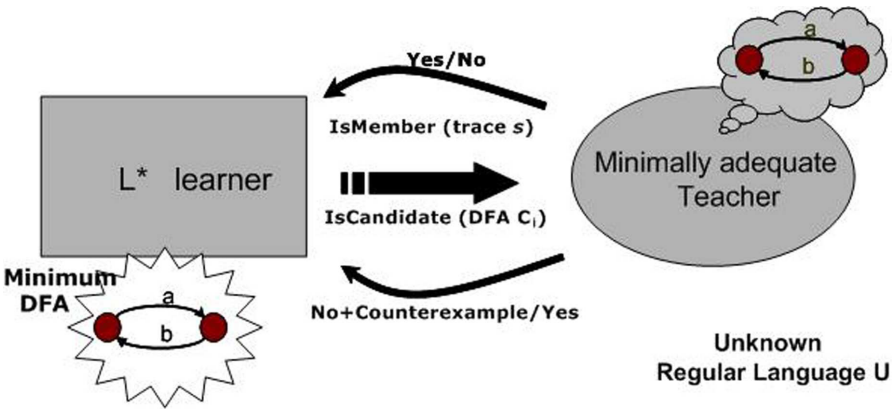


Figure 3.3: The interaction between L* Learner and the Teacher.

- $S \in \Sigma^*$ *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* Σ^**,* $T(s) = true$ *means* $s \in U$*, otherwise* $s \notin U$*.*

Remark 3.2.1 *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 λ 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 is constructed (line 8) from the closed table (S, E, T) as follows:

Definition 3.2.2 *(Closed observation table to DFA). A DFA* $M = \langle Q, \alpha M, \delta, q_0, F \rangle$ *is constructed from a closed table* (S, E, T)*, where:*

- $Q = S$.
- *Alphabet* $\alpha M = \Sigma$, *where* Σ *is the alphabet of the unknown language* U.
- *The transition* δ *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, Σ: an unknown regular language U over some alphabet Σ
Output: M: a DFA M such that M is a minimal deterministic automata corresponding to U and $L(M) = U$

```
1: Initially, S = {λ}, E = {λ}
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 ∈ S and a ∈ Σ
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 ∈ Σ* that witnesses the counterexample cex to E
14:   end if
15: end loop
```

Example 3.2.1 *Figure 3.4 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, λ 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.*

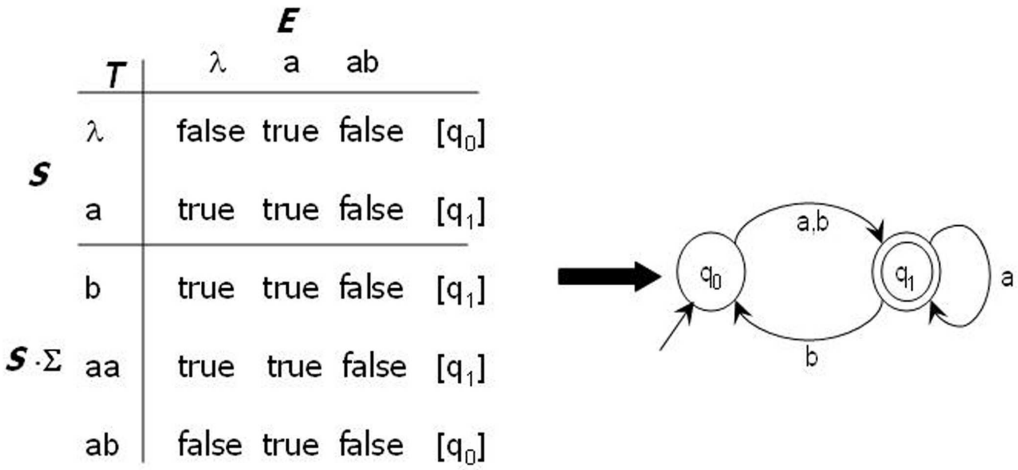


Figure 3.4: An illustration of a closed observation table (S, E, T) and its candidate DFA.

Remark 3.2.2 *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 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 3.5 shows a general view of assume-guarantee verification. The goal is to verify 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 models and a required property represented by LTSs M_1 , M_2 and p, the main goal in this problem is to find an LTS $A(p)$ such that $L(A(p)\|M_1)\uparrow\alpha p \subseteq L(p)$ and $L(M_2)\uparrow\alpha A(p) \subseteq L(A(p))$.

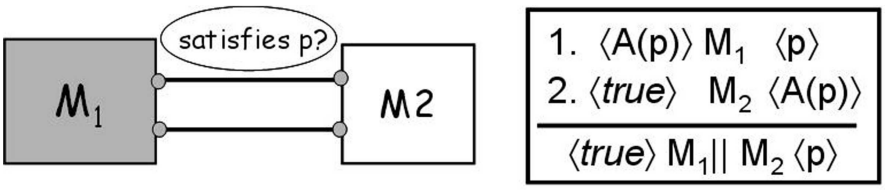


Figure 3.5: A general view of assume-guarantee verification.

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 π in $M_1 \| p_{err}$. For all 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 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 ap-

propriate assumptions, the method applies the compositional rules in an iterative fashion illustrated in Figure 3.7. 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 an environments that guarantees 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 the 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 check 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 an LTS defined as follows:

Definition 3.2.3 *($[cex]$). Let an 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*
- $q^0 = q_0$.

Example 3.2.2 *Figure 3.6 illustrates the LTS $[cex]$ is created from the counterexample $cex = a_1 a_2 \ldots a_k$.*

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

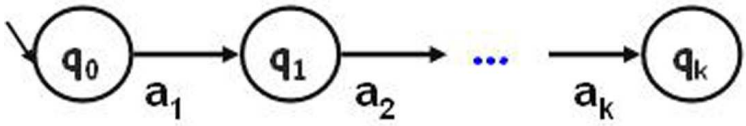


Figure 3.6: The LTS $[cex]$ is created from the counterexample cex.

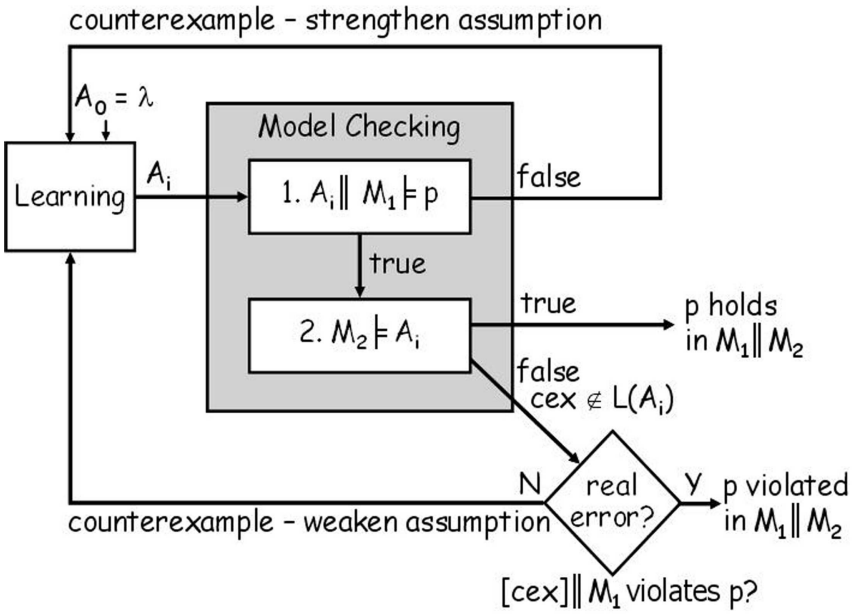


Figure 3.7: The L*-based assumption generation framework.

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 i in the framework illustrated in Figure 3.7. 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 3.1.3. 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 σ 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 $\sigma = a_1 a_2 \ldots a_n$ whether in $\Sigma^* = L(A_W)$, the Teacher simulates the query on the composition $M_1 \| p_{err}$. For the string σ, 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 π 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 σ. 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 check 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 3.7 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 π 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 3.7. 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 3.6. 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 π 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 3.8. 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.

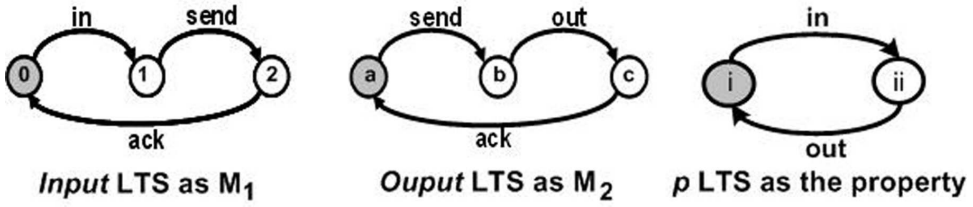


Figure 3.8: Components and order property of the illustrative system.

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 3.9 by setting S and E to $\{\lambda\}$, where λ 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 $send \in L(A_W)?$.

The updated table presented in Figure 3.9 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 3.10. 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* $\in L(A_W)?$.

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

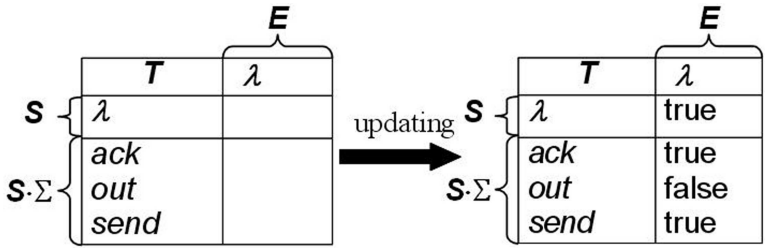


Figure 3.9: The empty observation table and its updated table.

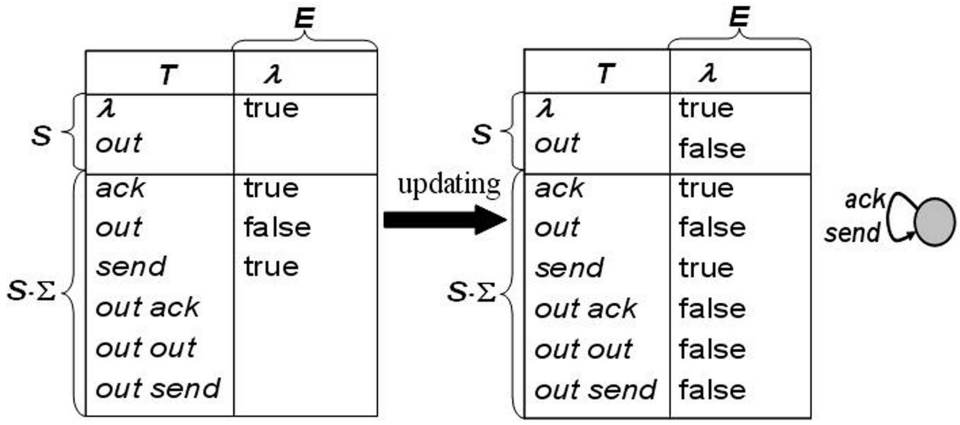


Figure 3.10: The table after adding *out* to S, its updated table, and the candidate assumption A_1.

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 π 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 3.10 is updated by adding the suffix e = *ack* to E. The observation table S, E, T after adding *ack* to E illustrated in Figure 3.11. 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 3.11 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 3.12.

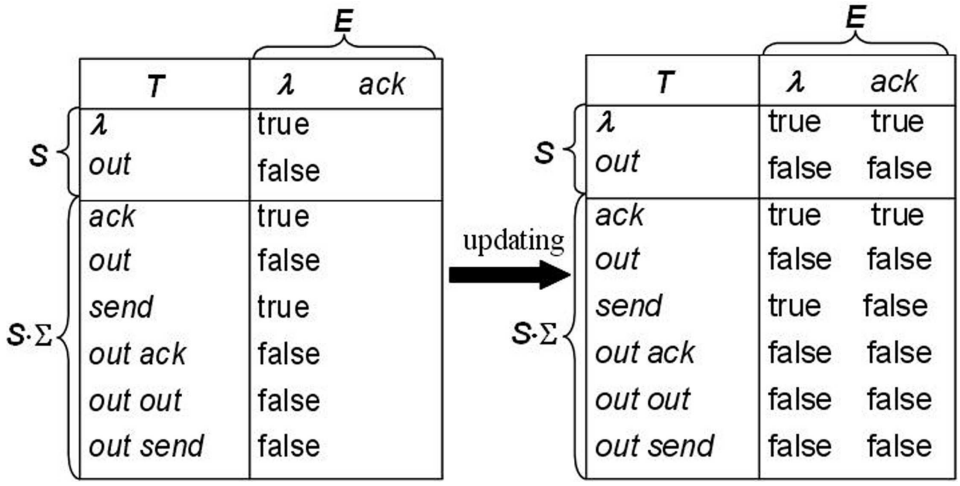


Figure 3.11: The observation table after adding *ack* to S and its updated table.

This table is updated again by making membership queries to the Teacher.

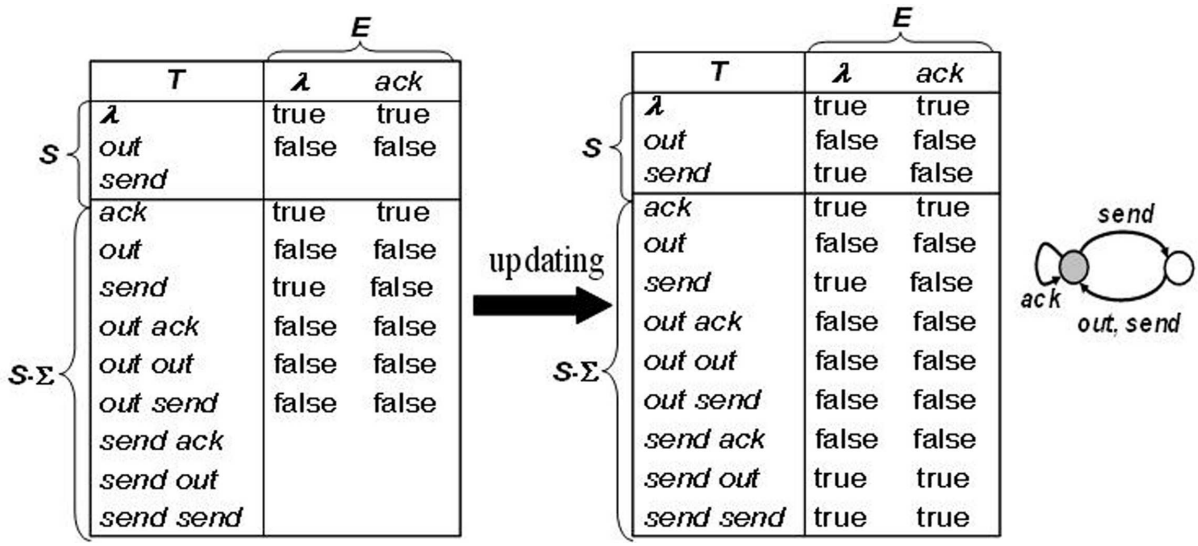


Figure 3.12: The table after adding *send* to S, its updated table, and the candidate assumption A_2.

The updated table S, E, T presented in Figure 3.12 is closed. A candidate DFA A_2 is constructed from this closed observation table shown in Figure 3.12. 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.

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 π 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 π 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 Black-box Checking

This section describes the steps of the black-box checking strategy for verification a system without a model of the system proposed in [47]. The strategy alternates between incremental learning of the system by using a learning algorithm called L* [9, 17], and the black box testing of the learned model against the actual system by using the VC algorithm [46, 49].

The black-box checking strategy is an iterative process shown in Figure 3.13. At any iteration, a candidate model which approximately describes behavior of the actual system is produced by L*. The model is checked (via model checking) whether it satisfies a property. If the model satisfies the property, the VC algorithm is applied to compare the model with the actual system. If they are in conformance, the current model is accurate for the system. In this case, the system is verified and the iterative process terminates. Otherwise, the VC algorithm returns a string/trace, called a discrepancy, that distinguishes the behavior of the system from the model. The discrepancy returned by this step is provided for L* in order to update the approximated model. In the case where the current model does not satisfy the property, a counterexample is returned to witness this fact. The counterexample then is compared with the actual system. If it is an actual execution of the system, the system violates the property and iterative process terminates. Otherwise, the counterexample witnesses a difference between the model and the system. Thus, it is feeded to L* for improving the accuracy of the model.

3.3.1 Learning a System Model via L*

The black-box checking approach use the L* learning algorithm to learn the minimal deterministic automaton M corresponding to a system S, where the system S is seen as a black-box.

In order to learn M, L* needs to interact with a Teacher. The Teacher uses the VC algorithm presented in Subsection 3.3.2 to answer that for each candidate model M_i whether $L(M_i) = L(S)$. If it does not, the Teacher returns a discrepancy d distinguishing M_i from S, i.e., d that is either in $L(M_i) \setminus L(S)$ or $L(S) \setminus L(M_i)$. Otherwise, the Teacher

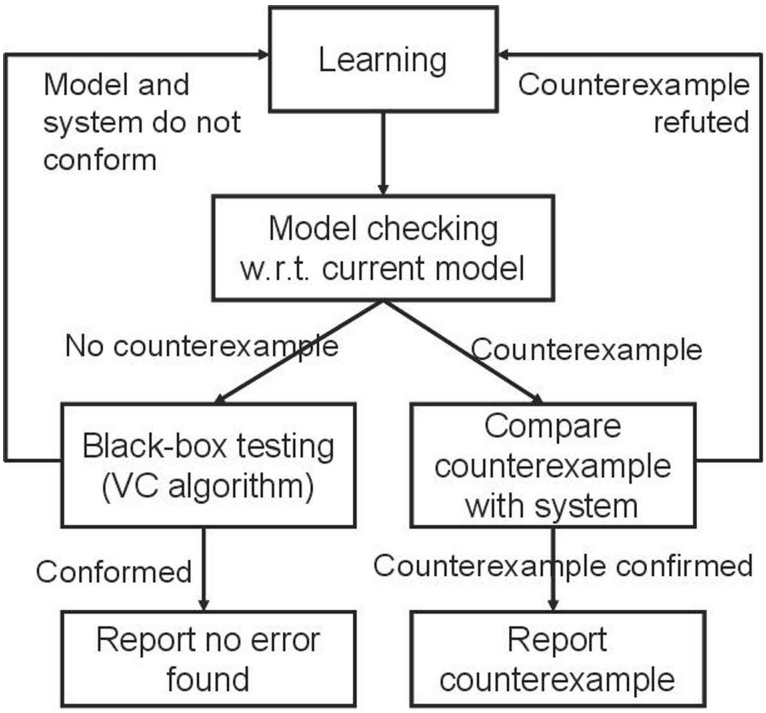


Figure 3.13: The black-box checking strategy [12].

returns *true*. This means that the candidate model M_i is an accurate model of S.

At a higher level, L* maintains a table T that records whether string s in Σ^* belong to $L(S)$ by performing the experiment s on the system S. 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 discrepancy d returned by the Teacher to maintain the table with d that witnesses differences between $L(M_i)$ and $L(S)$.

For more details, in order to learn a system model, L* builds an observation table (V, W, T) defined as follows:

Definition 3.3.1 *(Observation table for learning a system model).* (V, W, T) *is an observation table built by L*, where:*

- $V \in \Sigma^*$ *is a set of prefixes. It presents equivalence classes or states.*
- $W \in \Sigma^*$ *is a set of suffixes. It presents the distinguishing.*
- T*:* $(V \cup V.\Sigma).W \mapsto \{true, false\}$*. With a string* s *in* Σ^**,* $T(s) = true$ *means* $s \in L(S)$*, otherwise* $s \notin L(S)$*.*

Remark 3.3.1 *An observation table* (V, W, T) *is closed if* $\forall v \in V$*,* $\forall a \in \Sigma$*,* $\exists v' \in V$*,* $\forall w \in W$*:* $T(vaw) = T(v'w)$*. In this case,* v' *presents the next state from* v *after seeing* a*,* va *is undistinguishable form* v' *by any of suffixes. Intuitively, the observation table* (S, E, T) *is closed means that every row* va *of* $V.\Sigma$ *has a matching row* v' *in* V*.*

The iterative process of the black-box checking strategy shown in Figure 3.13 starts to verify the system with the empty model. In this process, at any iteration, if the current model is required to be updated with a discrepancy d returned by the Teacher, the L* is called to maintain the current observation table with d for this purpose. Algorithm 2 is the formal description of one phase of the L* algorithm after the Teacher returned a discrepancy d. If the given observation table (V, W, T) is empty (line 1), this means that it is the initial step of the iterative process for the black-box checking strategy, L* sets V and W to $\{\lambda\}$ (line 2&3). Subsequently, it updates the function T (line 4) by setting $T(a, \lambda) = true$ for all $a \in \Sigma$ executable in the black box S after a **reset**, where λ presents the empty string. Otherwise, the given observation table (V, W, T) is not empty (line 5). The discrepancy d is analyzed by L* to find all of its suffixes that witness a difference between $L(M_i)$ and $L(S)$. After that, for each suffix v' of d, if $v' \notin W$ then v' must be added to W (line 6&7). Consequently, T must be updated by performing the experiments on the system S (line 9). L* then checks whether the observation table (V, W, T) is closed (line 11). If (V, W, T) is not closed, then va is added to V, where $v \in V$ and $a \in \Sigma$ are the elements for which there is no $v' \in V$ (line 12). Because va has been added to V, T must be updated again by performing the experiments on the system S (line 13). Line 12 and line 13 are repeated until the table (V, W, T) is closed. When the observation table (V, W, T) is closed, this phase of the L* algorithm terminates and returns the updated closed table (V, W, T).

Algorithm 2 $Lstar(V, W, T, d)$

Input: V, W, T, d: the current observation table (V, W, T) and the discrepancy d returned by the Teacher

Output: V, W, T: a maintained table (V, W, T) after using the discrepancy d

1: **if** (V, W, T) is empty **then**
2: $V = \{\lambda\}$
3: $W = \{\lambda\}$
4: update T by performing the experiments on the system S.
5: **else**
6: **for** each $v' \in suffix(d)$ that is not in W **do**
7: add v' to W
8: **end for**
9: update T by performing the experiments on the system S.
10: **end if**
11: **while** (V, W, T) is *not closed* **do**
12: add va to V to make V closed, where $v \in V$ and $a \in \Sigma$
13: update T by performing the experiments on the system S.
14: **end while**
15: **return** (V, W, T)

Characteristics of the L* Leaning Algorithm. The complexity of the L* learning algorithm is $\mathcal{O}(kn^2 + n \log m)$ [9], where k is the size of alphabet Σ, n is the number of states of the minimal deterministic automaton modelling the black box S, and m is the length of the longest discrepancy returned by the Teacher when a conjecture is made.

3.3.2 Vasilevskii-Chow Algorithm

As mentioned above, the Teacher is built from the VC algorithm. In the iterative process of the black-box checking strategy shown in Figure 3.13, at each iteration i, L* first produces a candidate DFA D_i based on the updated closed observation table V, W, T as follows.

Definition 3.3.2 *$(V, W, T$ to DFA). A candidate DFA $D_i = \langle Q, \alpha D_i, \delta, q_0, F \rangle$ is constructed from the closed table (V, W, T), where:*

- $Q = V$.
- *Alphabet $\alpha D_i = \Sigma$, where Σ is the alphabet of the language $L(S)$ of the actual system S.*
- *The transition δ is defined as $\delta(v, a) = v'$ where $\forall w \in W : T(vaw) = T(v'w)$*
- *initial state $q_0 = \lambda$.*
- $F = \{v \in V \mid T(v) = true\}$.

L* then translates the candidate DFA D_i into a safety LTS as a candidate model M_i of the system S by applying the definition 3.1.3. In order to check conformance between the candidate model M_i and the system S, the VC algorithm checks some strings $s \in \Sigma^*$ whether s is either in both $L(M_i)$ and $L(S)$ or in neither of these sets. Let *check* be a function which maps Σ^* to $\{0, 1\}$. For each string $s \in \Sigma^*$, $check(s) = 0$ if and only if either $s \in L(M_i)$ and $s \in L(S)$ or $s \notin L(M_i)$ and $s \notin L(S)$. For this purpose, the algorithm uses the sets V, W of the given closed observation table (V, W, T) and a known upper bound n on the size of the minimal deterministic automaton modelling the black box S shown in Algorithm 3. The strings that are checked are those of the form $s = vxw$, where $v \in V$, $w \in W$, and $|x| \leq n - |V|$.

Characteristics of the VC Algorithm. The complexity of the VC learning algorithm is $\mathcal{O}(t^2 k^{n-t+1})$ [46, 49], where t is the size of V (the number of states of M_i), k is the size of alphabet Σ, n is the known upper bound n on the size of the minimal deterministic automaton modelling the black box S.

Algorithm 3 $VC(V, W, S, n)$

Input: V, W, n: the set V of prefixes and the set W of suffixes of (V, W, T), the actual system S as a black box, and a known upper bound n on the size of the minimal deterministic automaton modelling the black box S.

Output: *true* or d: *true* if M_i and S are in conformance, and a discrepancy d, otherwise

```
1: t = |V|
2: for l = 1 to n − t do
3:    for each string x of size l, v ∈ V, w ∈ W do
4:       if check(vxw) then
5:          return vxw
6:       end if
7:    end for
8: end for
9: return true
```

3.4 New Assumption Regeneration Method

The L*-based assumption generation method presented in Section 3.2 designed for checking of the fixed systems. Thus, it is not prepared for future evolutions. When the model M_2 is evolved to M_2' by adding some new behaviors to the model, if the current assumption $A(p)$ of the CBS $M_1 \| M_2$ before evolving is too strong to be satisfied by M_2', a new assumption $A_{new}(p)$ must be generated again. 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 Σ. The L* learning algorithm described in Section 3.2 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 Σ^*. 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 regenerates the new assumption $A_{new}(p)$ also using the L* learning algorithm illustrated in Figure 3.7. 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 λ 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 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 λ). Because the initial assumption $A(p)$ is weaker than λ (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 4. 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 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 3.1.3. 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 3.14 presents an iterative framework to 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

Algorithm 4 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 Σ, 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 $L(M) = U$

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

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 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 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 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 an 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.

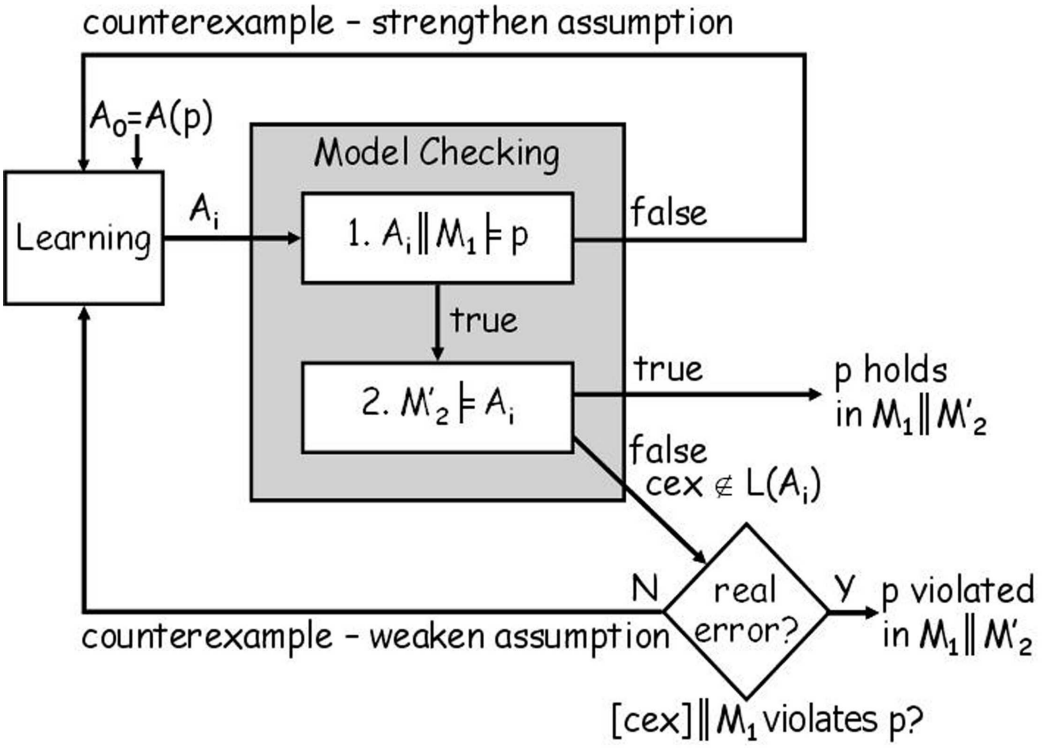


Figure 3.14: The iterative framework for the new assumption regeneration using the improved L* learning algorithm.

With regard to effectiveness of the proposed algorithm, Figure 3.15 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, λ is the strongest assumption. It is the initial assumption in the L* used to generate the assumption $A(p)$ of the compositional CBS $M_1 \| M_2$ before evolving [10]. In the case where the model M_2' of the evolved component does not satisfy the assumption $A(p)$ (i.e., $M_2' \not\models A(p)$), we reuse the entire assumption $A(p)$ as the initial assumption for the improved L* to generate again 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 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 assumption $A(p)$, the proposed method can reduce the large number of steps required in the assumption regeneration process.

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 $A(p)$.

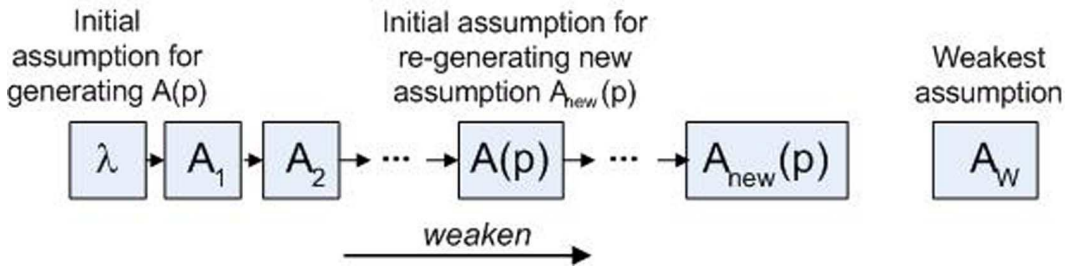


Figure 3.15: The process for new assumption regeneration using the improved L*.

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 $\mathcal{O}(kn^2 + n \log m)$ and $\mathcal{O}(kn'^2 + n' \log m)$ respectively [10], where k is size of the alphabet Σ, 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 $\mathcal{O}(k(n'^2 - n^2) + (n' - n) \log m)$. These facts imply that our method is more effective than the method proposed in [10] in the context of the rechecking of the evolving CBS.

Chapter 4

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.

4.1 Minimized Assumption Generation Method

The assumptions generated by the assume-generation verification proposed in [10] are not minimal. Figure 4.1 is a counterexample to prove this fact. In this counterexample, given two 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 $\Sigma = (\alpha M_1 \cup \alpha p) \cap \alpha M_2$ and produces a DFA that accepts it. In order to learn this language, L* builds an observation table (S, E, T) (see definition 3.2.1). The technique for answering membership queries

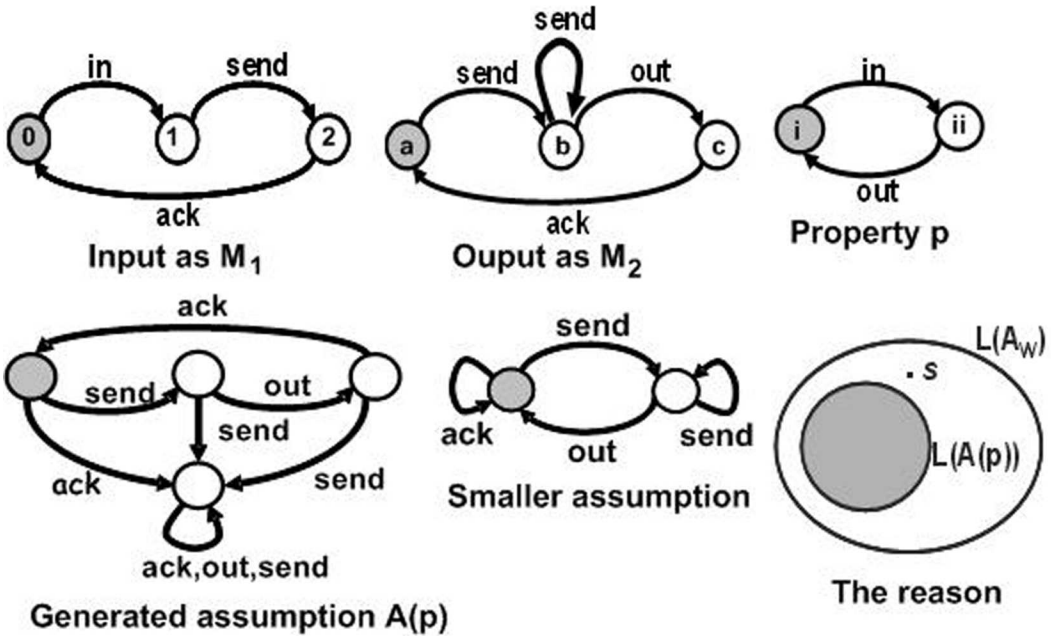


Figure 4.1: A counterexample and the reason to show that the assumptions generated in [10] are not minimal.

used in this method is defined as follows:

Definition 4.1.1 *(Function for answering membership queries). Given an observation table (S, E, T), T is a function which maps $(S \cup S.\Sigma).E$ to $\{true, false\}$ such that for any string $s \in (S \cup S.\Sigma).E$, $T(s) = true$ if $s \in L(A_W)$, and $false$ otherwise.*

Remark 4.1.1 *In the counterexample showed in Figure 4.1, 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 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 being learned.*

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, 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 large-scale systems.

This section proposes a method for generating minimal assumptions for assume-guarantee verification of CBS. We also define a new technique for answering membership queries to deal 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).

4.1.1 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 $\{true, false\}$ (see definition 4.1.1). In the case where $s \in L(A_W)$, we cannot ensure whether s belongs to the language being 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, , where "?" can be seen as a "don't know" value. We define an improved technique for answering membership queries as follows.

Definition 4.1.2 *(Improved function for answering membership queries). Given an observation table (S, E, T), T is a function which maps $(S \cup S.\Sigma).E$ to $\{true, false,$ "?"$\}$ such that for any string $s \in (S \cup S.\Sigma).E$, if s is the empty string $(s = \lambda)$ then $T(s) = true$, else $T(s) = false$ if $s \notin L(A_W)$, and "?" otherwise.*

Remark 4.1.2 *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 being learned or not.*

4.1.2 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 5. 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 $S_0 = E_0 = \{\lambda\}$ (λ 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 4.1 and one of its instance obtained by replacing all "?" entries with $true$ value are showed in Figure 4.2. 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 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 whether M_1 satisfies p in an environment that guarantees 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 the language of the assumption being 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 5 Minimized assumption generation.

Input: M_1, M_2, p: two 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

```
1: Initially, q = empty {q is an empty queue}
2: q.put(OT_0) {OT_0 = (S_0, E_0, T_0), S_0 = E_0 = {λ}, where λ is the empty string}
3: while q ≠ empty do
4:   OT_i = q.get() {getting OT_i from the top of q}
5:   if OT_i contains "?" value then
6:     for each instance OT of OT_i do
7:       q.put(OT) {putting OT into q}
8:     end for
9:   else
10:    if OT_i is not closed then
11:      OT = make_closed(OT_i)
12:      q.put(OT)
13:    else
14:      construct a candidate DFA A_i from the closed OT_i
15:      if Step1(A_i) fails with cex then
16:        add the suffix e of the counterexample cex to E_i
17:        OT = update(OT_i)
18:        q.put(OT)
19:      else
20:        if Step2(A_i) fails with cex then
21:          if cex witnesses violation of p then
22:            return cex
23:          else
24:            add the suffix e of the counterexample cex to E_i
25:            OT = update(OT_i)
26:            q.put(OT)
27:          end if
28:        else
29:          return A_i
30:        end if
31:      end if
32:    end if
33:  end if
34: end while
```

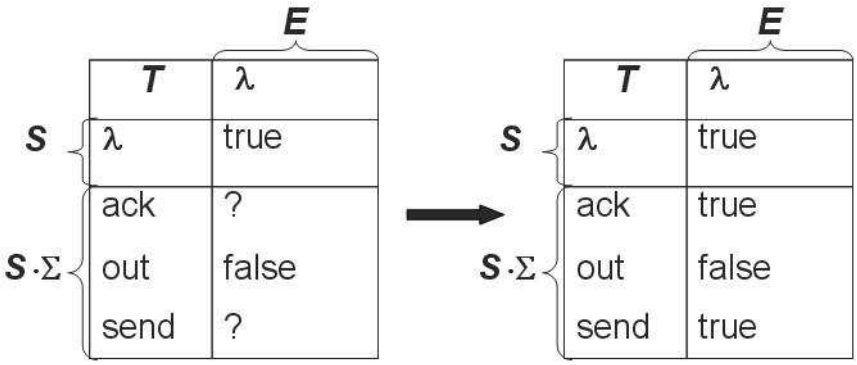


Figure 4.2: The initial observation table and one of its instances.

4.1.3 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 jth candidate assumption generated from the jth observation table (denoted OT_{ij}) at the depth i of the search tree. From the way to build the search tree presented in Algorithm 5, 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 $i < k$. ∎

4.1.4 Termination and Correctness

The termination and correctness of the proposed algorithm for the minimized assumption generation showed in Algorithm 5 are proved by the following theorem.

Theorem 2 *Given two models M_1 and M_2, and a property p, the proposed algorithm for the minimized assumption generation presented in Algorithm 5 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 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 σ 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 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 generate A less than the depth of the table used to generate $A_m(p)$. This means that the table used to generate 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. ∎

4.2 Reducing the Search Space

In the algorithm for minimal assumption generation shown in Algorithm 5, the queue has to hold an exponentially growing of the number of the observation tables. This makes

our method unpractical for large-scale systems because it consumed too much memory. For large-scale systems, the computational cost for generating the minimal assumption is very high. 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.

4.2.1 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 6. In this algorithm, we use a stack data structure which contains the generated observation tables with the *last-in first-out* order.

4.2.2 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 cannot 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 7 presents the algorithm named $IDDFS$ for generating the assumption by using iterative-deepening depth first search. Algorithm 8 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 $DLS(d)$ cannot find an

Algorithm 6 Assumption generation algorithm by using DFS.

Input: M_1, M_2, p: two 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

```
1: Initially, s = empty {s is an empty stack}
2: 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}
3: while s ≠ empty do
4:   OT_i = s.pop() {getting OT_i from the top of s}
5:   if OT_i contains "?" value then
6:     for each instance OT of OT_i do
7:       s.push(OT) {putting OT into the top of s}
8:     end for
9:   else
10:    if OT_i is not closed then
11:      OT = make_closed(OT_i)
12:      s.push(OT) {putting OT into the top of s}
13:    else
14:      construct a candidate DFA A_i from the closed OT_i
15:      if Step1(A_i) fails with cex then
16:        add the suffix e of the counterexample cex to E_i
17:        OT = update(OT_i)
18:        s.push(OT) {putting OT into the top of s}
19:      else
20:        if Step2(A_i) fails with cex then
21:          if cex witnesses violation of p then
22:            return cex
23:          else
24:            add the suffix e of the counterexample cex to E_i
25:            OT = update(OT_i)
26:            s.push(OT) {putting OT into the top of s}
27:          end if
28:        else
29:          return A_i
30:        end if
31:      end if
32:    end if
33:  end if
34: end while
```

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

Algorithm 7 $IDDFS(M_1, M_2, p, max)$

Input: M_1, M_2, p, max: two 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

```
1: for i = 0 to max do
2:   DLS(i) {Applying depth-limited search with depth limit i}
3:   if DLS(i) returns an assumption A(p) then
4:     return A(p)
5:   else
6:     if DLS(i) returns a counterexample cex then
7:       return cex
8:     end if
9:   end if
10: end for
```

4.2.3 Reusing the Previous Verification Result

Another idea to reduce the search space of the observation tables of the proposed method presented in Section 4.1 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 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 model M_2 is evolved to a new model M_2' by adding some new behaviors to M_2. The evolved compositional system $M_1 \| M_2'$ must be rechecked whether it satisfies the property p. For this purpose, the proposed method in this section only checks the evolved 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 4.1. 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 evolved model M_2' is regenerated. The proposed method regenerates the new minimal assumption $A_{mnew}(p)$ by applying the proposed method presented in Section 4.1 with the initial observation table as the observation table of $A_m(p)$. By staring from the observation table of $A_m(p)$,

Algorithm 8 $DLS(d)$

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

1: Initially, $s = empty$ {s is an empty stack}
2: $s.push(OT_0)$ {putting OT_0 into the top of s, where $OT_0 = (S_0, E_0, T_0), S_0 = E_0 = \{\lambda\}$, and λ is the empty string}
3: $depth(OT_0) = 0$
4: **while** $s \neq empty$ **do**
5: $OT_i = s.pop()$ {getting OT_i from the top of s}
6: **if** $depth(OT_i) \leq d$ **then**
7: **if** OT_i contains "?" value **then**
8: **for** each instance OT of OT_i **do**
9: $s.push(OT)$ {putting OT into the top of s}
10: $depth(OT) = depth(OT_i) + 1$
11: **end for**
12: **else**
13: **if** OT_i is not closed **then**
14: $OT = make\_closed(OT_i)$
15: $s.push(OT)$ {putting OT into the top of s}
16: $depth(OT) = depth(OT_i) + 1$
17: **else**
18: construct a candidate DFA A_i from the closed OT_i
19: **if** $Step1(A_i)$ fails with cex **then**
20: add the suffix e of the counterexample cex to E_i
21: $OT = update(OT_i)$
22: $s.push(OT)$ {putting OT into the top of s}
23: $depth(OT) = depth(OT_i) + 1$
24: **else**
25: **if** $Step2(A_i)$ fails with cex **then**
26: **if** cex witnesses violation of p **then**
27: **return** cex
28: **else**
29: add the suffix e of the counterexample cex to E_i
30: $OT = update(OT_i)$
31: $s.push(OT)$ {putting OT into the top of s}
32: $depth(OT) = depth(OT_i) + 1$
33: **end if**
34: **else**
35: **return** A_i
36: **end if**
37: **end if**
38: **end if**
39: **end if**
40: **else**
41: **return** $notfound$
42: **end if**
43: **end while**

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 $L(A_i)$ is exactly a subset of the language of the assumption being 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 5.2.2 of Chapter 5.

4.3 Small Experiment

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) [31]. 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 [31]. Details of the introduction to functional programming and OCaml can be found in [18, 19] and in [26, 31, 32, 34] respectively. Although the AG tool for L*-based assumption generation method proposed in [10] have been implemented and presented in [36], 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 4.3 shows the architecture of the implemented AG tool and an example which illustrates how to use the tool. Inputs of this tool are two 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 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 4.3. With regard to correctness of our implementation about the AG tool, checking correctness of the tool is very difficult. The correctness of the AG tool implementation means that we have to check 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 2.2.4 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, correctness of A generated by our tool is proven. Figure 4.4 presents an example for checking correctness of the assumption A generated by the AG tool in the LTSA tool.

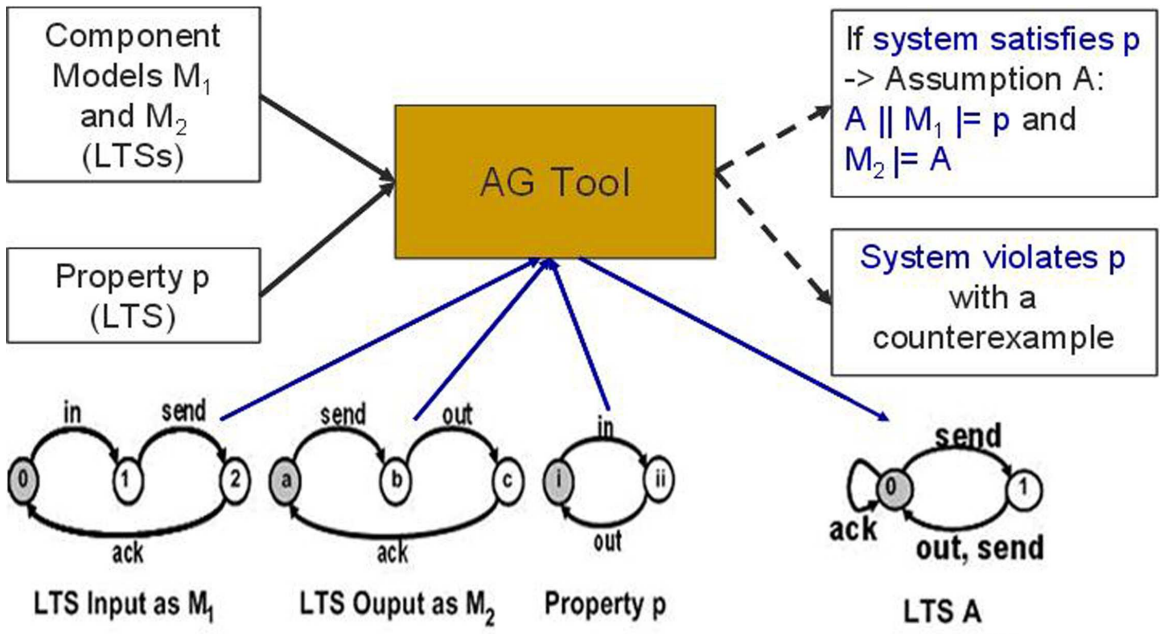


Figure 4.3: The architecture of the AG tool and an example.

With regard to the MAG tool which supports the proposed minimized assumption generation method, Figure 4.5 shows the architecture of the implemented MAG tool and an example which illustrates how to use the tool. Inputs of this tool are two models 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 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 4.5. The correctness of A_m also is checked by using the LTSA tool presented in Figure 4.6.

In order to evaluate the effectiveness of the proposed method, we have tested our method by using several variations of the Sender/Receiver example shown in Figure 3.8 and compared the method with that proposed in [10]. The sizes, the numbers 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

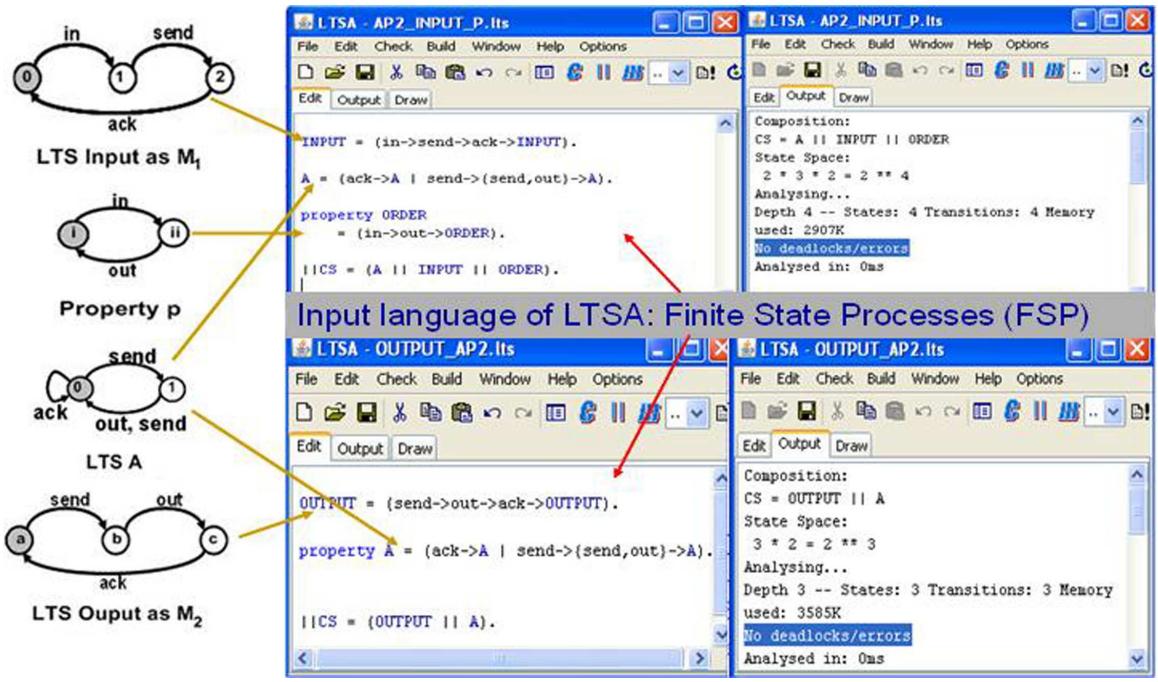


Figure 4.4: An example for checking correctness of the AG tool via LTSA.

assumptions for checking of the compositional rules. Table 4.1 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 [33].

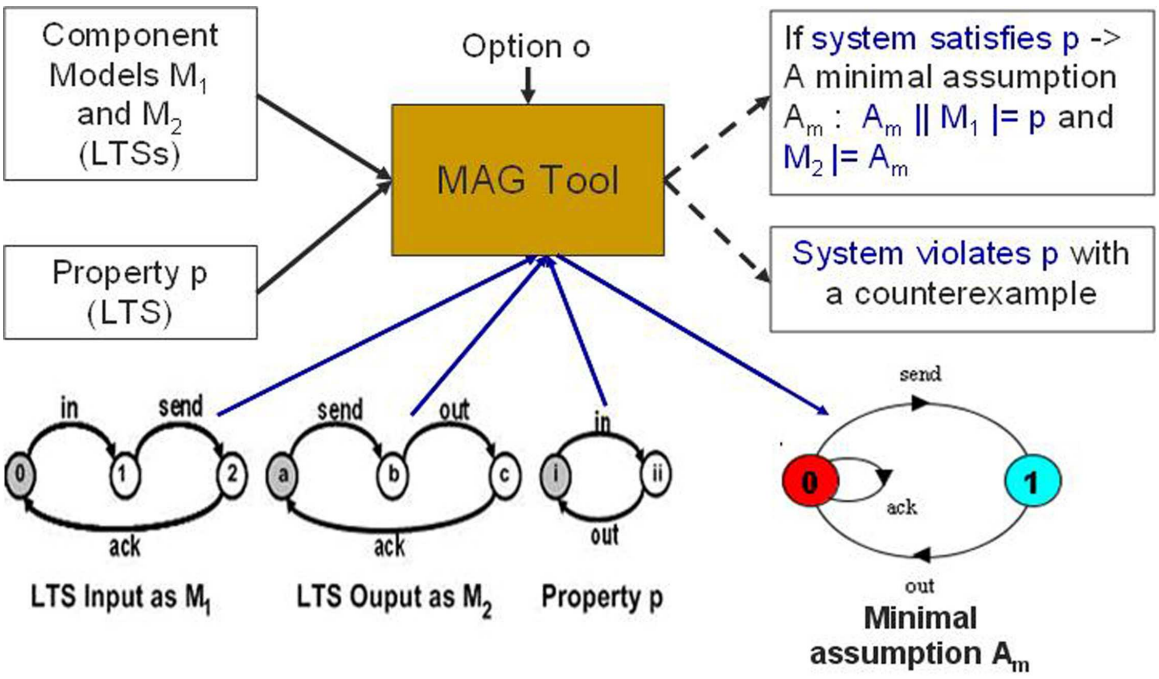


Figure 4.5: The architecture of the MAG tool and an example.

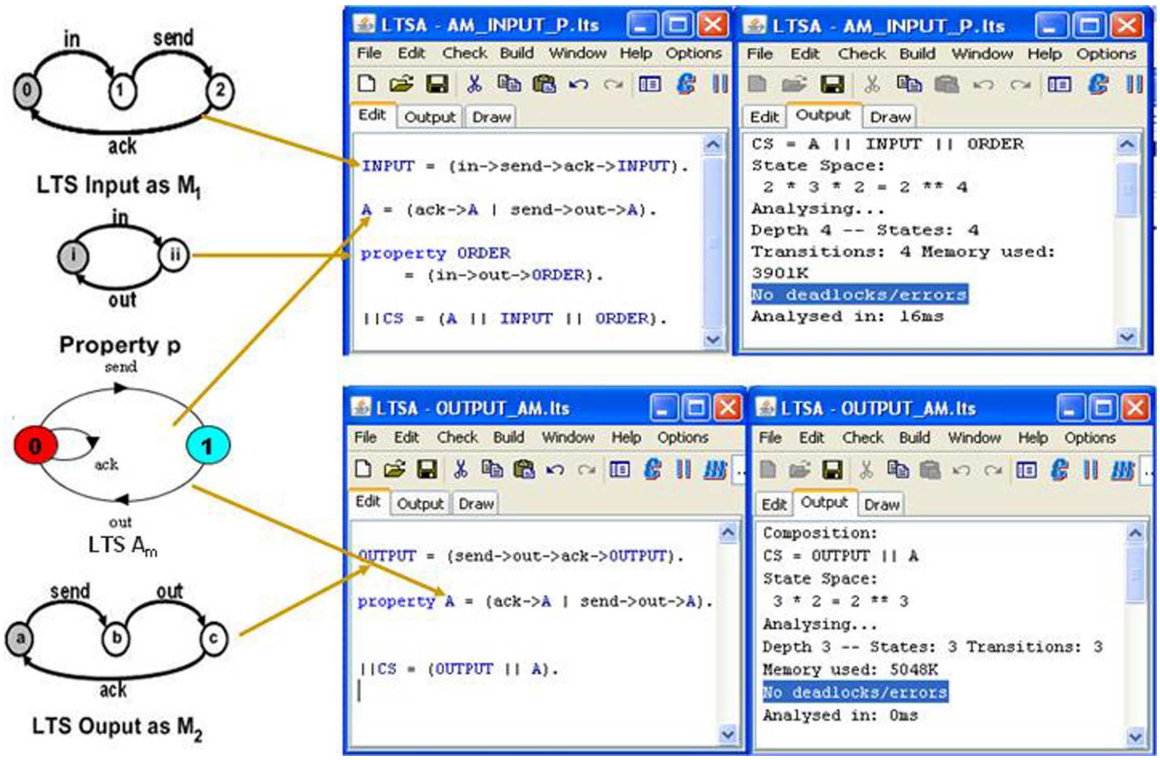


Figure 4.6: An example for checking correctness of the MAG tool via LTSA.

Table 4.1: 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)
Sender/Receiver	18	**2**	4	**93**	7.8	**2**	3	**94**	7.6
Sender/Receiver (Evolved Channel)	18	**4**	9	**97**	9.5	**2**	4	**102**	6.3
Sender/Receiver (Two Channels)	75	**3**	12	**94**	37.5	**3**	6	**107**	23.5

Chapter 5

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 design level. In this framework, if the model of a component is evolved after adapting some refinements, the whole component-based software (CBS) of many models of the existing components and the evolved model of the evolved component is not required to be rechecked. The method only checks whether the evolved model satisfies the assumption of the system before evolving. If so, the evolved CBS still satisfies the property. Otherwise, if the assumption is too strong to be satisfied by the evolved model, and 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.

5.1 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 design level. The component evolution means *adding only some new behaviors to the component without losing the old behaviors*. In the proposed framework, if the design model of a component is evolved to a new model by adding some new behaviors, the whole evolved CBS is not required to be rechecked. We only focus on the evolved model to recheck the evolved CBS. The

framework only checks whether the evolved model satisfies the current assumption of the CBS before evolving. 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 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.

5.1.1 A Framework for Assume-Guarantee Verification of Evolving CBS

Currently, there are many approaches proposed in modular verification of CBS [20, 10, 16, 37, 38, 39, 40]. In these approaches, modular verification is rather closed for fixed 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 models of the evolved components 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 5.1. Suppose that there is a simple component-based software which contains a model M_1 of the base component as a fixed framework, and a model M_2 of the extensional component. 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 model M_2' by adding some new behaviors to the model M_2. The major goal of the proposed method is to verify if 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 system $M_1 \| M_2$ satisfying the property p, we have generated an assumption $A(p)$ that is strong enough for

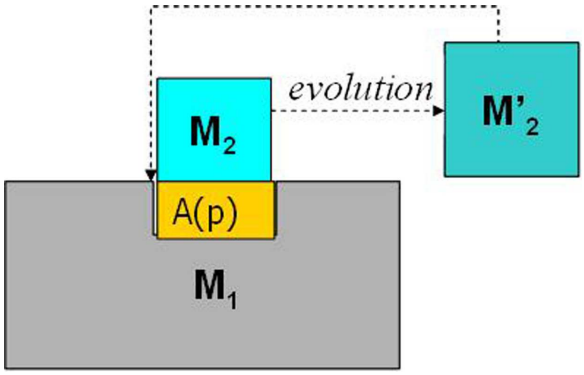


Figure 5.1: A simple case for evolving component-based software.

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 5.2 presents the proposed framework for rechecking the evolved CBS. When the model M_2 of the extensional 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 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 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 π 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 new assumption regeneration method presented in Section 3.4 of Chapter 3 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 assump-

tions. In some cases where the current assumptions are actual assumptions of the evolved CBS, these CBS are verified in the fastest way without regenerating new assumptions.

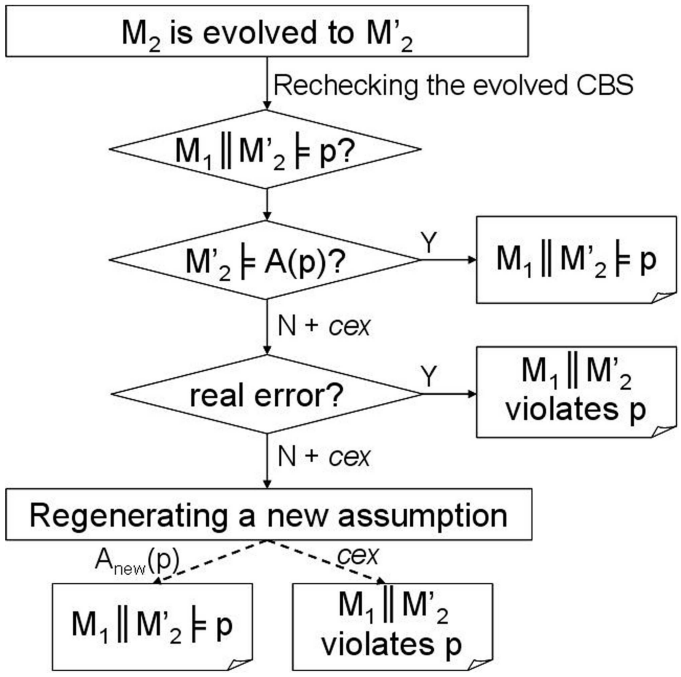


Figure 5.2: The proposed framework for modular verification of evolved CBS.

5.1.2 Correctness and Termination

Correctness and termination of the proposed framework is proved by the following theorem.

Theorem 3 *Given an accurate model M_1, an evolved 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 framework terminates and returns true if M_2' still satisfies $A(p)$, returns a new assumption $A_{new}(p)$ if the evolved compositional CBS $M_1 \| M_2'$ satisfies p, and $false$ otherwise.*

Proof The proposed framework terminates and returns *true* if M_2' satisfies $A(p)$, and $false$ if the evolved system $M_1 \| M_2'$ violates p. Otherwise, the current assumption $A(p)$ is too strong to be satisfied by M_2'. In this case, the proposed method for new assumption regeneration uses two steps of the compositional rule (i.e., $\langle A(p) \rangle$ M_1 $\langle p \rangle$ and $\langle true \rangle$ 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

σ 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 whether we always achieve the new assumption $A_{new}(p)$ from the 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]. ∎

5.1.3 Examples

Figure 5.3 describes an illustrative concurrent CBS which contains two 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 3.8 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 3.7 in Chapter 3 that is strong enough for M_1 to satisfy p but weak enough to be discharged by M_2.

The model M_2 of the extensional component is then evolved to a evolved model M_2' presented in Figure 5.4 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 assumption $A(p)$ to regenerate the new assumption $A_{new}(p)$ shown in Figure 5.4. 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

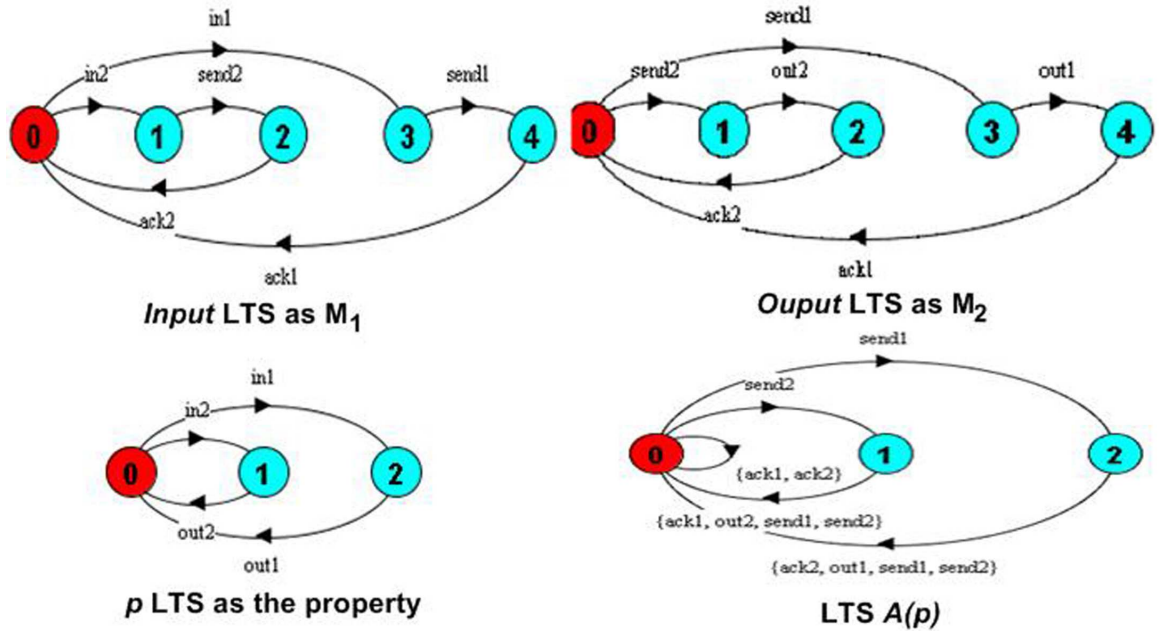


Figure 5.3: Models of the components, order property, and assumption $A(p)$ of the illustrative CBS.

much lower computational cost with 3 generated candidate assumptions and 210 required membership queries.

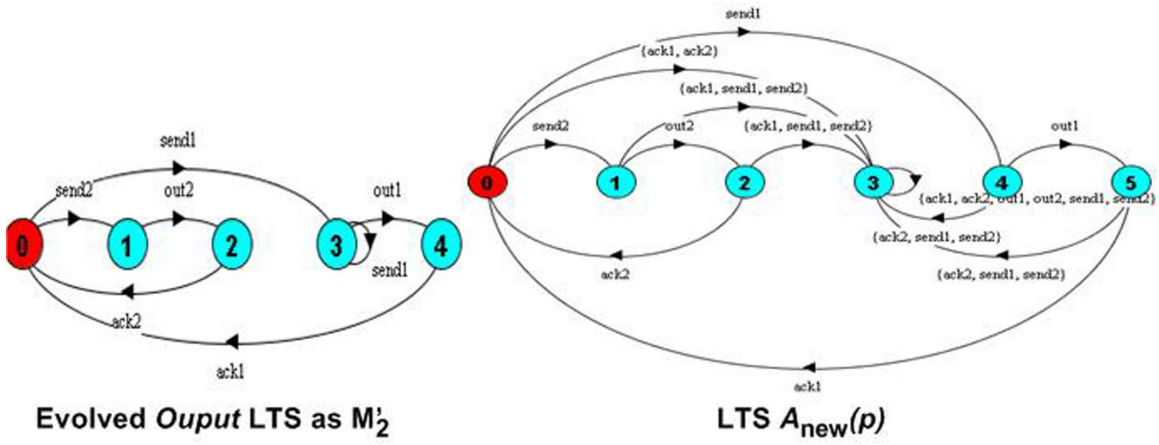


Figure 5.4: The evolved model M_2' and the new assumption $A_{new}(p)$ of the evolved CBS.

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 5.5 by adding a new behavior which allows multiple $send2$ actions to occur before producing $out2$ 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.

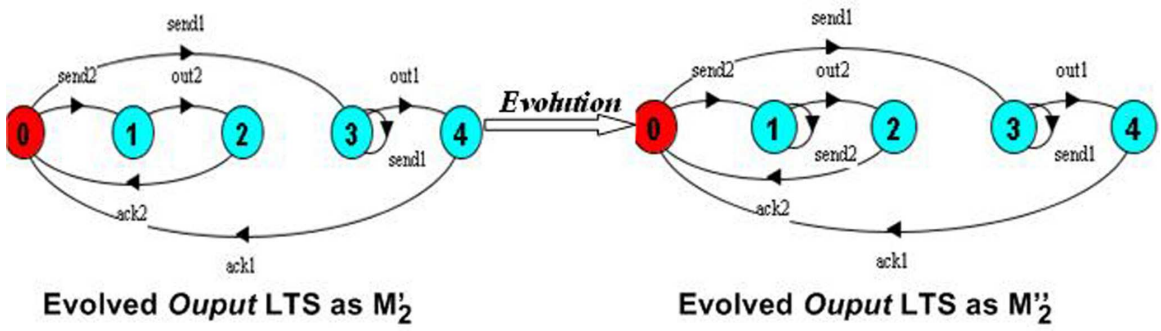


Figure 5.5: The evolved model M_2'' of the model M_2'.

5.2 Optimized the Proposed Framework

5.2.1 Reducing the Number of Candidate Queries

Recall the iterative framework for the new assumption regeneration shown in Figure 3.14. Although the proposed framework reuses the assumption $A(p)$ of the system before evolving 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 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 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, the strengthening of a candidate assumption is defined as follows.

Definition 5.2.1 *(Candidate assumption strengthening). Let cex be a counterexample returned by the step* 1 *such that* $cex \in L(A_i)$ *but* $cex \notin U$*, where* U *is the language of the target assumption being learned. The strengthening* A_i *means that the counterexample cex should be removed from* $L(A_i)$*.*

If 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 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, the weakening of a candidate assumption is defined as follows.

Definition 5.2.2 *(Candidate assumption weakening). Let cex be a counterexample returned by the step* 2 *such that* $cex \in U$ *but* $cex \notin L(A_i)$*. The weakening* A_i *means that the counterexample cex should be added to* $L(A_i)$*.*

Example 5.2.1 *Figure 5.7 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 5.6.*

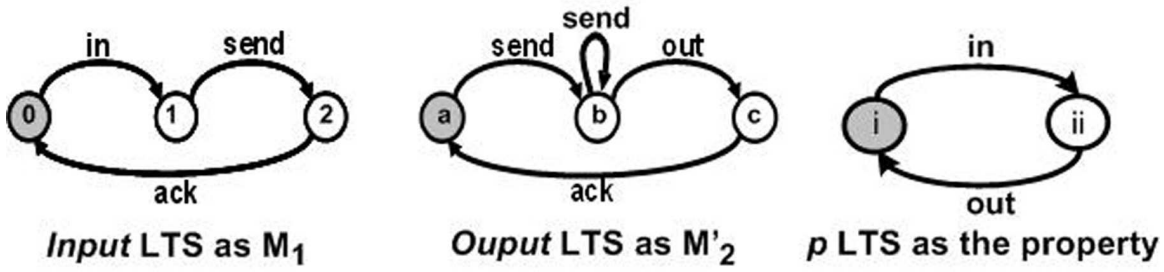


Figure 5.6: An evolved CBS including the model M_1, the evolved model M_2', and the required property.

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

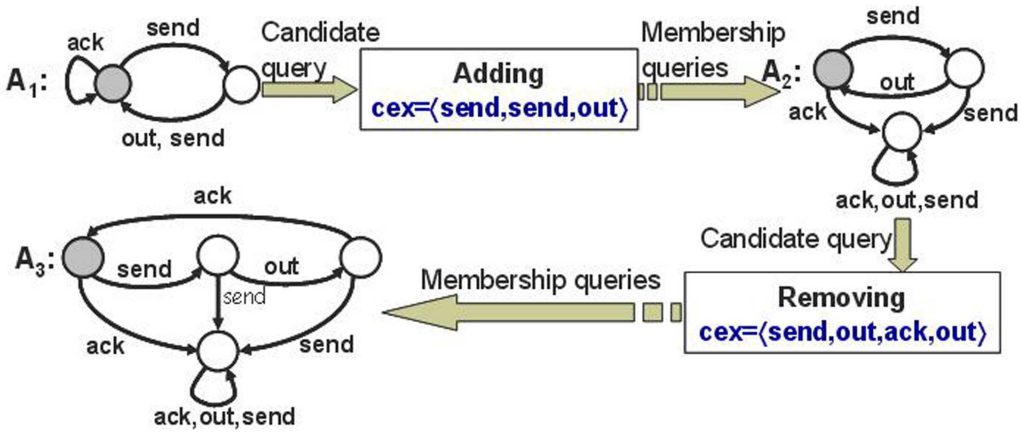


Figure 5.7: The meaning of the strengthening and weakening of the generated candidate assumptions.

presented in Figure 3.14. 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 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 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 3.14.

5.2.2 A Minimized Assumption Regeneration Method

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 4.1 of Chapter 4. In this method, if the current assumption is too strong to be satisfied by the evolved model of the evolved component, 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 5.1 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 4.1 of Chapter 4, 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 5 of Chapter 4, the queue has to hold an exponentially growing of the number of the observation tables. This makes the method unpractical for large-scale systems because it consumed too much memory. For 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 model presented in Section 5.1, we guarantee that the new minimal assumption can be obtained directly from the strong assumption.

Algorithm 9 presents the proposed algorithm for the new assumption regeneration by improving the algorithm for minimized assumption generation presented in Algorithm 5. Recall the proposed framework presented in Subsection 5.1.1, 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 $A(p)$ is too strong to be satisfied by M_2' in the context of cex, the minimized assumption regeneration method presented in Algorithm 9 is applied to regenerate 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 $update(OT_0)$ (line 4). The remaining of the proposed algorithm is the same as the algorithm presented in Algorithm 5. 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.

5.3 Small Experiment

In order to evaluate the effectiveness of the proposed framework for assume-guarantee verification of evolving CBS presented in Section 5.1.1, we have implemented the L*-based assumption generation method proposed in [10] (the AG tool shown in Section 4.3 of Chapter 4) and the proposed assumption regeneration method (called AR tool) in the Objective Caml (OCaml) functional programming language [31]. Figure 5.8 shows the architecture of the implemented AR tool and an example which illustrates how to use the tool. The inputs of this tool are a model M_1 of the fixed framework, an evolved model M_2' of the model M_2, an assumption $A(p)$ of the 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 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 5.8. The correctness of A_{new} also is checked by using the LTSA tool.

The concurrent system Sender/Receiver (Evolved channel) illustrated in Figure 5.6 and two evolved versions (i.e., Sender/Receiver (Two channels) and Sender/Receiver (Evolved two channels)) 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

Algorithm 9 Minimized assumption regeneration.

Input: $M_1, M_2', p, OT_{old} = (S_{old}, E_{old}, T_{old})$: existing 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

```
1: Initially, q = empty {q is an empty queue}
2: OT_0 = OT_old {S_0 = S_old, E_0 = E_old, T_0 = T_old}
3: add the suffix e of the counterexample cex to E_0
4: OT_0 = update(OT_0)
5: q.put(OT_0)
6: while q ≠ empty do
7:   OT_i = q.get() {getting OT_i from the top of q}
8:   if OT_i contains "?" value then
9:     for each instance OT of OT_i do
10:      q.put(OT) {putting OT into q}
11:    end for
12:  else
13:    if OT_i is not closed then
14:      OT = make_closed(OT_i)
15:      q.put(OT)
16:    else
17:      construct a candidate DFA A_i from the closed OT_i
18:      if Step1(A_i) fails with cex then
19:        add the suffix e of the counterexample cex to E_i
20:        OT = update(OT_i)
21:        q.put(OT)
22:      else
23:        if Step2(A_i) fails with cex then
24:          if cex witnesses violation of p then
25:            return cex
26:          else
27:            add the suffix e of the counterexample cex to E_i
28:            OT = update(OT_i)
29:            q.put(OT)
30:          end if
31:        else
32:          return A_i
33:        end if
34:      end if
35:    end if
36:  end if
37: end while
```

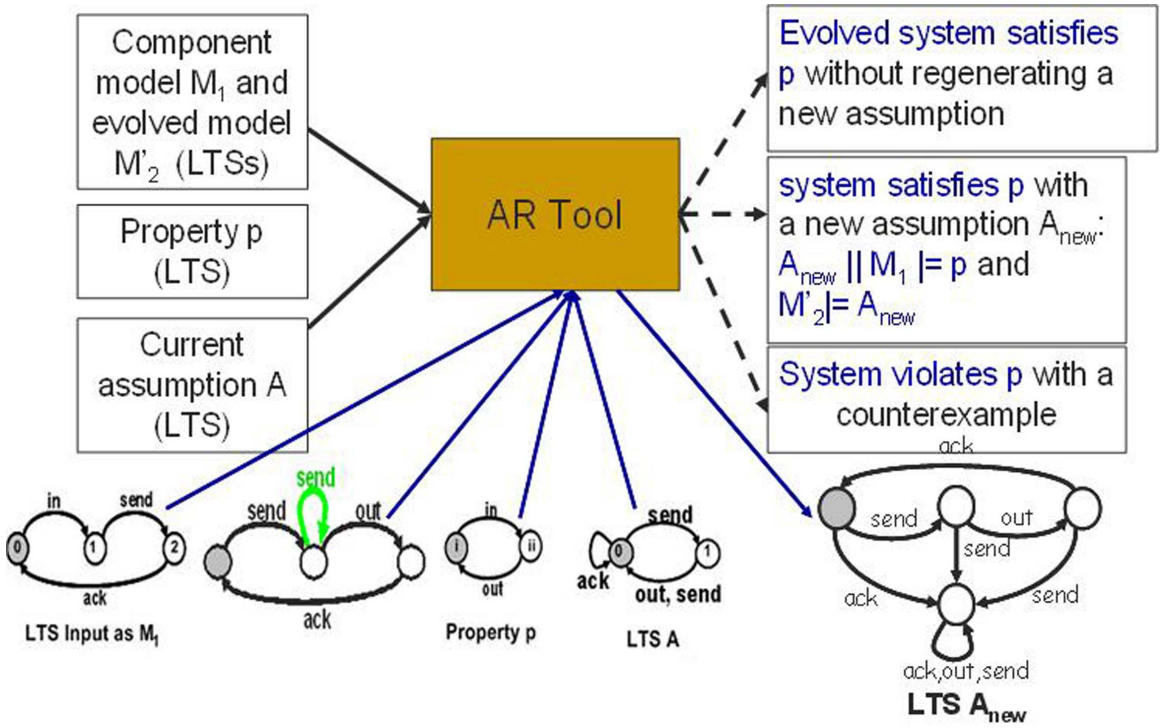


Figure 5.8: The architecture of the AR tool and an example.

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 5.1 shows experimental results for this purpose. In our method, for the first and second evolved systems (Sender/Receiver (Evolved channel) and Sender/Receiver (Two channels)), 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 (Sender/Receiver (Evolved two channels)), the current assumption is an actual assumption of this system so the system is verified without regenerating a new assumption.

The implemented tool and the illustrative systems which are used in our experimental results is available at the site [45].

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 whether $A_{new}(p)$ satisfies the compositional rule (i.e., $\langle A_{new}(p)\rangle$ 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.

With regard to the overhead of the proposed framework, we have verified some CBS examples directly by using the model checker named LTSA. Based on the checking time

Table 5.1: Experimental results

System	Sys. Size	Current AG Method			Proposed AR Method		
		Membership queries	Candidate assumptions	Conjectures	Membership queries	Candidate assumption	Conjectures
Sender/Receiver (Evolved Channel)	18	80	4	6	56	2	3
Sender/Receiver (Two channels)	75	294	6	8	210	3	4
Sender/Receiver (Evolved two channels)	75	294	6	8	0	0	0

evaluated by LTSA, the overheads do not increase so much for our approach. The reason is as follows. In the case, where the current assumption $A(p)$ is actual assumption of the evolved CBS $M_1 \| M_2'$, the evolved CBS still satisfies the property p without regenerating a new assumption. As a result, we can recheck the evolved CBS faster than model checking directly. In others, our approach generates new assumptions from the current assumption $A(p)$ by using the improved L*. Our approach returns new assumptions and the evolved CBS systems are rechecked successfully. This means that rechecking the evolved CBS via generating a new assumption.

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

Chapter 6

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

This chapter proposes a framework for modular conformance testing and assume-guarantee verification of evolving component-based software at 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 assume-guarantee verification processes.

6.1 Modular Conformance Testing

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 [47, 12, 48], the proposed MCT only performs conformance testing to compare the evolved component with its current 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. By this approach, we can reduce the computational cost for learning the accurate models of the evolved components of the evolved CBS.

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. Suppose that, for each component C_i (i=1,2), we have obtained an accurate model M_i by applying the L* learning algorithm. 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 C_i and M_i satisfy the definition about accurate model defined in Chapter 2. C_2 then is evolved to a new component C_2' by adding some new behaviors to the 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 $v \in (T_{C_2'} \setminus T_{C_2})$ via the VC algorithm. 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 a discrepancy d returned by VC algorithm. The L* performs experiments on the evolved component C_2' and produces a minimized finite automaton representing behavior of this component.

At a higher level, the learning algorithm is an iterative process illustrated in Figure 6.1. At the initial step, we use the current model M_2 as the initial model for learning an accurate model M_2' of the evolved component C_2'. In our research, the component evolution means that adding only some new behaviors to the component C_2 without losing the old behaviors. This means that $L(M_2)$ is a subset of $L(M_2')$. As a result, we guarantee that M_2' can be learned from the entire M_2 directly. 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 discrepancy d that distinguishes the behavior of C_2' from the candidate model M_{2i} 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.

The following is more detailed presentation of the L*-based algorithm for learning an accurate model M_2' of the evolved component C_2'. As mentioned above, before the component C_2 is evolved to C_2', we have obtained an accurate model M_2 of C_2 by applying the L* learning algorithm. Let V_2, W_2, T_2 be the observation table which has been used to construct the model M_2, from now on called the old observation table $(V_{2old}, W_{2old}, T_{2old})$.

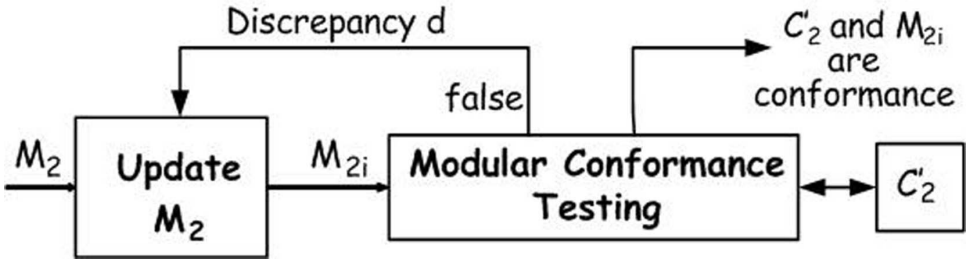


Figure 6.1: The modular conformance testing framework.

In order to obtain the accurate model M_2' of the evolved component C_2', we also use the L* learning algorithm but with the initial observation table $(V_{2old}, W_{2old}, T_{2old})$. With this approach, we can significantly reduce the number of steps involved in learning M_2'.

Algorithm 10 presents the L*-based algorithm for learning an accurate model M_2' of the evolved component C_2' after obtaining the discrepancy d returned by VC algorithm when checking conformance between C_2' and M_2. The discrepancy d witnesses that M_2 is an inaccurate model of the evolved component C_2'. Initially, L* sets the initial observation table (V, W, T) to the old observation table $(V_{2old}, W_{2old}, T_{2old})$ (i.e., L* sets V to V_{2old}, W to W_{2old}, and T to T_{2old}) (line 1). The discrepancy d is analyzed by L* to find all of its suffixes that witness a difference between $L(M_2)$ and $L(C_2')$. After that, for each suffix v' of d, if $v' \notin W$ then v' must be added to W (line 2&3). Consequently, T must be updated by performing the experiments on the evolved component C_2' (line 6). L* then checks whether the observation table (V, W, T) is closed (line 7). If (V, W, T) is not closed, then va is added to V, where $v \in V$ and $a \in \Sigma$ are the elements for which there is no $v' \in V$ (line 8). Because va has been added to V, T must be updated again by performing the experiments on C_2' (line 9). Line 8 and line 9 are repeated until the table (V, W, T) is closed (line 10). When the observation table (V, W, T) is closed, L* checks whether the candidate model corresponding to the closed table (V, W, T) is an accurate model of C_2' by applying the VC algorithm ($conform = VC(V, W, C_2', n_2)$) (line 11). If they are in conformance ($conform == true$) (line 12), a DFA $M_{2i} = \langle Q, \alpha M_{2i}, \delta, q_0, F \rangle$ is constructed (line 13) from the closed table (V, W, T) using the approach described in the definition 3.2.2. L* then returns M_{2i} and terminates (line 14). Otherwise, $conform$ is a discrepancy that distinguishes the behavior of C_2' from the candidate model M_{2i} (line 15). The discrepancy $conform$ then is analyzed by L* to find all of its suffixes that witness a difference between $L(M_{2i})$ and $L(C_2')$. After that, for each suffix v' of $conform$, if $v' \notin W$ then v' must be added to W (line 16&17). When all suffixes of $conform$ have been added to W, L* iterates the entire process by looping around to line 6.

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

Algorithm 10 L*-based algorithm for learning an accurate model M_2' of the evolved component C_2'.

Input: $(V_{2old}, W_{2old}, T_{2old}), d, n_2$: the observation table $(V_{2old}, W_{2old}, T_{2old})$ of the inaccurate model M_2, the discrepancy d, and a known upper bound n_2 on the size of the minimal deterministic automaton modelling the evolved component C_2'.

Output: M_2': an accurate model M_2' of the evolved component C_2'

1: Initially, $V = V_{2old}, W = W_{2old}, T = T_{2old}$
2: **for** each $v' \in suffix(d)$ that is not in W **do**
3: add v' to W
4: **end for**
5: **loop**
6: update T by performing the experiments on the system C_2'.
7: **while** (V, W, T) is *not closed* **do**
8: add va to V to make V closed, where $v \in V$ and $a \in \Sigma$
9: update T by performing the experiments on the system C_2'.
10: **end while**
11: $conform = VC(V, W, C_2', n_2)$
12: **if** $conform == true$ **then**
13: construct a *candidate* DFA M_{2i} from the closed (V, W, T)
14: **return** M_{2i}
15: **else**
16: **for** each $v' \in suffix(conform)$ that is not in W **do**
17: add v' to W
18: **end for**
19: **end if**
20: **end loop**

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., $v \in L(M_1 \| M_2)$). ∎

6.2 Assume-Guarantee 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 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 $M_1 \| M_2'$, we apply the proposed framework presented in Section 5.1.1 of Chapter 5. The framework 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, a counterexample *cex* returned. The counterexample *cex* then is analysed to check whether the evolved system violates the property p in the context of the counterexample *cex* or the assumption $A(p)$ is too strong to be satisfied by M_2'. If the assumption $A(p)$ is too strong, a new assumption $A_{new}(p)$ is regenerated by applying the new assumption regeneration method with the initial assumption $A(p)$. By this approach, the method can reduce the number of the membership queries and the candidate assumptions which are needed to regenerate the new assumption $A_{new}(p)$. In some cases where the current assumptions are actual assumptions of the evolved CBS, these CBS are verified in the fastest way without generating again the new assumptions.

6.3 Framework for MCT and Assume-Guarantee Verification of Evolving CBS

This section proposes an integrated framework for modular conformance testing and assume-guarantee verification of component-based software in the context of the component evolution. In this framework, the model learning (i.e., MCT) and model checking (assume-guarantee verification) are separated into two independent processes. The MCT process first generates an accurate model of the evolved component. The model is seen as the input of the assume-guarantee verification process for rechecking the evolved CBS. With this approach, the assume-guarantee verification method is applied only once. Thus, the computational cost for rechecking the evolved CBS is lower than the computational cost in AMC [12].

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, models of components, 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.

6.3.1 Proposed Framework

We integrate the proposed modular conformance testing and the assume-guarantee verification into a framework for rechecking component-based software in the context of component evolution. This framework is illustrated in Figure 6.2. 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 an updated accurate model M_2' of C_2'.

2. Checking whether the evolved system $M_1 \| M_2'$ satisfies the property p by applying the assume-guarantee 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 $M_1 \| M_2'$ still satisfies p. Otherwise, it returns a counterexample cex to witness this fact.

3. Further analysis is required to identify 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 if 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'.

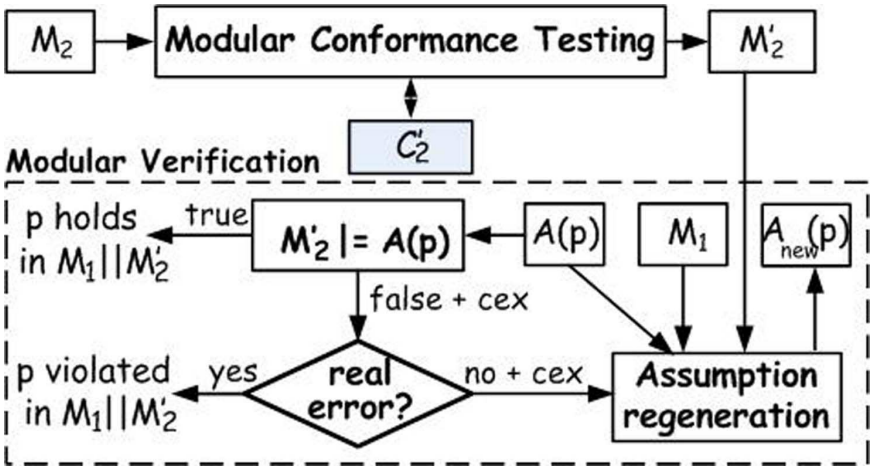


Figure 6.2: The proposed framework for MCT and modular verification of evolving CBS.

Though the proposed framework considers the simple case where the CBS only consists of two components C_1 and C_2, we can generalize it for a larger CBS containing n-components $C_1, C_2, ..., C_n$ $(n \geq 2)$. Let $C_1, C_2, ..., C_i$ $(i \geq 1)$ be fixed components. This means that these components do not allows to be evolved. In order to apply the proposed framework for the CBS, we can consider the CBS as a software system which contains two compositional components, i.e., $C_1 \| C_2 \| ... \| C_i$ and $C_{i+1} \| C_{i+2} \| ... \| C_n$. With the approach, the framework for the CBS consists of the similarly steps as described above because we only focus on the evolved compositional component $C_{i+1} \| C_{i+2} \| ... \| C_n$..

6.3.2 An Example

Figure 6.3 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 3.7 of Chapter 3 that is strong enough for M_1 to satisfy p but weak enough to be discharged by M_2.

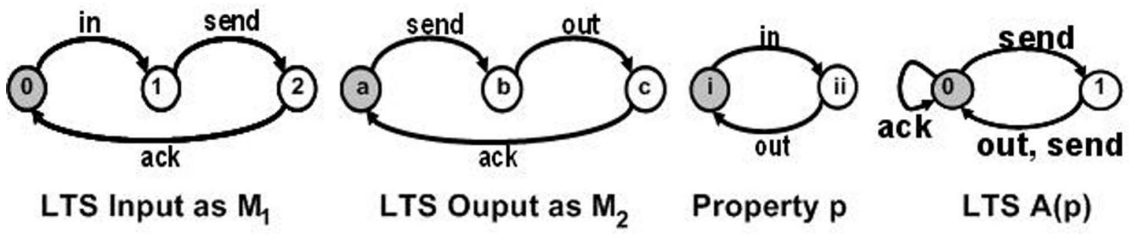
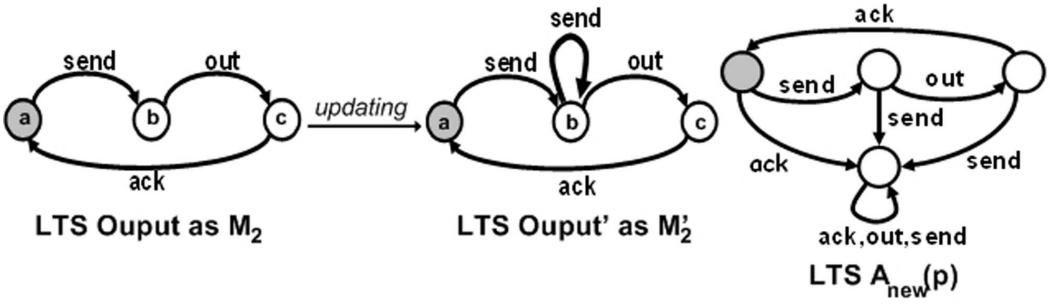


Figure 6.3: Models of the components, order property and assumption $A(p)$ of the illustrative system.

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 6.4. In order to recheck the evolved compositional system $M_1 \| M_2'$, the proposed framework only checks the formula $\langle true \rangle \ M_2' \ \langle A(p) \rangle$. 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 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 generated again. For this purpose, the framework reuses the assumption $A(p)$ as the initial assumption and applies the improved L* learning algorithm showed in Algorithm 4 to generate again the new assumption $A_{new}(p)$ illustrated in Figure 6.4.



LTS Ouput as M_2 LTS Ouput' as M_2' LTS $A_{new}(p)$

Figure 6.4: The updated model and the regenerated assumption $A_{new}(p)$.

Chapter 7

Experiment and Evaluation

This chapter presents three component-based software (CBS): automobile cruise control system, gas oven control system, and banking subsystem and experimental results obtained by applying the proposed approaches for these systems. These experiments are not only to show the practical usefulness of our proposed approaches but also to present how to generalize the proposed approaches for larger CBS (i.e., CBS containing more than two components).

7.1 An Automobile Cruise Control System

The automobile cruise control system (ACCS) has been recognized as a control system for modern vehicles. It is also used popularly in software engineering community for several purposes. This system is taken from the text book by J. Magee and J. Kramer [13].

7.1.1 Description of ACCS

The function of ACCS is to accurately maintain the driver's desired set speed, without intervention from the driver, by actuating the throttle-accelerator pedal linkage. The ACCS has the following requirements. It is controlled by three buttons: *on*, off, and *resume* (Figure 7.1). When the engine is running and the *on* button is pressed, the automobile cruise control system records the current speed and maintains the speed of the car at the recorded setting. When the *accelerator*, *brake* or off is pressed, the system disengages but retains the speed setting. If *resume* button is pressed, the system accelerates or de-accelerates the car back to the previously-recorded speed.

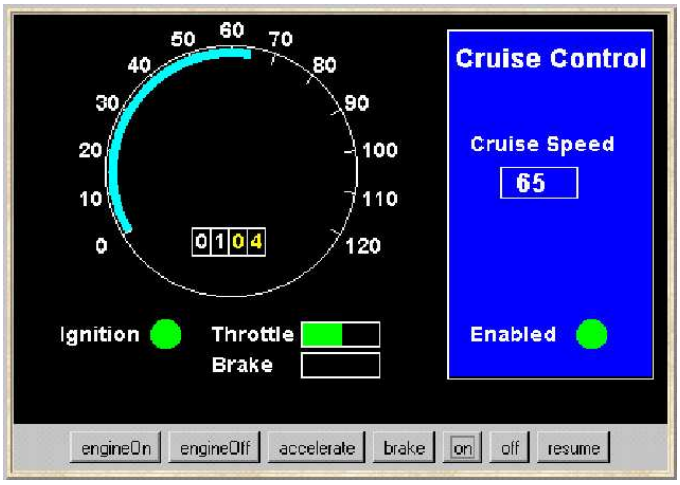


Figure 7.1: Automobile cruise control system.

7.1.2 Structure of ACCS

Structure diagram and observable actions for the ACCS shown in Figure 7.2 can be produced using the following design activities:

- Identify the main observable actions and interactions of the system.
- Identify and define the main components of the system.
- Identify the main properties of the system.
- Structure the components into a structure diagram.

The main internal control for the system is provided by two components: the cruise controller and the speed control. The interface to the external sensors and actuators is provided by the other three components: sensor scan, input speed, and throttle. The cruise controller receives the buttons, brake, accelerator and engine events from the sensor scan. The input speed component monitors the speed when the engine is switched on and provides the current speed readings to the speed control. Depending on the circumstances, the cruise controller triggers clear or record the speed, and enable or disable the speed control. The speed control then sets the throttle accordingly. In this way, the sensor scan encapsulates (information hiding) the periodic process of scanning sensors, the cruise controller encapsulates the decision as to when speed maintenance is activated, and the speed control encapsulates how to record and maintain speed.

With regard to the required properties, the behavior of the system can be checked using the particular scenarios as follows:

- Is the system enabled after the engine is switched on and the *on* button is pressed?

- Is the system disabled when the *brake* is pressed?
- Is the system enabled when the *resume* button is pressed?
- Is the system disabled when the engine is switched off?

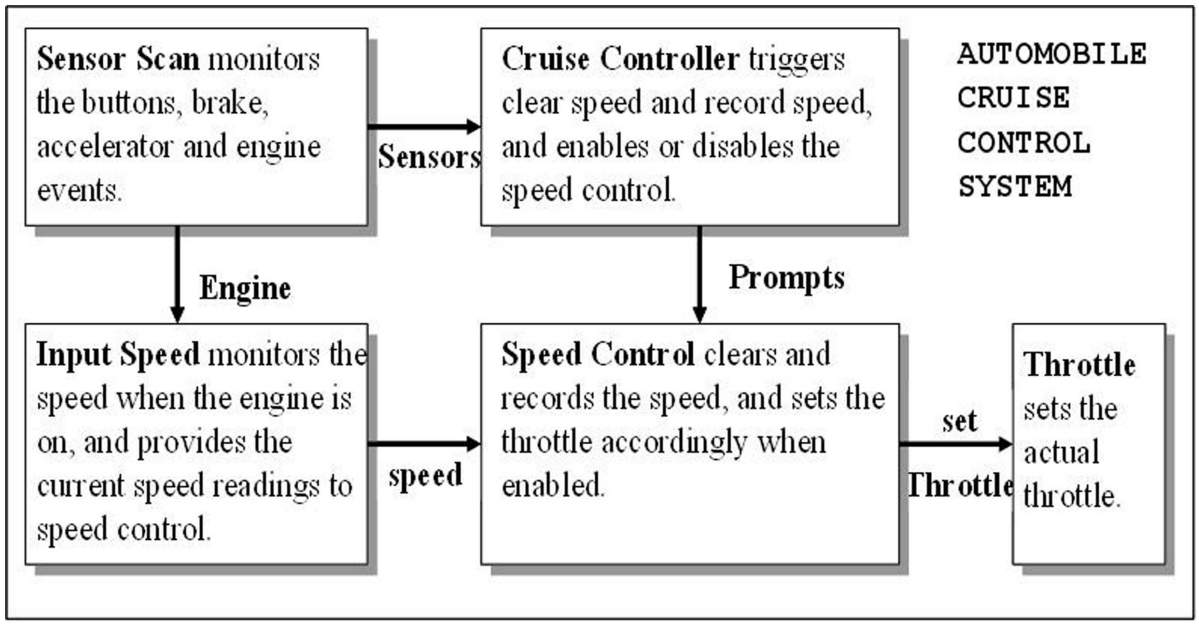


Figure 7.2: Structure diagram and observable actions of ACCS.

7.1.3 Design Models of Components

Each component is defined in Figure 7.3 by FSP as a process-algebra style notation with LTS semantics. The sensors are repeatedly scanned; the input speed is repeatedly monitored when the engine is on; and, when the throttle is set, the car "zooms" off. Speed control is initially disabled. It clears and records the current speed setting and, when it is enabled, it sets the throttle according to the current speed and the recorded speed. The behavior of the cruise controller becomes active. When active, pressing the *on* button triggers the recording of the current speed and enables the speed control. The system is then cruising. Pressing the *on* button again triggers the recording of the new current speed and the system remains cruising. Pressing the off button, *brake* or *accelerator* disables the speed control and sets the system to standby. Switching the engine off at any time makes the system inactive.

```
SENSORSCAN = ({Sensors} -> SENSORSCAN).

INPUTSPEED   = (engineOn -> CHECKSPEED),
CHECKSPEED = (speed -> CHECKSPEED
                |engineOff -> INPUTSPEED ).

THROTTLE =(setThrottle -> zoom -> THROTTLE).

SPEEDCONTROL = DISABLED,
DISABLED  = ({speed,clearSpeed,recordSpeed}->DISABLED
             | enableControl -> ENABLED ),
ENABLED   = ( speed -> setThrottle -> ENABLED
             | {recordSpeed,enableControl} -> ENABLED
             | disableControl -> DISABLED ).

CRUISECONTROLLER = INACTIVE,
INACTIVE = (engineOn -> clearSpeed -> ACTIVE),
ACTIVE   = (engineOff -> INACTIVE
          | on->recordSpeed->enableControl->CRUISING ),
CRUISING = (engineOff -> disableControl-> INACTIVE
           | { off,brake,accelerator} -> disableControl
             -> STANDBY
           |on->recordSpeed->enableControl->CRUISING ),
STANDBY  = (engineOff -> INACTIVE
           | resume -> enableControl -> CRUISING
           | on->recordSpeed->enableControl->CRUISING ).
```

Figure 7.3: Design models of the components for ACCS.

7.1.4 Safety Properties

A safety property required of the ACCS named CRUISESAFETY is shown in Figure 7.4. This property states that if the system is enable by pressing the *on* or *resume* buttons, then pressing the off button, the *brake* or the *accelerator* should result in the system being disabled.

The described CRUISESAFETY safety property does not include a check on the engine status. This must now be included to form an improved safety property named IMPROVEDSAFETY show in Figure 7.5. It is clear that control should be disabled when the engine is switched off.

7.1.5 Assume-Guarantee Verification of ACCS

Checking of ACCS

```
property CRUISESAFETY =
 ({off, accelerator, brake, disableControl} -> CRUISESAFETY
 |{on,resume} -> SAFETYCHECK ),
SAFETYCHECK =
 ({on, resume} -> SAFETYCHECK
 |{off, accelerator, brake} -> SAFETYACTION
 |disableControl -> CRUISESAFETY ),
SAFETYACTION = (disableControl -> CRUISESAFETY).
```

Figure 7.4: A required safety property of ACCS.

```
property IMPROVEDSAFETY =
 ({off, accelerator, brake, disableControl, engineOff} -> IMPROVEDSAFETY
 |{on,resume} -> SAFETYCHECK ),
SAFETYCHECK =
 ({on, resume} -> SAFETYCHECK
 |{off, accelerator, brake, engineOff} -> SAFETYACTION
 |disableControl -> CRUISESAFETY ),
SAFETYACTION = (disableControl -> IMPROVEDSAFETY).
```

Figure 7.5: An improved safety property of ACCS.

At first, we have to verify whether the ACCS before evolving satisfies the required properties, i.e., CRUISESAFETY denoted p_1 and IMPROVEDSAFETY denoted p_2. In order to apply the assume-guarantee verification approach for checking the system, we consider ACCS as a CBS containing two compositional components: M_1 and M_2, where M_1 is a composition of SPEEDCONTROL and THROTTLE shown in Figure 7.3 (i.e., M_1 = SPEEDCONTROL $\|$ THROTTLE) and M_2 is a composition of SENSORSCAN, INPUTSPEED, and CRUISECONTROLLER shown in Figure 7.3 (i.e., M_2 = SENSORSCAN $\|$ INPUTSPEED $\|$ CRUISECONTROLLER). We verify the system for each property separately. In this case, the size of this system, which is the product of the sizes of the software components and the size of the required property, is 720 states.

Consider the CRUISESAFETY safety property shown in Figure 7.4, safety analysis using the assume-guarantee verification approach proposed in [10] (via the AG tool described in Section 4.3 of Chapter 4) verifies that the property violates in the ACCS with a counterexample *engineOn clearSpeed on recordSpeed enableControl engineOff disableControl*. It violates because the property does not include a check on the engine status.

With regard to the IMPROVEDSAFETY safety property (p_2 safety property) shown in Figure 7.5, the assume-guarantee verification approach proposed in [10] verifies that the property is not violated in the ACCS with the generated assumption $A(p_2)$ shown in Figure 7.6. We generate the assumption with 144 required membership queries, 3 generated candidate assumption, and 4 required conjectures.

```
A(p2) = ({speed, recordSpeed, off, engineOff, enableControl, disableControl,
          clearSpeed, brake, accelerator} -> A(p2) | {resume, on} -> A1),
A1 = (disableControl->A(p2)
      |{speed, resume, recordSpeed, on, enableControl, clearSpeed} -> A1
      | {off, engineOff, brake, accelerator} -> A2),
A2 = (disableControl->A(p2)
      | {speed, recordSpeed, enableControl, clearSpeed} -> A2).
```

Figure 7.6: The generated assumption $A(p_2)$ of ACCS.

Checking of Evolving ACCS

The design model of the CRUISECONTROLLER component shown in Figure 7.3 is evolved to EVOLVEDCRUISECONTROLLER model shown in Figure 7.15 to ensure that the system should be inactive when the engine is switched off at any time. The EVOLVED-CRUISECONTROLLER model is obtained by adding a new behavior engineOff→INACTIVE (the bold typeface).

```
EVOLVEDCRUISECONTROLLER = INACTIVE,
INACTIVE = (engineOn -> clearSpeed -> ACTIVE),
ACTIVE   = (engineOff -> INACTIVE
            | on->recordSpeed->enableControl->CRUISING ),
CRUISING = (engineOff -> disableControl-> INACTIVE
            | engineOff -> INACTIVE
            | { off,brake,accelerator} -> disableControl
              -> STANDBY
            |on->recordSpeed->enableControl->CRUISING ),
STANDBY  = (engineOff -> INACTIVE
            | resume -> enableControl -> CRUISING
            | on->recordSpeed->enableControl->CRUISING ).
```

Figure 7.7: Evolved model of the CRUISECONTROLLER component.

In order to recheck the evolved ACCS $M_1 \| M_2'$ (M_2' = SENSORSCAN $\|$ INPUTSPEED

$\|$ EVOLVEDCRUISECONTROLLER), our framework presented in Section 5.1.1 of Chapter 5 only checks the formula $\langle true \rangle\ M_2'\ \langle A(p_2) \rangle$, where $A(p_2)$ is an assumption of ACCS before evolving (shown in Figure 7.6). This checking returns $false$ and the counterexample analysis implies that $A(p_2)$ is too strong to be satisfied by M_2'. A new assumption $A_{new}(p_2)$ for the evolved ACCS $M_1 \| M_2'$ must be regenerated. For the purpose, the new assumption regeneration method reuses the assumption $A(p_2)$ to regenerate the new assumption $A_{new}(p_2)$ shown in Figure 7.8. In the assumption generation method proposed in [10], for the same goal, the method has used 360 required membership queries, 5 generated candidate assumption, and 8 required conjectures to generate $A_{new}(p_2)$. Our method generates $A_{new}(p_2)$ at much lower computational cost with 216 required membership queries, 2 generated candidate assumption, and 4 required conjectures.

```
A_new(p2) = ({speed, recordSpeed, off, engineOff, enableControl,
              disableControl, clearSpeed, brake, accelerator} -> A_new(p2)
            | {resume, on}->A1),
A1 = (disableControl -> A_new(p2)
      | {speed, resume, recordSpeed, on, clearSpeed} -> A1
      | {off, engineOff, brake, accelerator} -> A2 | enableControl -> A4),
A2 = (disableControl -> A_new(p2) | {speed, recordSpeed, clearSpeed} -> A2
      | enableControl -> A3),
A3 = ({disableControl, clearSpeed} -> A_new(p2)
      | {speed, recordSpeed, enableControl} -> A3),
A4 = ({disableControl, clearSpeed} -> A_new(p2)
      | {off, engineOff, brake, accelerator} -> A3
      | {speed, resume, recordSpeed, on, enableControl} -> A4).
```

Figure 7.8: The new assumption $A_{new}(p_2)$ regenerated by the proposed framework.

7.1.6 Discussion

As mentioned above, the automobile cruise control system is used popularly in software engineering community for several purposes. In fact, there are many versions of this system. In this experiment, we use one of them presented in the text book by J. Magee and J. Kramer [13].

The obtained experimental result for rechecking the evolving ACCS is a nice example to show the practical usefulness of the proposed framework presented in Chapter 5. In this case, the current assumption $A(p_2)$ is not actual assumption of the evolved ACCS. It is too strong to be satisfied by M_2'. The new assumption $A_{new}(p_2)$ is regenerated with a lower computational cost. By reusing the entire current assumption $A(p_2)$, we reduce

144 required membership queries, 3 generated candidate assumption, and 4 required conjectures for regenerating the new assumption.

However, the reduced numbers are quite small. The reason is that the evolution of the CRUISECONTROLLER component is too small, i.e., adding engineOff→INACTIVE in to the model of the component. Even if the reduced computational cost is small, our approach is applied many times during the software life-cycle because the evolution often occurs at any time in any phases of the software development process. As a result, the obtained benefit from our approach becomes larger and larger.

7.2 A Gas Oven Control System (GOCS)

7.2.1 Description of GOCS

A gas oven has recognized as a control system that it can be remote-controlled at home or outside using mobile devices. This remote control system may be useful for turning off the gas oven when we forgot to turn it off at going outside or when we want to control the oven remotely at home. However, it is unsafe to control a gas oven remotely since we cannot check its status such as gas leakage and inflammable materials on it. Therefore, for safety, we need some complementary devices such as a flame detection sensor, which can be monitoring the status of the gas oven. Figure 7.9 shows the overall structure of the gas oven that can be remote-controlled. Now, is the gas oven system safety? [50].

Figure 7.9: An example of remote-controlled gas oven system.

7.2.2 Structure of GOCS

Figure 7.10 represents a structure diagram and observable actions of the remote-controlled gas oven system. For simplicity, we abstractly describe only core components. The gas

oven system is composed of a gas oven controller, a valve controller, a flame sensor, a communication media, and mobile devices.

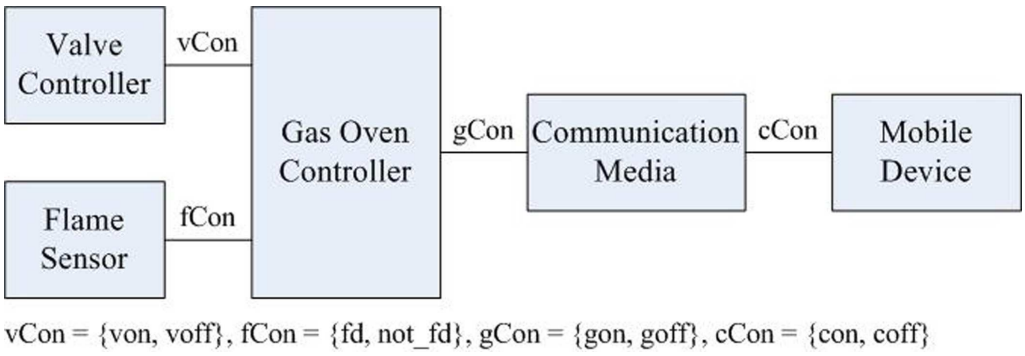


Figure 7.10: A structure diagram and observable actions of the remote-controlled gas oven system.

7.2.3 Design Models of Components

Each component of the structure diagram is described by FSP. Figure 7.11 shows the LTS models of the remote-controlled gas oven system. Communicating channels between components such as vCon and cCon are described by shared labels. In a LTS, all the states are considered as accepting states. The parallel composition of two LTS models, denoted by $P\|Q$, models the synchronized behavior of shared labels. Local events behave independently while the shared labels should be synchronized.

```
FLAMESENSOR = ({fd, not_fd} -> FLAMESENSOR).
VALVECONTROLLER = (von -> voff -> VALVECONTROLLER).
COMMUNICATIONMEDIA =
                    (con -> gon -> COMMUNICATIONMEDIA
                    | coff->goff->COMMUNICATIONMEDIA).
GASOVENCONTROLLER = (gon ->  G1),
G1 = (goff -> GASOVENCONTROLLER | von -> G2),
G2 = (fd -> G2 | not_fd -> G3 | goff -> G4),
G3 = (voff -> G1 | goff -> GASOVENCONTROLLER),
G4 = (voff -> GASOVENCONTROLLER).
MOBILEDEVICE = (con -> coff -> MOBILEDEVICE).
```

Figure 7.11: Design models of the components for GOCS.

7.2.4 Safety Property

A safety property required of the GOCS named GOCSSAFETY is shown in Figure. This property states that after a gas valve is opened, it should be closed.

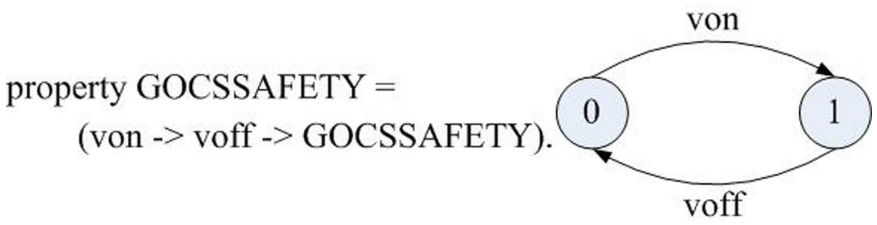


Figure 7.12: A required safety property of GOCS.

7.2.5 Assume-Guarantee Verification of GOCS

Checking of GOCS

In order to apply the assume-guarantee verification approach for verifying the GOCS such that whether the system satisfies the GOCSSAFETY safety property denoted p, we consider GOCS as a CBS containing two compositional components: M_1 and M_2, where M_1 is a composition of GASOVENCONTROLLER and MOBILEDEVICE shown in Figure 7.11 (i.e., M_1 = GASOVENCONTROLLER $\|$ MOBILEDEVICE) and M_2 is a composition of FLAMESENSOR, VALVECONTROLLER, and COMMUNICATIONMEDIA shown in Figure 7.11 (i.e., M_2 = FLAMESENSOR $\|$ VALVECONTROLLER $\|$ COMMUNICATIONMEDIA).

The assume-guarantee verification approach proposed in [10] (via the AG tool described in Section 4.3 of Chapter 4) verifies that the GOCSSAFETY safety property is not violated in the GOCS with the generated assumption $A(p)$ shown in Figure 7.13. The generated assumption $A(p)$ has 14 states and 110 transitions.

For the same purpose, the proposed minimized assumption generation method presented in Section 4.1 of Chapter 4 generates a minimal assumption $A_m(p)$ shown in Figure 7.14. The generated minimal assumption $A_m(p)$ has 6 states and 26 transitions.

Checking of Evolving GOCS

The design model of the COMMUNICATIONMEDIA component shown in Figure 7.11 is evolved to EVOLVEDCOMMUNICATIONMEDIA model shown in Figure 7.15 to ensure that the gas valve is opened after the event "con" be occurred. The EVOLVEDCOMMUNICATIONMEDIA model is obtained by adding a new behavior con→von→EVOLVEDCOMMUNICAT (the bold typeface).

```
A(p) = ({von, voff, not_fd, fd, goff, coff} -> A(p) | con -> A1 | gon -> A2),
A1 = ({von, voff, not_fd, fd, goff, con, coff} -> A(p) | gon -> A3),
A2 = ({voff, not_fd, fd, gon, goff, coff} -> A(p) | con -> A3 | von -> A4),
A3 = ({voff, not_fd, fd, gon, con} -> A(p) | goff -> A1 | coff -> A2
      | von -> A5),
A4 = (goff -> A7 | not_fd -> A8 | {von, voff, gon, coff} -> A(p)
      | fd -> A4 | con -> A5),
A5 = (not_fd -> A9 | {von, voff, gon, con} -> A(p) | coff -> A4
      | fd -> A5 | goff -> A6),
A6 = (coff -> A7 | {von, not_fd, gon, goff, fd, con} -> A(p) | voff -> A1),
A7 = ({von, voff, not_fd, gon, goff, fd, coff} -> A(p) | con -> A6),
A8 = (con -> A9 | {von, not_fd, gon, fd, coff} -> A(p) | goff -> A10
      | voff -> A2),
A9 = (coff -> A8 | {von, not_fd, gon, fd, con} -> A(p) | goff -> A11
      | voff -> A3),
A10 = ({von, voff, not_fd, goff, fd, coff} -> A(p) | con -> A11
      | gon -> A12),
A11 = ({von, voff, not_fd, goff, fd, con} -> A(p) | coff -> A10
      | gon -> A13),
A12 = ({voff, not_fd, gon, fd, coff} -> A(p) | goff -> A10 | con -> A13),
A13 = ({voff, not_fd, gon, fd, con} -> A(p) | goff -> A11 | coff -> A12).
```

Figure 7.13: The generated assumption $A(p)$ of GOCS.

In order to recheck the evolved GOCS $M_1 \| M_2'$ (M_2' = FLAMESENSOR ‖ VALVE-CONTROLLER ‖ EVOLVEDCOMMUNICATIONMEDIA), our framework presented in Section 5.1.1 of Chapter 5 only checks the formula $\langle true \rangle\ M_2'\ \langle A(p) \rangle$, where $A(p)$ is an assumption of GOCS before evolving (shown in Figure 7.13). This checking returns *true*. This means that the evolved GOCS still satisfies the required safety property without regenerating a new assumption.

7.2.6 Discussion

The gas oven control system is a nice example to show the effectiveness of the proposed minimized assumption generation method presented in Chapter 4. Our obtained experimental result imply that the generated minimal assumption $A_m(p)$ (6 states and 26 transitions) has smaller size and number of transitions than the generated one (14 states and 110 transitions) by the method proposed in [10]. The minimal assumption is effective

$A_m(p)$ = ({not_fd, fd} -> $A_m(p)$ | coff -> A1 | con -> A2 | von -> A3),
A1 = (goff -> $A_m(p)$ | {not_fd, fd} -> A1 | von -> A4),
A2 = (gon -> $A_m(p)$ | {not_fd, fd} -> A2 | von -> A5),
A3 = (voff -> $A_m(p)$ | {not_fd, fd} -> A3 | coff -> A4 | con -> A5),
A4 = (voff -> A1 | goff -> A3 | {not_fd, fd} -> A4),
A5 = (voff -> A2 | gon -> A3 | {not_fd, fd} -> A5).

Figure 7.14: The generated minimal assumption $A_m(p)$ of GOCS.

EVOLVEDCOMMUNICATIONMEDIA =
(con -> von -> EVOLVEDCOMMUNICATIONMEDIA
| con -> gon -> EVOLVEDCOMMUNICATIONMEDIA
| coff -> goff -> EVOLVEDCOMMUNICATIONMEDIA).

Figure 7.15: Evolved model of the COMMUNICATIONMEDIA component.

for rechecking the systems with a lower computational cost.

In order to recheck the evolved GOCS, the proposed framework for verification of evolving CBS presented in Chapter 5 only checks the formula $\langle true \rangle\ M_2''\ \langle A(p) \rangle$. The result of this checking is *true*. This means that the evolved GOCS still satisfies the GOCSSAFETY safety property without regenerating a new assumption. In this case, our approach can recheck the evolved system in the fastest way.

However, the current assumption $A(p)$ is still strong enough to be satisfied by the evolved model M_2' because the evolution of the COMMUNICATIONMEDIA component is too small by adding a new behavior con→von→EVOLVEDCOMMUNICATIONMEDIA into the model of the component. If the evolution will be bigger, the current assumption will be too strong for the evolved model to satisfy. In this case, a new assumption must be regenerated and this example will be better to show the effectiveness of both proposed approach (minimized assumption generation and new assumption regeneration).

7.3 A Banking Subsystem (BS)

7.3.1 Description of BS

Consider a subsystem of a banking system which contains two components: Deposit and Withdraw. Figure 7.16 shows the overall structure of the banking subsystem. Two requests for withdrawal and deposit from the same account comes to a bank from two different ATMs (or ATM and Banker, etc.). The request for deposit requires via the

RequireDepo action. After completing the deposit, it releases the shared bank account via the ReleaseDepo action. Similarity with the request for withdrawal, it requires and releases the shared bank account via RequireWithd action and ReleaseWithd action respectively. The bank account is called as a critical section (shared data) of the system. In the banking system, one of the key issues is to ensure that only one component accesses the bank account at any given time (the mutual exclusion problem).

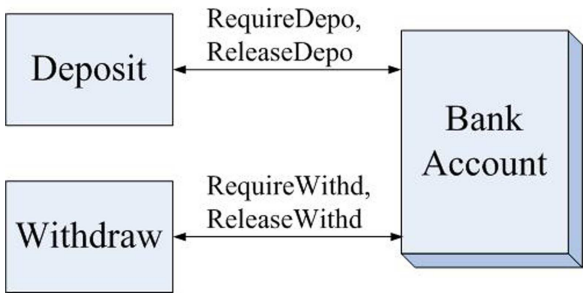


Figure 7.16: A banking subsystem.

7.3.2 Peterson's Algorithm

One of the popular solutions to deal with the the mutual exclusion is the Peterson's algorithm [51]. In the work, Peterson presents an elegant algorithm to solve the mutual exclusion problem for two processes, A and B, which use only shared memory for communication. There are three shared variables, x, y and *turn*. Both x and y are initially set to zero. The variable turn can hold one of two possible values A and B (we use these instead of 0 and 1 for clarity of presentation), and is initially set to A. The algorithm is shown in Figure 7.17.

Process A:	**Process B:**
1. x ← 1	1. y ← 1
2. turn ← B	2. turn ← A
3. while y = 1 and turn = B do;	3. while x = 1 and turn = A do;
4. enter **critical section**	4. enter **critical section**
5. leave **critical section**	5. leave **critical section**
6. x ← 0	6. y ← 0

Figure 7.17: Peterson's algorithm.

7.3.3 Design Models of Components

The described Peterson's algorithm is applied to deal with the the mutual exclusion problem in the banking subsystem. Since every variable can only take one of a finite number of values, the state space of the composition of the models of the components, Deposit and Withdraw, is finite. We have modelled both components as FSPs. The design model of the Deposit component is shown in Figure 7.18, the design model of the Withdraw component in Figure 7.19.

The states are labelled by four-tuples, which represent the shared variables as well as the program counter. The tuples are of the form $\langle turn; x; y; pc \rangle$, where the program counter pc counts the number of steps taken so far in each component. The program counter starts with 0 and is increased with each transition that the component makes. Both models of the described components have the same label for their initial states. Once a component has left it critical section, it returns to the initial state to start a new run.

Since our notion of labeled transition system does not include reading and writing of variables, we model the communication between the models using the semantics of the parallel composition, introduced in Section 2.1 of Chapter 2. Hence each model also contains those transitions which correspond to the writing of shared variables by the respective other model. The alphabet includes actions like $x_e q_1$, which are taken synchronously by both models whenever the model sets x to 1. The actions requireDepo, ReleaseDepo, requireWithd, and releaseWithd indicate that a model enters or leaves its critical section. Since this does not affect the shared variables, the DEPOSIT model does not have any transitions labelled by requireWithd or releaseWithd, and similarly, the WITHDRAW model does not have any transitions labelled by requireDepo or releaseDepo.

7.3.4 Safety Properties

A safety property required of the banking subsystem named ME (mutual exclusion) is shown in Figure 7.20. This property states that only one component accesses the bank account at any given time.

7.3.5 Assume-Guarantee Verification of BS

In order to verify whether the compositional system $DEPOSIT \| WITHDRAW$ satisfies the described ME property, the assume-guarantee verification approach proposed in [10] verifies that the property is not violated in the system with the generated assumption $A(p)$ shown in Figure 7.21. The generated assumption $A(p)$ has 13 states and 102 transitions.

For the same purpose, our proposed minimized assumption generation method gener-

```
DEPOSIT = ({y_eq_0,t_eq_D}->DEPOSIT | y_eq_1->D010 | x_eq_1->D101),
D010 = ({y_eq_1,t_eq_D}->D010 | y_eq_0->DEPOSIT | x_eq_1->D111),
D111 = ({y_eq_1,t_eq_D}->D111 | y_eq_0->D101 | t_eq_W->W112),
D101 = (y_eq_1->D111 | {y_eq_0,t_eq_D}->D101 | t_eq_W->W102),
W102 = (y_eq_0->W102 | t_eq_D->D102 | y_eq_1->W112 | requireDepo->W103),
W103 = (y_eq_0->W103 | y_eq_1->W113 | releaseDepo->W104 | t_eq_D->D103),
W104 = (y_eq_0->W104 | y_eq_1->W114 | x_eq_0->W000 | t_eq_D->D104),
W113 = (y_eq_1->W113 | y_eq_0->W103 | releaseDepo->W114 | t_eq_D->D113),
W000 = (y_eq_0->W000 | y_eq_1->W010 | x_eq_1->W101 | t_eq_D->DEPOSIT),
W114 = (y_eq_1->W114 | y_eq_0->W104 | x_eq_0->W010 | t_eq_D->D114),
W101 = (y_eq_0->W101 | y_eq_1->W111 | t_eq_D->D101 | t_eq_W->W102),
W010 = (y_eq_1->W010 | y_eq_0->W000 | x_eq_1->W111 | t_eq_D->D010),
W111 = (y_eq_1->W111 | y_eq_0->W101 | t_eq_W->W112 | t_eq_D->D111),
W112 = (y_eq_1->W112 | t_eq_D->D112 ),
D102 = ({y_eq_0,t_eq_D}->D102 | y_eq_1->D112 | requireDepo->D103),
D103 = ({y_eq_0,t_eq_D}->D103 | y_eq_1->D113 | releaseDepo->D104),
D112 = ({y_eq_1,t_eq_D}->D112 | y_eq_0->D102 | requireDepo->D113),
D104 = ({y_eq_0,t_eq_D}->D104 | y_eq_1->D114 | x_eq_0->DEPOSIT),
D113 = ({y_eq_1,t_eq_D}->D113 | y_eq_0->D103 | releaseDepo->D114),
D114 = ({y_eq_1,t_eq_D}->D114 | y_eq_0->D104 | x_eq_0->D010).
```

Figure 7.18: Design model of the Deposit component.

ates a minimal assumption $A_m(p)$ shown in Figure 7.22. The generated minimal assumption $A_m(p)$ has 12 states and 48 transitions.

```
WITHDRAW =
        (x_eq_0->WITHDRAW | y_eq_1->D011 | x_eq_1->D100 | t_eq_W->W000),
D011 = (x_eq_0->D011 | t_eq_D->D012 | t_eq_W->W011 | x_eq_1->D111),
D012 = (x_eq_0->D012 | requireWithd->D013 | t_eq_W->W012 | x_eq_1->D112),
D013 = (x_eq_0->D013 | releaseWithd->D014 | t_eq_W->W013 | x_eq_1->D113),
D113 = (x_eq_1->D113 | x_eq_0->D013 | t_eq_W->W113 | releaseWithd->D114),
D014 = (x_eq_0->D014 | t_eq_W->W014 | x_eq_1->D114 | y_eq_0->WITHDRAW),
D114 = (x_eq_1->D114 | x_eq_0->D014 | y_eq_0->D100 | t_eq_W->W114),
W014 = ({x_eq_0,t_eq_W}->W014 | x_eq_1->W114 | y_eq_0->W000),
D100 = (x_eq_1->D100 | x_eq_0->WITHDRAW | y_eq_1->D111 | t_eq_W->W100),
W000 = ({x_eq_0,t_eq_W}->W000 | y_eq_1->W011),
D111 = (x_eq_1->D111 | t_eq_W->W111 | t_eq_D->D112 | x_eq_0->D011),
W011 = ({x_eq_0,t_eq_W}->W011 | t_eq_D->D012 | x_eq_1->W111),
W111 = ({x_eq_1,t_eq_W}->W111 | t_eq_D->D112 | x_eq_0->W011),
D112 = (x_eq_1->D112 | x_eq_0->D012 | t_eq_W->W112),
W112 = ({x_eq_1,t_eq_W}->W112 | x_eq_0->W012 | requireWithd->W113),
W012 = ({x_eq_0,t_eq_W}->W012 | x_eq_1->W112 | requireWithd->W013),
W013 = ({x_eq_0,t_eq_W}->W013 | x_eq_1->W113 | releaseWithd->W014),
W113 = ({x_eq_1,t_eq_W}->W113 | x_eq_0->W013 | releaseWithd->W114),
W114 = ({x_eq_1,t_eq_W}->W114 | x_eq_0->W014 | y_eq_0->W100),
W100 = ({x_eq_1,t_eq_W}->W100 | x_eq_0->W000 | y_eq_1->W111).
```

Figure 7.19: Design model of the Withdraw component.

ME =
(requireDepo->releaseDepo -> ME
 | requireWithd->releaseWithd -> ME).

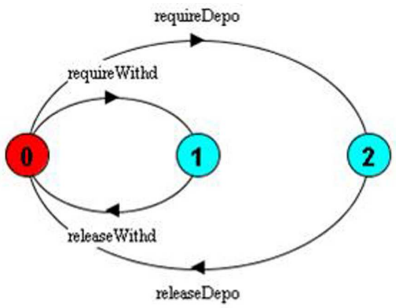


Figure 7.20: A required safety property of the banking subsystem.

```
A(p)  = ({y_eq_0,t_eq_D}->A(p) | requireWithd->A1 | {x_eq_0,t_eq_W}->A6
      | x_eq_1->A7 | y_eq_1->A12),
A1 = (releaseWithd->A(p) | {y_eq_0,t_eq_D}->A1 | x_eq_1->A2 | y_eq_1->A5
      | {x_eq_0,t_eq_W}->A6),
A2 = (releaseWithd->A7 | {y_eq_0,t_eq_D}->A2 | y_eq_1->A3
      | {x_eq_0,x_eq_1}->A6),
A3 = (releaseWithd->A10 | y_eq_0->A2 | {y_eq_1,t_eq_D}->A3 | t_eq_W->A4
      | {x_eq_0,x_eq_1}->A6),
A4 = (releaseWithd->A11 | y_eq_1->A4 | {x_eq_0,x_eq_1,t_eq_W}->A6),
A5 = (y_eq_0->A1 | x_eq_1->A3 | releaseWithd->A12 | {y_eq_1,t_eq_D}->A5
      | {x_eq_0,t_eq_W}->A6),
A6 = ({y_eq_0,y_eq_1,x_eq_0,x_eq_1,t_eq_D,t_eq_W,requireWithd,releaseWithd}
       ->A6),
A7 = ({y_eq_0,t_eq_D}->A7 | t_eq_W->A8 | y_eq_1->A10 | requireWithd->A2
      | {x_eq_0,x_eq_1}->A6),
A8 = ({y_eq_0,t_eq_D}->A8 | y_eq_1->A9 | x_eq_0->A(p)
      | {x_eq_1,t_eq_W}->A6),
A9 = (y_eq_0->A8 | {y_eq_1,t_eq_D}->A9 | x_eq_0->A12
      | {x_eq_1,t_eq_W}->A6),
A10 = (y_eq_0->A7 | {y_eq_1,t_eq_D}->A10 | t_eq_W->A11 | requireWithd->A3
      | {x_eq_0,x_eq_1}->A6),
A11 = (y_eq_0->A8 | t_eq_D->A9 | y_eq_1->A11 | requireWithd->A4
      | {x_eq_0,x_eq_1,t_eq_W}->A6),
A12 = (y_eq_0->A(p) | x_eq_1->A10 | {y_eq_1,t_eq_D}->A12 | requireWithd->A5
      | {x_eq_0,t_eq_W}->A6).
```

Figure 7.21: The generated assumption $A(p)$ of the banking subsystem.

$A_m(p)$ = ({x_eq_0,t_eq_W}->$A_m(p)$ | x_eq_1->A1 | y_eq_1->A2),
A1 = (x_eq_0->$A_m(p)$ | {x_eq_1,t_eq_W}->A1 | y_eq_1->A11),
A2 = (x_eq_1->A11 | {x_eq_0,t_eq_W}->A2 | t_eq_D->A3),
A3 = (t_eq_W->A8 | x_eq_1->A9 | x_eq_0->A3 | requireWithd->A4),
A4 = (x_eq_1->A7 | {x_eq_0,t_eq_W}->A4 | releaseWithd->A5),
A5 = (y_eq_0->$A_m(p)$ | {x_eq_0,t_eq_W}->A5 | x_eq_1->A6),
A6 = (y_eq_0->A1 | x_eq_0->A5 | {x_eq_1,t_eq_W}->A6),
A7 = ({x_eq_1,t_eq_W}->A7 | x_eq_0->A4 | releaseWithd->A6),
A8 = ({x_eq_0,t_eq_W}->A8 | x_eq_1->A10 | requireWithd->A4),
A9 = (x_eq_1->A9 | t_eq_W->A10 | x_eq_0->A3),
A10 = (requireWithd->A7 | x_eq_0->A8 | {x_eq_1,t_eq_W}->A10),
A11 = (t_eq_D->A9 | x_eq_0->A2 | {x_eq_1,t_eq_W}->A11).

Figure 7.22: The generated minimal assumption $A_m(p)$ of of the banking subsystem.

Chapter 8

Related Works

There are many works that have been recently proposed in assume-guarantee verification of component-based systems, by several authors. We focus only on the most recent and closest ones as follows.

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 CBS. 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 improves these works to generate the minimal assumptions in order to reduce the computational cost for rechecking of the CBS. Our work about assume-guarantee verification of evolving CBS at design level is similar to the works proposed in [20, 10, 16, 37, 35]. However, our method differs these works in some key points. Firstly, our work presents a faster assume-guarantee method to verify component-based systems in the context of the component evolution. There is a strong relationship between two design models M_2 and M_2', where M_2' is the evolution of M_2'. For this reason, the proposed method is efficient to change. On the contrary, the component evolution is not mentioned in these works. Secondly, in the proposed method, if the model M_2 is evolved to a new model M_2' and if the formula $\langle true \rangle\ M_2'\ \langle A(p) \rangle$ does not hold, the new assumption $A_{new}(p)$ is regenerated on a faster approach. These works in [20, 10, 16, 37, 35] 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 generation method is run again on the whole evolved system, i.e., the model of the component has to be constructed again; and the assumption about the environment is then regenerated from that model. At source code level, our work is close to the works proposed in [20, 37, 25]. However, these works assume that the models which describe the behaviors of the software components are available and accurate. On the contrary, our work do not assume that. If a model is not accurate for its component, we provide a mechanism for updating an accurate model.

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.

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 component before evolving. In our opinion, adding is enough for the component 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.

Even though the proposed approaches in this dissertation is based on component-based modular model checking, there is a fundamental difference between the conventional modular verification works [38, 39, 40] and our work. Modular verification in the previous works [38, 39, 40] 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 $A(p)$.

Our work relates to many works have been proposed in model checking publish/subscribe systems [41, 42, 43, 44]. The paper in [44] 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 runtime event management and dispatch, and *(2)* components that are specific to the application being modelled. This work has been extended in [41, 42, 43] by the different ways. In particular, [42] uses architectural patterns as an abstraction to carry on, and reuse, formal reasoning on systems whose configuration can dynamically change. [43] presents a compositional reasoning to verify middleware-based software architecture descriptions. [41] embeds the asynchronous communication mechanisms of publish-subscribe infrastructures within Bogor. Our work and [43] share the compositional reasoning approach but we focus on solving a smaller part in the framework that is assumption regeneration in the context of the component evolution.

Peled et al. proposes an approach called black box checking [47] as a way to directly verify a system when its model is not given but a way of conducting experiments is provided. This work has been improved in [48] for grey-box checking. A related idea, called adaptive model checking (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.

Finally, the different approaches about assume-guarantee verification methods for component-based systems were proposed in [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, the component evolution means only adding some behaviors to the 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.

Chapter 9

Conclusion

9.1 Summary of the Dissertation

The research in this dissertation focuses on assume-guarantee verification of evolving component-based software (CBS) in the context of the component evolution at software design level and source code level. In the research, the component evolution means *adding only some new behaviors to the component without losing the old behaviors*. We think that adding behaviors to the component is enough for the software component evolution. With this approach, we have a simpler and faster assume-guarantee approach to recheck the evolved CBS. 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 the assume-guarantee verification processes. The research focuses only on checking the safety properties of CBS where behaviors of components can be represented by LTSs.

The first and the second chapters of the dissertation are about the context and the background of this research. The third chapter is about two current approaches for model checking a system and a proposed method for new assumption regeneration in the context of the component evolution. The main contributions of the research are in Chapters 4, 5, and 6. Chapter 7 is a larger experiment for three typical CBS systems. Chapter 8 is about related works.

In Chapter 4, we propose a method for generating minimal assumptions for the assume-guarantee verification of CBS. The method is an improvement of the described L*-based assumption generation method. The key idea of the proposed method is finding a minimal assumption in the search spaces of the candidate assumptions. These minimal assumptions are seen as the environments needed for the components to satisfy a property and for the rest of the system to be satisfied. In this method, we have improved the technique for answering membership queries of the Teacher which helps the L* 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 being 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 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 tools for the assumption generation method proposed in [10] and our minimized assumption generation method. This implementation is used to verify some typical CBS systems to show the effectiveness of the proposed method.

Chapter 5 proposes an effective framework for assume-guarantee verification of component-based software in the context of the component evolution at design level. The component evolution means that adding only some new behaviors to the component without losing the old behaviors. In this framework, if the model of a component is evolved after adapting some refinements, the whole CBS of many models of the existing components and the evolved model of the evolved component is not required to be rechecked. It only checks whether the evolve model satisfies the assumption of the system before evolving. If it does, 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 generated again by reusing the entire assumption as the previous verification result. We propose two methods for the new assumption regeneration: assumption regeneration and minimized assumption regeneration. Our work does not regenerate the new assumption from scratch. The methods reuse the current assumption as the previous verification result to regenerate the new assumption at much lower computational cost. Though the proposed framework considers the simple case where the CBS only consists of two components M_1 and M_2, we can generalize it for a larger CBS containing n-components $M_1, M_2, ..., M_n$ $(n \geq 2)$. Although evolution may occur on some components, theoretically, we can suppose that the CBS only allows us to evolve M_n. In order to apply the proposed framework for the larger CBS, we can consider the CBS as a software system which contains two components, i.e., the compositional component $M_1 \| M_2 \| ... \| M_{n-1}$ and M_n. The framework for the larger CBS consists of the similarly steps as described above because we only focus on the evolved component M'_n of M_n. 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 4.1 of Chapter 4. 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 evolved CBS systems to show the effectiveness of the proposed method. Although we understand that the CBS systems used in the experiment may not large enough to show effectiveness of the proposed framework, the systems are typical in practice. Even if the systems are not large enough, our approach can reduce the numbers of membership queries and candidate assumptions (see Table 5.1). Though the reduced numbers are small, our approach is applied many times during the software life-cycle because evolution often occurs at any time in any phases of the software development process. As a result, the obtained benefit from our approach becomes larger and larger. Moreover, LTSs are popular and practical models to describe behaviors of software in software engineering community. When verifying large-scale CBS systems, even if the behaviors of the software components are very complex, our work only focuses on the observable behaviors of each components. With this approach, we hope that our framework is effective for verifying practical software systems.

Chapter 6 proposes a framework for modular conformance testing and assume-guarantee verification of evolving CBS at source code level. This framework exactly is a combination of the proposed approaches presented in Chapters 4 & 5 and the modular conformance testing approach in order to deal with the described issues of AMC for rechecking of the evolved component-based software. The framework includes two stages: modular conformance testing for updating inaccurate models of the evolved components and assume-guarantee verification for evolving CBS. In this framework, when a component is evolved after adapting some refinements, the whole evolved system of many existing components and the evolved component are not required to be rechecked. The proposed framework focuses only on this component and its model in order to update the model and to recheck the whole evolved system. With this approach, we have a simpler assume-guarantee approach to recheck the evolved CBS at source code level. 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 via the VC algorithm. 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. The proposed framework in this Chapter not only focuses on the evolved component and its model but also reuses the entire current model of the component before evolving for learning the accurate model

and rechecking the evolved CBS. By this approach, the 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 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. Moreover, we separate the model learning and the assume-guarantee verification into two independent processes. The assume-guarantee verification method is applied once when an accurate model of the evolved component has been generated. Therefore, the approach is more effective than the current approaches proposed in [47, 12, 48]. The effectiveness of the assume-guarantee verification has been presented in Chapter 5.

In Chapter 7, we describe three CBS examples which have sizes larger than the sizes of the examples used in Chapter 4 & 5. We also present the experimental results obtained by applying the proposed approaches for the systems. These experiments are not only to show the practical usefulness of our proposed approaches but also to present how to generalize the proposed approaches for larger CBS.

9.2 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 all of the proposed frameworks and the proposed methods in the dissertation for larger component-based software, where CBS contains more than two components. We are also improving the frameworks and the methods, and applying some larger 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. Our work focuses only on checking the safety properties so we are going to extend the proposed approaches for checking other properties, e.g., liveness properties.

The models used in this dissertation are represented by LTSs as a kind of finite state machines. The LTS models are familiar to many programmes and engineers. They are used to specify the dynamic behaviors of objects in well-known object-oriented design methods such as object-oriented development [52], object modelling technique [54] and, more recently, the all-encompassing Unified Modelling Language [53]. They are also extensively used in the design of digital circuits - the original engineering use. Moreover, the LTS models have well-defined mathematical properties, which facilitate formal analysis and mechanical checking, thus avoiding the tedium and error introduction inherent in manual formal methods. However, the LTS models cannot describe behaviors of all software systems in practice. This is one of the limitations of our work. We only focus on checking the CBS systems where their components can be represented by LTSs. We are

going to apply other specification methods to model behaviors of software components, e.g., modal transition systems (MTSs).

In Chapter 4, we have proposed a method for generating minimal assumptions. However, the breadth-first-search which is used in our work, may be not practical because it consumed too much memory. For larger-scale systems, the computational cost for generating the minimal assumption is very high. 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 an exponentially growing of the number of the observation tables. This makes our method unpractical for 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 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 $L(A_i)$ is exactly a subset of the language of the assumption being 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. 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. Moreover, in the proposed method, we focus only on minimizing the size of the generated assumption. The generated minimal assumption does not correspond to the strongest assumption which satisfies the compositional rules. Instead of focusing on the size, it should be better to focus on the weakness of the generated assumption.

As mentioned in Chapter 5, 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 generate again 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 latter is based on the method proposed in Chapter 4. Although we reuse the current assumption as an approach for reducing the search space of the observation tables, the computational cost for regenerating the minimal assumptions still is high. We are investigating to apply

the improvements presented in Chapter 4 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. Moreover, in the case where the component evolution means that adding some new behaviors to the component and removing some old behaviors from the component, the proposed methods cannot reuse the entire assumption of the CBS before evolving directly because the component evolution may change the unknown language U of the assumption being learned. A potential solution for 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 CBS before evolving 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 $A'(p)$.

In Chapter 6, 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 VC algorithm. Furthermore, in the case where the component evolution means that adding some new behaviors to the component and removing some old behaviors from the component as mentioned above, the proposed modular conformance testing method cannot reuse the entire inaccurate model for learning the accurate one of the evolved component. The method should be improved in order to deal with this issue.

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 (June 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 (June 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 (June 2007).

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

[30] E. W. Stark: A Proof Technique for Rely/Guarantee Properties. In: the 5th Conf. on Found. of Soft. Tech. and Theoretical Computer Science, pp. 369–391 (1985).

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

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

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

[34] John H. .Reppy: "Concurrent Programming in ML", Cambridge University Press (1999).

[35] 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).

[36] 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).

[37] 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).

[38] 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).

[39] 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).

[40] 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).

[41] 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).

[42] 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).

[43] 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).

[44] 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).

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

[46] 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).

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

[48] E. Elkind, B. Genest, D. Peled, and H. Qu: "Grey-Box Checking", Proc. of 26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems, pp. 420–435 (Sept. 2006).

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

[50] W. J. Lee, H. -J. Kim, and H. S. Chae: "Safety Property Analysis Techniques for Co-operating Embedded Systems Using LTS", Proc. of 5th IFIP Workshop on Software Technologies for Embedded and Ubiquitous Systems, pp. 114–124 (May 2007).

[51] G. L. Peterson: "Myths About the Mutual Exclusion Problem", Information Processing Letters, vol. 12, no. 3, pp. 115–116 (1981).

[52] G. Booch: "Object-Oriented Development", IEEE Transactions on Software Engineering, vol. 12, no. 2, pp. 211-221 (1986).

[53] G. Booch, J. Rumbaugh, and I. Jacobson: "The Unified Modeling Language User Guide", Addison Wesley Longman Publishing Co., Inc. (1998).

[54] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen: "Object-Oriented Modeling and Design" Prentice-Hall, Inc. (1991).

Publications

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

[2] P. N. Hung, T. Aoki and T. Katayama: "An Effective Framework for Assume-Guarantee Verification of Evolving Component-Based Software", The Joint ERCIM Workshop on Software Evolution and the International Workshop on Principles of Software Evolution (IWPSE-EVOL), ACM, New York, pp. 109–118 (Aug. 2009).

[3] P. N. Hung, T. Aoki and T. Katayama: "A Minimized Assumption Generation Method for Component-Based Software Verification", In The 6th International Colloquium on Theoretical Aspects of Computing, LNCS 5684, pp. 277–291, Springer-Verlag Berlin Heidelberg (Aug. 2009).

[4] 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), pp. 479–486, IEEE Computer Society Press, Los Alamitos (Dec. 2008).

[5] P. N. Hung, N. T. Thang and T. Katayama: "An Assume-Guarantee Method for Modular Verification of Evolving Component-Based Software", In: 6th WADS in conjunction with the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 160–165 (June 2007).

Japan-Vietnam Workshop

[6] 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).

[7] 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).