JAIST Repository
https://dspace.jaist.ac.jp/

Title	モデル検査によるコンポーネントベースシステム検証へのアプローチ
Author(s)	Pham, Ngoc Hung
Citation	
Issue Date	2006-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/2039
Rights	
Description	Supervisor:Professor Takuya Katayama, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

An Approach Towards the Verification of Component-Based Systems via Model Checking

By Pham Ngoc Hung

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Takuya Katayama

September, 2006

An Approach Towards the Verification of Component-Based Systems via Model Checking

By Pham Ngoc Hung (410205)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Takuya Katayama

and approved by
Professor Takuya Katayama
Professor Kunihiko Hiraishi
Associate Professor Xavier Defago

August, 2006 (Submitted)

Copyright © 2006 by Pham Ngoc Hung

Abstract

Verification of software has received a lot of attentions of the software engineering community, specially modular verification of component-based software. However, to realize such an ideal component-based software paradigm, one of the key issues is to ensure that those separately specified and implemented components do not conflict to each other when composed - the *component consistency* issue. A potential solution to the above issue is modular verification of component-based software via model checking. The main goal in this thesis is to combine the best advantages of model checking and component-based development.

Currently there are many approaches have been proposed in modular verification of component-based software [2, 4, 7, 8, 10, 11, 22]. In [10, 11, 22], modular verification is rather closed. It 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 must be re-checked altogether. For this reason, the *"state space explosion problem"* will occur when it checks complex software. The approach in [2, 4, 7, 8] focuses on checking a system composed of two components; M_1 and M_2 which satisfies the property p *without composing* M_1 with M_2. For this goal, this technique finds an assumption A such that it is strong enough for M_1 to satisfy p and weak enough to be discharged by M_2. From these, the composition system $M_1 \| M_2$ satisfies p. However, this approach is viewed from a static perspective to re-generate new assumption. If the component changes after adapting some *refinements*, the assumption-generating approach is re-run on the whole component from beginning, i.e., the component model has to be re-constructed; and the assumption about the environment is then regenerated from that model. Therefore, this approach is not efficient to change the system. This thesis proposes a faster assume-guarantee verification approach for component-based software verification in the context of component refinement. In this approach, if a component is refined into a new component, the whole system of many existing components and the new component is not required to be re-checked altogether. It only checks the new component satisfying the assumption of the old system. If so, the new system also satisfies the property. Otherwise, the proposed technique performs some analysis to determine whether the property is indeed violated in the new system or whether the assumption of the old system is too strong for the new component to satisfy. If the assumption is too strong, a new assumption is re-generated. The technique in this thesis tries to reuse the results of the previous verification in order to have an incremental manner to re-generate the new assumption. It doesn't re-generate the new assumption from beginning. A case study is presented to illustrate the proposed approach. The LTSA [12] tool also is used to check correctness of the technique by some concrete examples.

Acknowledgement

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

I am specially grateful to Dr. Nguyen Truong Thang for his advices and support in my research and my living life. During my research, Dr. Thang has been as second advisor to me, and working with him was a great experience. I have learned a lot from his research and problem solving methodology.

I wish to continue my gratitude to Professor Kunihiko Hiraishi and Associate Professor Xavier Defago for gladly agreeing to serve as members of my thesis committee and for providing helpful advice to improve my thesis. Also, I would like to MITANI SANGYO CO., LTD. for the financial support which I have been provided with for staying and studying at JAIST.

On this occasion, I wish to thank all members of Katayama Laboratory in JAIST, specially Mr. Tanizaki Hiroaki, for their kind helps, discussions, and suggestions regarding my research.

I would like to take this chance to thank all Vietnamese people in JAIST, specially Prof. Ho Tu Bao, Research Associate Dam Hieu Chi, and Research Associate Huynh Van Nam, for their helps and supports not only in my research but also in my living life.

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

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

Contents

Abstract **i**

Acknowledgement **ii**

1 Introduction **1**

2 Modular Verification of Component-Based Software: A Survey **3**

2.1 Assume-Guarantee Verification of Component-Based Software 3

2.1.1 Weakest Assumption Generation Method 4

2.1.2 Assumption Generation Method using L* 9

2.2 Open Incremental Model Checking 10

2.3 Component-Interaction Automata 14

2.4 Some Open Problems 16

3 A Framework Towards Verification of Component-Based Software via Model Checking **18**

3.1 Background 18

3.1.1 Labeled Transition Systems 18

3.1.2 Deterministic Finite State Automata 22

3.1.3 Component Refinement 23

3.1.4 Assume-Guarantee Reasoning 24

3.1.5 LTSA Tool 24

3.2 Motivation 25

3.3 Related works 27

4 Assumption Generation using Learning Algorithms - L* **29**

4.1 The L* Learning Algorithms 29

4.2 Assumption Generation using The L* Learning Algorithms 32

4.3 New Assumption Re-generation 36

5 A Case Study **40**

5.1 System Specification 40

5.2 Assumption Generation 42

5.3 New Assumption Regeneration 46

5.4 Experiments 50

6 Conclusion and Future Works **56**

References **58**

List of Figures

2.1 The problem and the main idea of assume-guarantee verification approach. 4
2.2 LTSs for a *Mutex*, a *Writer* and mutual exclusion property. 5
2.3 The process for generating weakest assumption. 6
2.4 The composition system of *Mutex* and *Writer* with the mutual exclusion property. 7
2.5 The result after backward error propagation. 7
2.6 Generated weakest assumption. 8
2.7 Framework to generate assumption A. 9
2.8 An illustration of labelling at the interface states between M_1 and M_2. . . 10
2.9 The main idea of open incremental model checking. 11

3.1 An illustration of LTSs. 19
3.2 An illustration of parallel composition. 20
3.3 The parallel composition $Input \| Output$. 20
3.4 Order property. 21
3.5 Computing the composition $Input \| Output \| p_{err}$. 21
3.6 An illustration of DFA. 22
3.7 An illustration of getting a safety LTS from a DFA. 23
3.8 An illustration of component refinement. 24
3.9 A framework for component refinement. 25
3.10 A framework to generate assumption [4, 8]. 26
3.11 The new assumption re-generation process using L*. 28

4.1 The interaction between L* Learner and the Teacher. 30
4.2 The L* Algorithms. 31
4.3 An illustration of a closed observation table (S,E,T) and its candidate DFA. 32
4.4 A general view of assume-guarantee verification. 33
4.5 The LTS A_{cex} is created from the counterexample *cex*. 34
4.6 A framework to generate assumption using the L* learning algorithms [4, 8]. 35
4.7 The L* learning algorithms for new assumption regeneration. 38
4.8 The process for new assumption regeneration using L*. 39

5.1 Components and order property of the illustration system. 41
5.2 The component *Output* is *refined* into new component *Output'*. 42
5.3 The empty observation table at initial step. 43

5.4 Simulation the empty string λ on the composition system $Input \| P_{err}$. . . . 43
5.5 The observation table after updating by making membership queries. . . . 44
5.6 The observation table after adding *out* into *S*. 44
5.7 The observation table after re-updating by making membership queries. . . 44
5.8 Computing the composition system $A_1 \| Input \| P_{err}$. 45
5.9 The observation table after adding *ack* into *S*. 46
5.10 The observation table after re-updating by making membership queries. . . 46
5.11 The observation table after adding *send* into *S*. 47
5.12 Computing the composition system $A_2 \| Input \| p_{err}$. 47
5.13 Computing the composition system $Output' \| A_{2_{err}}$. 48
5.14 Simulating the $cex \uparrow \Sigma$ on the composition system $Input \| p_{err}$. 48
5.15 The safety LTS A_3 is created from the closed table (S,E,T). 49
5.16 Computing the composition system $A_3 \| Input \| p_{err}$. 49
5.17 The safety LTS A_4 is created from the closed table (S,E,T). 49
5.18 Computing the composition system $A_4 \| Input \| p_{err}$. 50
5.19 Computing the composition system $Output' \| A_{4_{err}}$. 50
5.20 FSPs and LTSs of illustration system in LTSA tool. 51
5.21 The checking result of composition system $A_1 \| Input \| p_{err}$. 52
5.22 The checking result of composition system $A_2 \| Input \| p_{err}$. 52
5.23 The checking result of composition system $Output \| A_{2_{err}}$. 53
5.24 The checking result of composition system $Output' \| A_{2_{err}}$. 53
5.25 The checking result of composition system $A_3 \| Input \| p_{err}$. 54
5.26 The checking result of composition system $A_4 \| Input \| p_{err}$. 55
5.27 The checking result of composition system $Output' \| A_{4_{err}}$. 55

Chapter 1

Introduction

Component-based development, also known that developing software systems through composition of well-defined independent components, is one of the most important technical initiatives in software engineering. Component-based development continues to hold the attention of the software engineering community as it is considered to be an open, effective and efficient approach to reduce development cost and time, while increasing software quality. A wide range of concepts and technologies have been proposed for component-based software development.

However, to realize such an ideal component-based software paradigm, one of the key issues is to ensure that those separately specified and implemented components do not conflict to each other when composed - the *component consistency* issue. Currently well-known technologies such as CORBA (OMG), COM/DCOM or .NET (Microsoft), Java and JavaBean (Sun) etc., only support component *plugging*. However, components often fail to co-operate, i.e, the *plug-and-play* mechanism fails. A potential solution to the above issue is modular verification of component-based software via model checking. Model checking is an important approach for improving software reliability. It provides exhaustive state space coverage for the systems being checked and is particularly effective in detecting difficult coordination errors which frequently result from component composition. However, a major problem of model checking is *"state space explosion"*. The main goal in this thesis is to combine the best advantages of the two; model checking and component-based development.

Currently there are many approaches proposed in modular verification of component-based software [2, 4, 7, 8, 10, 11, 22]. In [10, 11, 22], modular verification is rather closed. It 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 must be re-checked altogether. For this reason, the *"state space explosion problem"* will occur when it checks complex software. The approach in [2, 4, 7, 8] focuses on checking a system composed of two components; M_1 and M_2 which satisfies the property p *without composing* M_1 with M_2. For this goal, this technique finds an assumption A such that it is strong enough for M_1 to satisfy p and weak enough to be discharged by M_2. From these, the composition system $M_1 \| M_2$ satisfies p. However, this approach is viewed from a static perspective to re-generate new assumption. If the component changes after adapting some

refinements, the assumption-generating approach is re-run on the whole component from beginning, i.e., the component model has to be re-constructed; and the assumption about the environment is then regenerated from that model. Therefore, this approach is not efficient to change the system. This thesis proposes a faster assume-guarantee approach for component-based software verification in the context of component refinement. Suppose that there is a fixed based architecture F as a framework and an extension C_1. The extension C_1 is plugged with the framework F by some mechanism. Firstly, we know that the system which contains of F and C_1 satisfies a property p (i.e., $F \| C_1 \models p$). After that, the component C_1 is refined into a new component C_2 by adding some states and transitions into C_1. The new system which contains of the framework F and the new component C_2 must be re-checked that whether it satisfies the property p (i.e., $F \| C_2 \models p?$). For this purpose, the proposed technique only checks formula $\langle \text{true} \rangle\ C_2\ \langle A(p) \rangle$, where $A(p)$ is an assumption between two components; F and C_1, that is strong enough for F to satisfy p but weak enough to be discharged by C_1 (i.e., $\langle A(p) \rangle\ F\ \langle p \rangle$ and $\langle \text{true} \rangle\ C_1\ \langle A(p) \rangle$ both hold). The assumption $A(p)$ is generated by using a learning algorithms called L* [1, 24]. In this technique, components (i.e., F, C_1, and C_2), the property p, and assumptions are represented by LTSs. In order to check the formula $\langle \text{true} \rangle\ C_2\ \langle A(p) \rangle$, the technique computes the composition system $C_2 \| A(p)_{err}$, where the error LTS $A(p)_{err}$ is created from the LTS $A(p)$ by applying the definition 3.4. If the formula holds, the new composition system $F \| C_2$ satisfies the property p. Otherwise, this step returns a counterexample *cex* to witness this fact. The proposed approach then performs some analysis to determine whether p is indeed violated in the new system $F \| C_2$ or whether $A(p)$ is too strong for C_2 to satisfy. If the assumption $A(p)$ is too strong, a new assumption $A_{new}(p)$ between the framework F and the new component C_2 is re-generated. It re-generates the new assumption *without re-running* on these whole components from beginning. The proposed approach tries to reuse the results of the previous verification (between F and C_1) in order to have an incremental manner to re-generate new assumption. For this reason, it is very significant to verify the component-based systems in the context of component refinement.

This thesis is organized in six chapters as follows. I first provide an introduction in Chapter 1. Chapter 2 introduces a survey of some approaches for modular verification of component-based software. It also presents some limited problems and some open problems under research from these approaches. Chapters 3, 4, 5 present my contributions in this thesis. Chapter 3 introduces some background, my problem and proposed approach to verify component-based software in the context of component refinement in a general view. Chapter 4 presents the learning algorithms - L*, the assumption generation and the new assumption re-generation method. Correctness and termination of the new assumption re-generation method also presents in this chapter. Chapter 5 describes a case study to illustrate my approach presented in Chapters 3,4,5. Finally, Chapter 6 presents conclusion and future works.

Chapter 2

Modular Verification of Component-Based Software: A Survey

This chapter presents a survey of some approaches for component-based software verification, i.e, assume-guarantee verification, open incremental model checking and component-interaction automata. With each approach, this chapter introduces some base concepts, the main idea for verification and some limited problems. The ending of this chapter presents some open problems which are under research from these approaches.

2.1 Assume-Guarantee Verification of Component-Based Software

Assume-guarantee verification of component-based software approach was proposed by D. Giannakopoulou [2, 4, 7, 8]. This approach is based on the idea of assumption model checking. It presents a promising way of dealing with the verification of large systems. It is based on *"divide and conquer"* approach, i.e., the property of a system is decomposed into properties of its components, which are then checked separately.

This technique focuses on checking a system composed of two components; M_1 and M_2. The main goal of this approach is that how to check this system whether it satisfies the property p *without composing* M_1 with M_2 (i.e., $M_1 \| M_2 \models p?$). For this goal, this technique finds an assumption A such that A as environment of M_1, and M_1 satisfies p, and true as environment of M_2, and M_2 satisfies A, where A is strong enough for M_1 to satisfy p and A is weak enough to be discharged by M_2. From these, the rule $\langle \text{true} \rangle$ $M_1 \| M_2$ $\langle p \rangle$ also holds. Figure 2.1 illustrates the goal and the solution of assume-guarantee verification approach in a general view.

The assume-guarantee verification approach uses *Labeled Transition Systems* (LTSs) to model the behavior of two components M_2 and M_2, the assumption A, and the property p. A LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ has a communicating alphabet αM. Internal/local actions

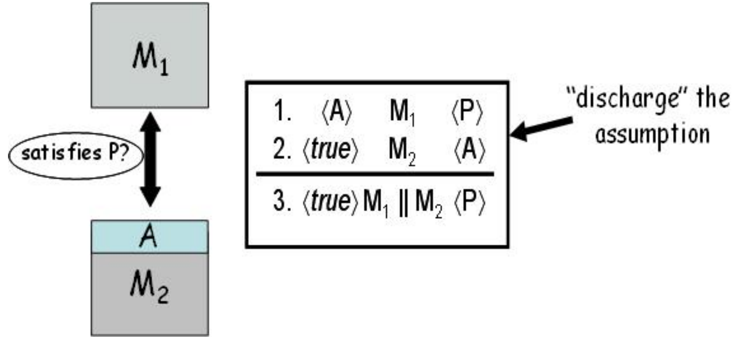


Figure 2.1: The problem and the main idea of assume-guarantee verification approach.

represented by action "τ". LTSs assembled with parallel composition operator "$\|$" by synchronizing shared actions and interleaving remaining actions. Error state π is included into the automaton of p before the cross-product automata of M_1, M_2 and p is taken. The error LTS p_{err} (with the error state π) is constructed from the original p by adding transitions from all states in p to π on which the actions are the complement of all existing outgoing transitions at each state.

From the above solution, the major problem in this approach is that how to find the assumption A. There are two methods in generating the assumption A automatically as follows:

1. The first method was presented in [7]. Given component M_1, property p, and the interface of M_1 with its environment, generate the weakest environment assumption A_W such that: assuming A_W, $M_1 \models p$. The weakest assumption A_W means that it restricts the environment no more and no less than it needs to, i.e for all environments E: $E \| M_1 \models p$ iff $E \models A_W$. This technique 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 $(\alpha M_1 \cup \alpha p) \cap \alpha M_2$ which do not lead to the state π in $M_1 \| p$. In this case M_2 is not known and considered as the environment of M_1.

2. The second method was presented in [2, 4, 8]. This technique uses a learning algorithms called L* to iteratively generating an assumption A such that A is weak enough to be guaranteed by M_2 but is strong enough to make M_1 satisfy p. The assumption A generated in this approach is stronger than the weakest assumption A_W. This learning algorithm often stops before A converges to A_W.

2.1.1 Weakest Assumption Generation Method

The traditional method to verifying a property of an open system (i.e., a software component that interacts with an environment, represented by other components) is to check it for all the possible environments. The result of verification is either true, if the property holds for all the possible environments, or false, if there exists some environment that can lead the component to falsify the property. The traditional approach is overly pessimistic

and only appropriate for the analysis of closed systems, where no further interaction with the environment is expected. When analyzing open systems, an optimistic view, which assumes a helpful environment, is more appropriate. Usually, software components are required to satisfy properties in specific environments, so it is natural to accept a component if there are some environments in which the component does not violate the property.

In this method, the result of component verification is also true, if the property holds for all environments. However, the result is false only if the property is falsified in all environments. If there exist some environments in which the component satisfies the property, the result of verification is *not false*, as in the traditional method, but rather true in environments that satisfy a specific assumption. This assumption, i.e., a property LTS, is automatically generated and characterizes exactly those environments. Intuitively, this environment assumption encodes all possible winning strategies of the environment in a game between the system, which attempts to get to the error state, and the environment, which attempts to prevent this.

For example, Figure 2.2 illustrates LTSs for two components *Writer* and *Mutex*. The state 0 is the initial state. The *Writer* acquires the *Mutex* (action W.acquire), enters and subsequently exists a critical section (W.enterCS, W.exitCS) used to model the fact that the *Writer* updates some shared variable, and then releases the *Mutex* (W.release). The *Mutex* component can be acquired and released by the *Writer* (W.acquire, W.release) or its environment (E.acquire, E.release), but a single component can hold it at any one time. A mutual exclusion property for a system consisting of the LTSs of *Mutex* and *Writer*. The property comprises states 0, 1, 2 and the transitions denoted by solid arrows. It expresses the fact that the component and its environment should never be in their critical sections at the same time. In other words, the intervals defined by their mutual enterCS and exitCS actions should never overlap. The dashed arrows illustrate the transitions to the error state that are added to the property to obtain its error LTS.

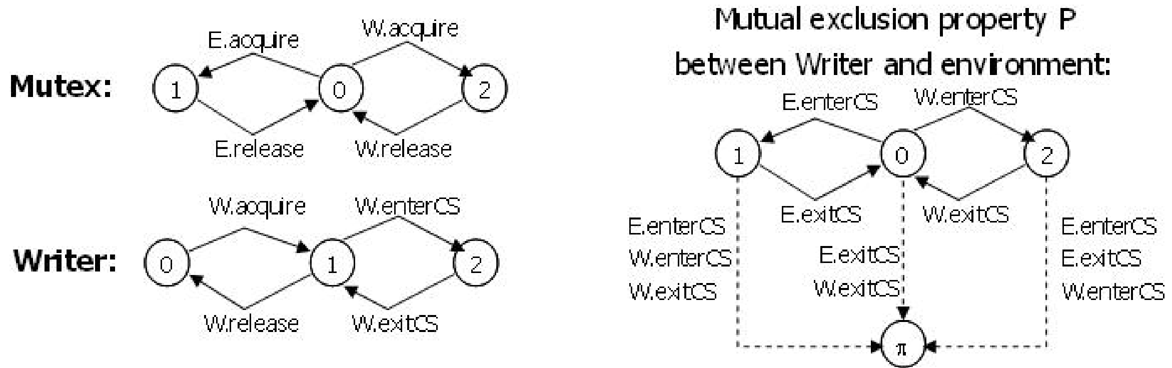


Figure 2.2: LTSs for a *Mutex*, a *Writer* and mutual exclusion property.

There are three steps to generate the weakest assumption A_W illustrated in Figure 2.3 as follows:

Step 1: Composition and Minimization

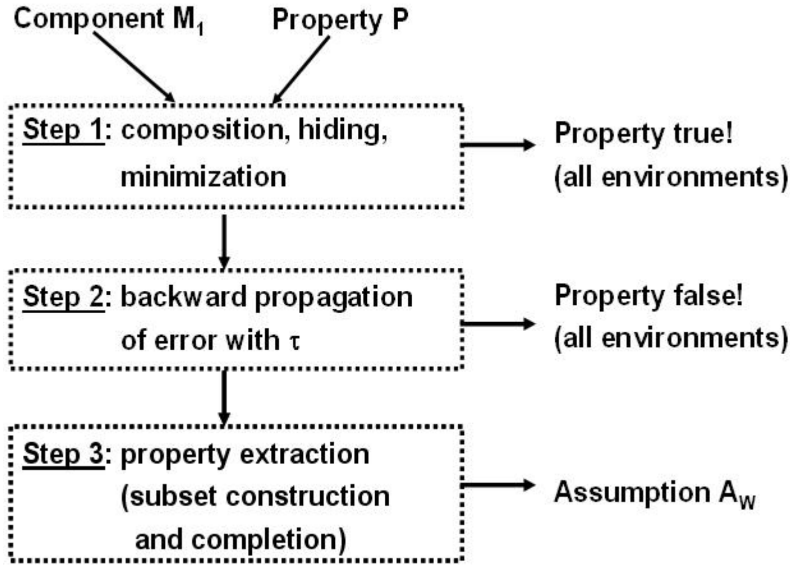


Figure 2.3: The process for generating weakest assumption.

Given an open system and a property LTS that may relate the behaviour of the system with the behaviour of the environment, the first step is to compute all the violating traces of the system for unrestricted environments, and turn into τ all actions in these traces over which the environment has no control, i.e., the internal actions of the system. We perform this step by building the composition of the system with the error LTS of the property, and subsequently hiding the internal actions of the system. The resulting LTS can be minimized with respect to observational equivalence, since such minimization preserves traces.

For example, Figure 2.4 depicts the result of composing the components with the mutual exclusion property depicted in Figure 2.2, after minimization. The internal actions of the system, i.e., the W labelled transitions, were abstracted to τ.

If the error state is not reachable in this composition, the property is true in any environment, and this is reported to the user. Otherwise, we identify whether there exist environments that can help the system avoid the error in all circumstances; this is achieved through the following steps.

Step 2: Backward Error Propagation

This step first performs backward propagation of the error state over τ transitions, thus pruning the states where the environment cannot prevent the error state from being entered via one or more τ steps. Since we are interested only in the error traces, we also eliminate the states that are not backward reachable from the error state. If, as a result of this transformation, the initial state becomes an error state, it means that no environment can prevent the system from possibly reaching the error state, so the property is false (for all environments) and this is reported to the user.

Consider again the composite system in Figure 2.4 . The thicker line marks the only transition that remains in the system after minimization. As a result of backward prop-

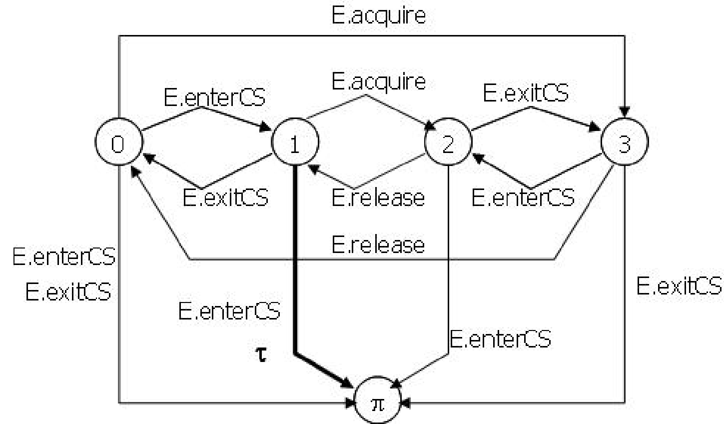


Figure 2.4: The composition system of *Mutex* and *Writer* with the mutual exclusion property.

agation, we identify state 1 with the error state; the result is shown in Figure 2.5. The intuition here is that, if the component is in a state from which it can violate the property by some number of internal moves, then no environment can prevent the violation from occurring.

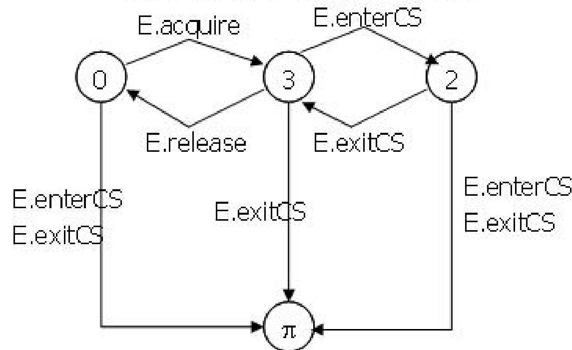


Figure 2.5: The result after backward error propagation.

Step 3: Property Extraction

This step builds the property LTS that is our assumption. It performs this in two stages; first it builds the error LTS for the assumption, from which it extracts the corresponding property LTS. Note that the LTS resulting from Step 2 might not be an error LTS, although it contains an error state. Recall from the background section that the error LTS is deterministic and complete.

In order to get an error LTS we make the LTS obtained from step 2 deterministic by applying to it τ elimination and the subset construction, but by taking special care of the π state as follows. During subset construction, the states of the deterministic LTS that is being generated are sets of states in the original non-deterministic LTS. In our context, if any one of the states in the set is π, the entire set becomes π. Intuitively, a trace that non-deterministically may or may not lead to an error has to be considered as an error trace. Such non-determinism reflects the fact that, by performing a particular sequence of actions, the environment cannot guarantee that the component will avoid error states.

For example, consider again the composite system in Figure 2.4. There are two outgoing transitions from the initial state 0 that are labelled by the same environment action E.enterCS: one leads to the error state, while the other one leads to state 1. This means that if the environment performs action E.enterCS, it can not prevent the system from getting to the error, so we would like to identify state 1 with π. In Figure 2.4, this was achieved during Step 2, but this may not be the case in general.

What remains to be performed at this stage is to make the resulting LTS complete. Completion is performed by adding a new "sink" state to the LTS, and adding a transition to this state for each missing transition in the "incomplete" LTS. The missing transitions in the incomplete LTS represent behaviour of the environment that is never exercised by the open system under analysis. As a result, no assumptions need to be made about these behaviours. The sink state reflects exactly this fact, since it poses no implementation restrictions to the environment.

Once we have the error LTS, we obtain the assumption by deleting the error state and the transitions that lead to it. Figure 2.6 depicts the assumption generated for our example. Since the result from Step 2 is already deterministic, we get the assumption by completing it with the sink state, denoted by θ, and deleting the π state. The assumption expresses the fact that the environment should only access its critical section protected by the *Mutex*. Moreover, as imposed by the *Mutex*, E.acquire and E.release actions of the environment can only alternate, and therefore any different behaviour is inconsequential.

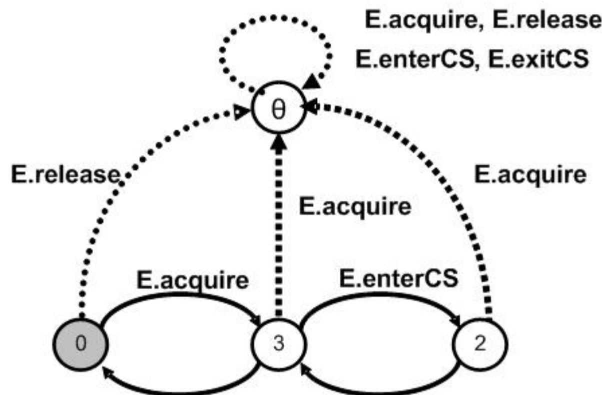


Figure 2.6: Generated weakest assumption.

2.1.2 Assumption Generation Method using L*

This method uses a learning algorithms called L* to iteratively generating an assumption A such that A is weak enough to be guaranteed by M_2 but is strong enough to make M_1 satisfy p illustrated in Figure 2.1.

To obtain appropriate assumptions, this method applies the compositional rule in an iterative fashion as illustrated in Figure 2.7. At each iteration i, a 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 rule are then applied. Step 1 checks the formula $\langle A_i \rangle\ M_1\ \langle p \rangle$. If this formula doesn't hold, it means that this candidate assumption is *too weak*. So the assumption A_i needs to be strengthened. Otherwise (step 1 returns true), it means that A_i is strong enough for the property to be satisfied. The assumption A_i is forwarded to step 2. The step 2 checks the formula $\langle \text{true} \rangle\ M_2\ \langle A_i \rangle$. If this formula holds (step 2 returns true), then the property p holds in the composition system $M_1 \| M_2$. L* terminates and returns generated assumption A=A_i. Otherwise, further analysis is required to identify whether p is indeed violated in $M_1 \| M_2$ or whether A_i is too strong for M_2 to satisfy. If the assumption A_i is too strong, A_i must be weakened in iteration i+1. The new assumption A_{i+1} may of course be too weak, and therefore the entire process must be repeated.

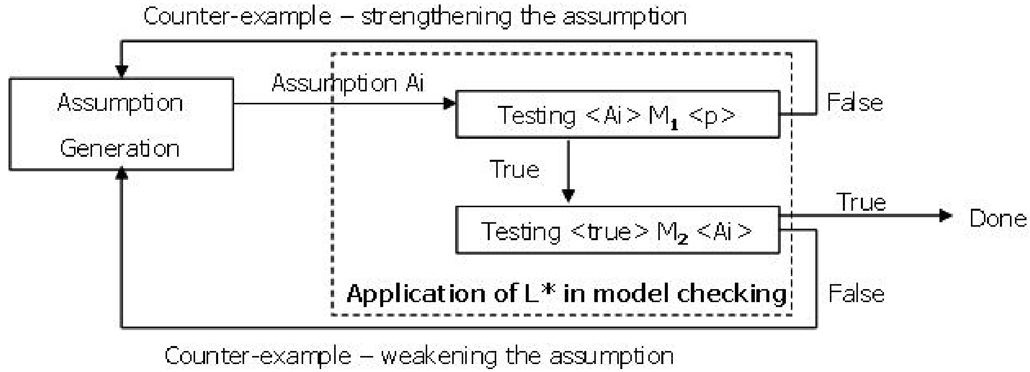


Figure 2.7: Framework to generate assumption A.

Although, the assume-guarantee verification approach presented an effective and efficient approach for verification of component-based systems. There are some limited problems on this approach as follows:

- The system in this approach only has two components M_1 and M_2. It therefore is very simple.
- In this approach, two components M_1 and M_2 can not change. Therefore, this approach will not be efficient to change the system in the context of component refinement.
- When the system has more than two component, i.e., M_1, M_2, ..., M_n. This approach will has to generate (n-1) assumptions among these components. It will be very

difficult because these assumptions are not independent together and in this case, the learning algorithms will be low scalability.

- This approach didn't mention the interfaces of these components. It also didn't mention the interactions among these components. The interesting problem is that we need to check $M_1 \| M_2 \| ... \| M_n \models p$? The interfaces of each component M_i has to what information related p?

2.2 Open Incremental Model Checking

Open incremental model checking (OIMC) approach was proposed by Fisler, Thang&Katayama [6, 17, 20]. This approach is also based on the idea of assumption model checking. The assumption model checking approach [11] focuses on the system which contains of two sequential modules M_1 and M_2. It is based on the idea that possible to model check within M_1 only if knowing the labels at the interface states between M_1 and M_2 by representing the whole M_2 with those labels illustrated in Figure 2.8.

The OIMC approach focuses on the interaction between two components *Base* and *Extension*. A property p is guaranteed to hold in the component *Base*. This approach derives a set of preservation constraints at the interface states of the component *Base*. When a new component (*extension*) is added, the whole system which contains of two components; *Base* and *Extension* is not required to re-check altogether. It only checks the new component *Extension* preserve the constraints. If it is preserved, the property holds in the new system. Although OIMC focuses on component refinement, but also applicable to component addition, e.g. Commercial-Off-The-Shelf (COTS). This verification approach is incrementally modular because it model checks each component separately. This approach is also incremental openness for future changes even unanticipated future changes.

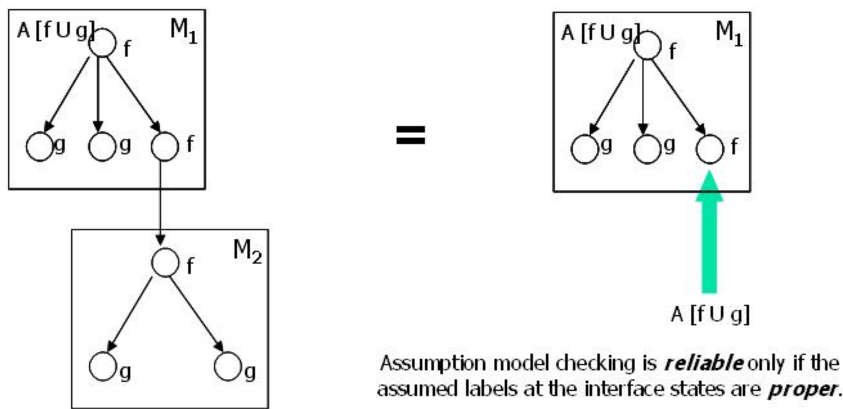


Figure 2.8: An illustration of labelling at the interface states between M_1 and M_2.

Figure 2.9 describes the main idea of OIMC approach. After assumption model checking in the component E, if the constraints at this exit state are preserved, there is no need to check further in component B.

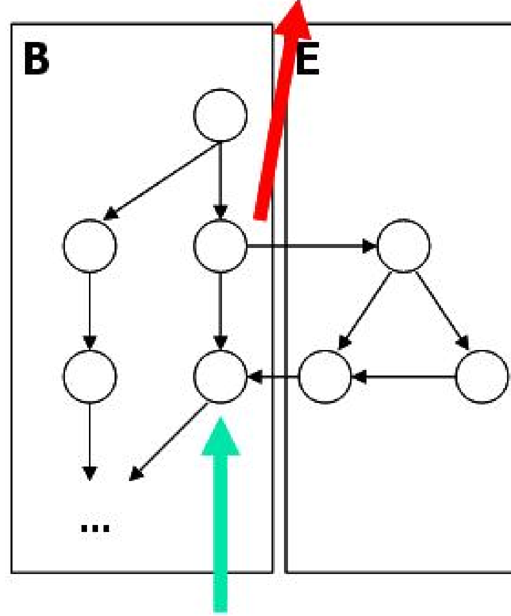


Figure 2.9: The main idea of open incremental model checking.

This approach consists of the following activities:

1. Deriving a set of *preservation constraints* at the interface states of the *Base* such that if those constraints are preserved, the property inherent to the base under consideration is guaranteed.

2. The *Extension* component does not violate the property p of the *Base* if, during its execution, the above constraints are preserved.

The current component specification often focuses on the syntactic level. It is limited in dealing with semantic constraints. Even so, only static aspects of components are specified. This approach gives a formal model to make component specification more comprehensive by including component semantic There are two semantic aspects: *dynamic behavior* and *inter-component consistency*. The component semantic is represented via the powerful temporal logic CTL.

Definition 2.1. A state transition model M is represented by a tuple $\langle S, \Sigma, s_0, R, L \rangle$, where S is a set of states, Σ is the set of input events, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \rightarrow S$ is the transition function (where $PL(\Sigma)$ denotes the set of guarded

events in Σ whose conditions are propositional logic expressions), and $L : S \to 2^{AP}$ labels each state with the set of atomic propositions true in that state.

A base is expressed by a transition model $\mathbf{B}= \langle S_{\mathbf{B}}, \Sigma_{\mathbf{B}}, s_{0\mathbf{B}}, R_{\mathbf{B}}, L_{\mathbf{B}} \rangle$ and an interface I. The interface is a tuple of two state sets $I=\langle exit,\ reentry \rangle$, where emphexit, *reentry* $\subseteq S_{\mathbf{B}}$. An extension is similarly represented by a model $\mathbf{E}= \langle S_{\mathbf{E}}, \Sigma_{\mathbf{E}}, \bot, R_{\mathbf{E}}, L_{\mathbf{E}} \rangle$, where $\bot$ denotes no-care value. Its interface is $J=\langle in,\ out \rangle$.

E can be semantically plugged with **B** via compatible interface states. Logically, along the computation flow, when the system is in an exit state $ex \in I.exit$ of **B** matched with an in-state $i \in J.in$ of **E**, denoted as $ex \leftrightarrow i$, it can enter **E** if the conditions to accept extension events, namely the set of atomic propositions at i, are satisfied. That is, $\bigwedge L_{\mathbf{B}}(ex) \Rightarrow \bigwedge L_{\mathbf{E}}(i)$, where $\bigwedge$ is the inter-junction of atomic propositions. Similar arguments are made for the matching of a reentry state $re \in I.reentry$ of **B** and an outstate $o \in J.out$ of **E**. The conditions resemble to pre- and post-conditions in design by contract.

Definition 2.2. Within interfaces I and J of **B** and **E**, the pairs $\langle ex,\ i \rangle$ and $\langle re,\ o \rangle$ can be respectively mapped according to the following conditions:

- $ex \leftrightarrow i$ if $\bigwedge L_{\mathbf{B}}(ex) \Rightarrow \bigwedge L_{\mathbf{E}}(i)$
- $re \leftrightarrow o$ if $\bigwedge L_{\mathbf{E}}(o) \Rightarrow \bigwedge L_{\mathbf{B}}(re)$

Unlike the current component technology using UML (Unified Modeling Language) and OCL (Object Constraint Language) to express *semantic constraints*, constraints in this paper are related to the Computation Tree Logic (CTL). CTL* logic is formally expressed via two quantifiers **A** (“for all paths”) and **E** (“for some path”) together with five temporal operators **X** (“next”), **F** (“eventually”), **G** (“always”), **U** (“until”) and **R** (“release”). CTL is a restricted subset of CTL* in which each temporal operator must be preceded by a quantifier. Specifically, the constraints at an interface (i.e., exit and reentry) state s is $\mathcal{V}_B(s, cl(p))$ defined as follows:

Definition 2.3. $cl(p)$ is the closure set of the property p holding in the component *Base* is the set of all sub-formulae of p including itself as follows:

- $p \in AP$: $cl(p) = \{p\}$, where AP is a set of atomic propositions.
- p is one of $\mathbf{AX}\ f$, $\mathbf{EX}\ f$, $\mathbf{AF}\ f$, $\mathbf{EF}\ f$, $\mathbf{AG}\ f$, $\mathbf{EG}\ f$: $cl(p) = \{p\} \cup cl(f)$
- p is one of $\mathbf{A}\ [f\ \mathbf{U}\ g]$, $\mathbf{E}\ [f\ \mathbf{U}\ g]$, $\mathbf{A}\ [f\ \mathbf{R}\ g]$, $\mathbf{E}\ [f\ \mathbf{R}\ g]$: $cl(p) = \{p\} \cup\ cl(f) \cup\ cl(g)$
- $p = \neg f$: $cl(p) = cl(f)$
- $p = f \vee g$ or $p = f \wedge g$: $cl(p)=cl(f) \cup\ cl(g)$

Definition 2.4. The truth values of a state s with respect to a set of CTL properties ps within a model $M = \langle S, \Sigma, s_0, R, L \rangle$, denoted as $\mathcal{V}_M(s, ps)$, is a function:$S \times 2^{CTL} \mapsto 2^{CTL}$, where:

- $\mathcal{V}_M(s, \emptyset) = \emptyset$
- $\mathcal{V}_M(s, \{p\} \cup ps) = \mathcal{V}_M(s, \{p\}) \cup \mathcal{V}_M(s, ps)$
- $\mathcal{V}_M(s, \{p\})= \{p\}$ if $M, s \models p$. Otherwise, $\mathcal{V}_M(s, \{p\}) = \{\neg p\}$

It known that the property p holds in the component **B**. When a new component **E** is added, it is required to re-check the new system **C** which contains of two components **B** and **E** satisfy the property p (i.e., $\mathbf{C}, s_{0C} \models p$?), where s_{0C} is the initial state of **C**. In the traditional approaches, they re-checked the whole system **C**. Therefore, it's very difficult to verify when the new system is very large and complex. In the OIMC approach, it only checks the new component **E** preserve the constraints. The important problem is that what constraints is and how to derive them. The answers of these questions are based on following definition:

Definition 2.5. **B** and **E** are in conformance at an *exit state ex* with respect to $cl(p)$ if $\mathcal{V}_{\mathbf{B}}(ex, cl(p)) = \mathcal{V}_{\mathbf{E}}(ex, cl(p))$.

The main idea of OIMC is based on the theorem as follows:

Theorem 2.1. Given a base **B** and a property p holding on **B**, an extension **E** is attached to **B** at some interface states. **E** does not violate property p if **B** and **E** conform with each other at all exit states. Namely, with all *exit state ex*, $\mathcal{V}_{\mathbf{B}}(ex, cl(p)) = \mathcal{V}_{\mathbf{E}}(ex, cl(p))$.

Now, it is required to check that **E** does not violate p. From above theorem, the OIMC approach only needs to verify the conformance at all exit states between **B** and **E**.

At the end of a task verifying p in **B**, at each state s, the truth values $\mathcal{V}_{\mathbf{B}}(s, cl(p))$ are also recorded. Corresponding to each exit state *ex*, within **E**, the algorithms to verify preservation constraints $\mathcal{V}_{\mathbf{B}}(ex, cl(p))$ as follows:

1. Seeding $\mathcal{V}_{\mathbf{B}}(re, cl(p))$ at any reentry state *re*
2. Executing a CTL assumption model checking procedure in **E** from *re* backward to *ex*. The formulae to check is (state=ex) $\rightarrow \mathcal{V}_{\mathbf{E}}(ex, \emptyset)$ $\forall \emptyset \in cl(p)$
3. Checking if $\mathcal{V}_{\mathbf{B}}(ex, cl(p)) = \mathcal{V}_{\mathbf{E}}(ex, cl(p))$.

The open incremental model checking approach focuses on the refinement aspect of components in which components are relatively coupled. However, the results of this approach can be equally applied to COTS. This approach advocates the inclusion of *dynamic behavior* and *component consistency* written in CTL to the component interface to better deal with component matching. However, there are some limited problems in this approach. The constraints are too strict so it is very difficult to apply this approach in practice. Moreover, the refinement case does not often arise.

2.3 Component-Interaction Automata

Component-interaction automata approach was proposed by L. Brim [3]. This approach presents a new approach to component interaction specification and verification process which combines the advantages of both architecture description languages (ADLs) at the beginning of the process, and a general formal verification-oriented model connected to verification tools at the end. After examining current general formal models with respect to their suitability for description of component-based systems, we propose a new verification-oriented model, Component-Interaction automata, and discuss its features. The model is designed to preserve all the interaction properties to provide a rich base for further verification, and allows the system behaviour to be configurable according to the architecture description (bindings among components) and other specifics (type of communication used in the synchronization of components).

The main ideas of this approach is base on combining the Architecture Description languages (ADLs) and a general formal model verification. The ADLs are very suitable for specification of hierarchical component architecture with defined interaction among components and behavior constraints put on component communication and interaction. But their specification power is limited by underlying model which is often not general enough to preserve all the interaction properties. The general formal model verification base on the automata theory. It is highly formal and general and usually supported by automated verification tools. It is designed for modeling of component interaction only. It is unable to describe the interconnection structure of hierarchical component architecture. The goal in this approach is that combining two approaches to gain the best benefits of both. It develops a general automata-base formalism which allows for the specification of component interaction according to the interconnection structure description in the ADL.

The idea is to support the specification and verification process automatically or semi-automatically so that it is accessible also for users with no special theoretical knowledge of the underlying model. The specification and verification process will constitute of the following phases:

1. The user selects an appropriate ADL and specifies the system architecture and component behaviour using an ADL tool.

2. Component behaviour description is transformed into the general formal model automatically using the model framework.

3. The hierarchical component composition is build within the framework with respect to the architecture description and synchronization type.

4. The result is verified directly within the model framework or transformed to a format accepted by verification tools.

Definition 2.6: Let $\mathcal{I} \subseteq \mathbb{N}$ be a finite set with cardinality k, and let for each $i \in \mathcal{I}, S_i$ be a set. Then $\prod_{i\in\mathcal{I}} S_i$ denotes the set $\{(x_{i_1}, x_{i_2}, ..., x_{i_k}) \mid (\forall j \in \{1, 2, ..., k\} : x_{i_j} \in S_{i_j}) \wedge \{i_1, i_2, ..., i_k\} = \mathcal{I} \wedge (\forall j_1, j_2 \in \{1, 2, ..., k\} : j_1 < j_2 \Rightarrow i_{j_1} < i_{j_2})\}$. If $\mathcal{I} = \phi$ then

$\prod_{i \in \mathcal{I}} S_i = \phi$. For $j \in \mathcal{I}$, $proj_j$ denotes the function $proj_j$: $\prod_{i \in \mathcal{I}} S_i \mapsto S_j$ for which $proj_j((q_i)_{i \in \mathcal{I}}) = q_j$.

Definition 2.7: A component-interaction automata C is a 5-tuple $\langle Q, Act, \delta, I, S \rangle$ where:

- Q is a set of states,
- Act is a finite set of actions, $\Sigma = ((X \cup \{-\} \times Act \times X \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$, where $X = \{n \mid n \in \mathbb{N}, n \text{ occurs in } S\}$, is a set of *symbols* called an *alphabet*,
- δ is a finite set of labelled transitions,
- $I \subseteq Q$ is nonempty set of initial states, and
- S is a tuple corresponding to a hierarchy of component name (from $\mathbb{N}$) whose composition C represents.

Symbols (-,a,B), (A,a,-), $(A, a, B) \in \Sigma$ are called *input*, *output*, and *internal symbols* of the alphabet Σ, respectively. Accordingly, transitions are called input, output and internal.

- The input symbol (-,a, B) represents that the component B receives an action a as an input.
- The output symbol (A, a,-) represents that the component A sends an action a as an output.
- The internal symbol (A, a, B) represents that the component A sends an action a as an output, and synchronously the component B receives the action a as an input.

Definition 2.8: Let $S = \{(Q_i, Act_i, \delta_i, I_i, S_i)\}_{i \in \mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a system of component-interaction automata such that sets of components represented by the automata are pairwise disjoint. Then $C = (\prod_{i \in \mathcal{I}} Q_i, \bigcup_{i \in \mathcal{I}} Act_i, \delta, \prod_{i \in \mathcal{I}} I_i, (S_i)_{i \in \mathcal{I}}$ is a component-interaction automata over S iff $\delta = \triangle_{OldInternal} \cup \delta_{NewInternal} \cup \delta_{Input} \cup \delta_{Output}$ where:

- $\triangle_{OldInternal} = \{(q, (A, a, B)), q' \mid \exists i \in \mathcal{I} : (proj_i(q), (A, a, B), proj_i(q')) \in \delta_i, \forall j \in \mathcal{I}, j{\neq}i : proj_j(q) = proj_j(q')\}$
- $\triangle_{NewInternal} = \{(q, (A, a, B)), q' \mid \exists i_1, i_2 \in \mathcal{I}, i_1{\neq}i_2 : (proj_{i_1}(q), (A, a, -), proj_{i_1}(q')) \in \delta_{i_1} \wedge (proj_{i_2}(q), (-, a, B), proj_{i_2}(q')) \in \delta_{i_2} \wedge \forall j \in \mathcal{I}, i_1{\neq}j \neq i_2 \; proj_j(q) = proj_j(q')\}$
- $\delta_{NewInternal} \subseteq \triangle_{NewInternal}$
- $\triangle_{Input} = \{(q, (-, a, B)), q' \mid \exists i_1 \in \mathcal{I} : (proj_{i_1}(q), (-, a, B), proj_{i_1}(q')) \in \delta_{i_1} \wedge \forall j \in \mathcal{I} : j{\neq}i_1 : (proj_j(q) = proj_j(q'))\}$
- $\delta_{Input} \subseteq \triangle_{Input}$

- $\triangle_{Output} = \{(q, (A, a, -)), q' \mid \exists i_2 \in \mathcal{I} : (proj_{i_2}(q), (A, a, -), proj_{i_2}(q')) \in \delta_{i_2} \land \forall j \in \mathcal{I} : j{\neq}i_2 : (proj_j(q) = proj_j(q'))\}$
- $\delta_{Output} \subseteq \triangle_{Output}$

Definition 2.9: An *execution fragment* of a component-interaction automata $C = \langle Q, Act, \delta, I, S\rangle$ is an infinite alternating sequence $q_0, x_0, q_1, x_1, ...$ of states and symbols of the alphabet Σ such that $(q_i, x_i, q_{i+1}) \in \delta$ for all $0 \leq i$. An *execution* of C is an *execution fragment* $q_0, x_0, q_1, x_1, ...$ such that $q_0 \in \mathcal{I}$. An execution fragment is closed if all its symbols are internal. A *trace* of C is a sequence $x_0, x_1, ...$ of symbols for which there is an execution $q_0, x_0, q_1, x_1, ...$

Verification: In real component-based systems one needs to verify various properties of system behaviour. If the system is modelled as a component-interaction automaton the behaviour capturing the interaction among components and architectural levels are the traces. Both linear and branching time temporal logics have proved to be useful for specifying properties of traces. There are several formal methods for checking that a model of the design satisfies a given specification. Among them those based on automata [6] are especially convenient for our model of Component-Interaction automata.

Although this approach evolves a new verification-oriented automata-based formal model. However, in this approach, the verification method is more general. It should be developed in more details.

2.4 Some Open Problems

From above approaches, there are some problems under research which may to continue researching as follows:

- We can use the ideas on the component-interaction automata approach and the OIMC approach to improve the limited problems of the assume-guarantee verification approach. If the system has more than two component, i.e., M_1, M_2, ..., M_n, we can use the idea of the component-interaction automata approach to combine two components as a bigger component and return to verify this system with two bigger components by applying the idea of assume-guarantee verification approach.
- We can improve the verification technique in the component-interaction automata approach by presenting its in more details.
- The interesting problem is that how to check the effects of two extension components C_1 and M_2 on the verification of F + $(C_1 \cap M_2)$. If it will be solved, when a new component is added, we checks the new system by only checking the new component cover the intersection $(C_1 \cap M_2)$. This is the main goal in this thesis.
- Implementation the OIMC approach to develop a tool for verification in practice.

- It will be significant if we propose a framework for unanticipated software changes for component-based systems. This problem was proposed by Thang&Katayama in [14] but it was more general.

Chapter 3

A Framework Towards Verification of Component-Based Software via Model Checking

This chapter proposes a faster assume-guarantee approach for component-based software verification in the context of component refinement in a general view. It contains of the problem, a framework to solve the problem, specially new assumption re-generation technique. Some basic concepts also are introduced (Section 3.1) in this chapter.

3.1 Background

3.1.1 Labeled Transition Systems

The proposed approach in this thesis uses *Labeled Transition Systems* (LTSs) to model the behavior of communicating components. A LTS is a directed graph with labeled edges. In addition to states and transitions, a set of labels called alphabet is associated with the system. All labels on transitions must be from that alphabet. Let *Act* 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 associated system. We require that the error state has no outgoing transition. A LTS is defined as follows:

Definition 3.1 (***Labeled Transition Systems***) [9]. A LTS M is a quadruple $\langle Q, \alpha M, \delta, q_0 \rangle$ where:

- Q is a non-empty set of states,
- $\alpha M \subseteq Act$ 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.

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

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

Definition 3.2 (***Traces***)[2, 4]. A trace t of an LTS M is a sequence of observable actions that M can perform starting at its initial state. For $\Sigma\subseteq Act$, we use $t\uparrow\Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a\notin\Sigma$. The set of all traces of M is called the language of M, denoted $L(M)$.

Let $t=\langle a_1, a_2, ..., a_n\rangle$ be a finite trace of a LTS M. We use [t] to denote the LTS $M_t=\langle Q, \alpha M, \delta, q_0\rangle$ with $Q=\langle q_1, q_2, ..., q_n\rangle$, and $\delta = \{(q_{i-1}, a_i, q_i)\}$, where $1 \leq i \leq n$.

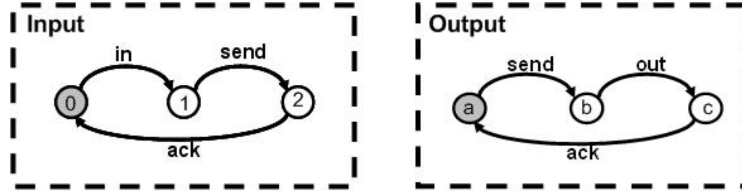


Figure 3.1: An illustration of LTSs.

Figure 3.1 shows graphical representations of two LTSs; *Input* and *Output*. The initial state of *Input* LTS in this example is *state 0*. The initial state of *Output* LTS is *state a*. 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 *Output* LTS, it 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. In this illustration, $\langle$*in*$\rangle$, $\langle$*in,send*$\rangle$, $\langle$*in,send,ack*$\rangle$ etc. are traces of Input LTS.

Definition 3.3 (***Parallel Composition***) [9]. The parallel composition operator $\|$ is a commutative and associative operator that combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions. Consider two LTSs; $M_1 = \langle Q_1, \alpha M_1, \delta_1, q_0^1\rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q_0^2\rangle$. The *parallel composition* between M_1 and M_2, denoted $M_1\|M_2$, is defined as follows. If $M_1 = \prod$ or $M_2 = \prod$, then $M_1\|M_2 = \prod$. Otherwise, $M_1\|M_2$ is a labeled transition system $M=\langle Q, \alpha M, \delta, q_0\rangle$ where $Q = Q_1\times Q_2$, $\alpha M = \alpha M_1 \cup \alpha M_2$, $q_0 = q_0^1 \times q_0^2$, and the transition relation δ is given by the rules:

$$(i)\frac{\alpha \in \alpha M_1 \cap \alpha M_2, (p, \alpha, p') \in \delta_1, (q, \alpha, q') \in \delta_2}{((p, q), \alpha, (p', q')) \in \delta} \tag{3.1}$$

$$(ii)\frac{\alpha \in \alpha M_1 \backslash \alpha M_2, (p, \alpha, p') \in \delta_1}{((p, q), \alpha, (p', q)) \in \delta} \tag{3.2}$$

$$(iii)\frac{\alpha \in \alpha M_2 \backslash \alpha M_1, (q, \alpha, q') \in \delta_2}{((p, q), \alpha, (p, q')) \in \delta} \tag{3.3}$$

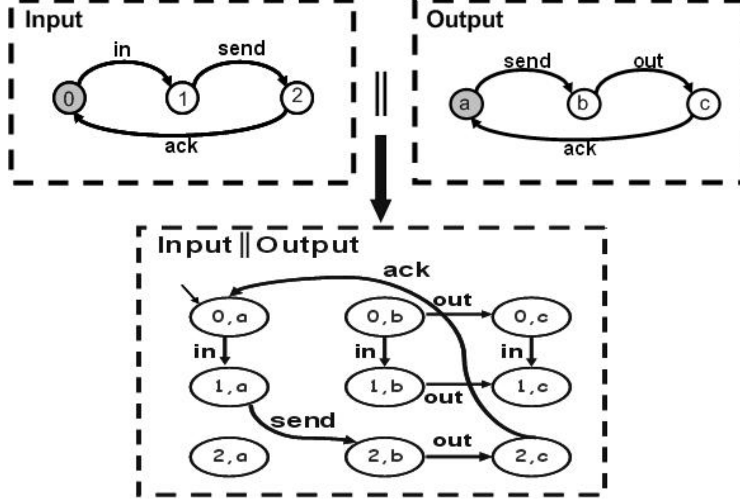


Figure 3.2: An illustration of parallel composition.

For example, Figure 3.2 describes the parallel composition *Input*$\|$*Output*. By removing all states which unreachable from the initial state *(0,a)* and their ingoing transitions, we obtain the parallel composition LTS *Input*$\|$*Output* as Figure 3.3. In this illustration, actions *send* and *ack* will each be synchronized and the others is interleaved.

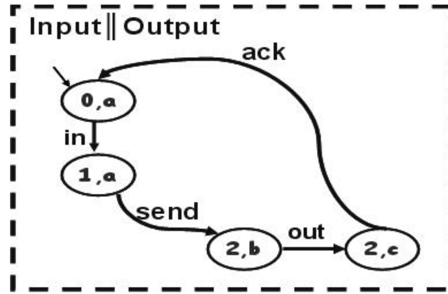


Figure 3.3: The parallel composition *Input*$\|$*Output*.

Definition 3.4 (***Safety LTSs, Safety Property, and error LTS***). We call a deterministic LTS that contains no π states a *safety LTS. A safety property* is specified as a safety LTS p, whose language $L(p)$ defines the set of acceptable behaviors over αp. An LTS M satisfies p, denoted as $M \models p$, if and only if $\forall \sigma \in L(M)$: $(\sigma \uparrow \alpha p) \in L(p)$. When checking the LTS M which satisfies the property p, an *error LTS*, denoted p_{err}, is created which traps possible violations with the π state. Formally, the *error LTS* of a property $p = \langle Q, \alpha p, \delta, q_0 \rangle$ is $p_{err} = \langle Q \cup \{\pi\}, \alpha p_{err}, \delta', q_0 \rangle$, where $\alpha p_{err} = \alpha p$ and $\delta' = \delta \cup \{(q, a, \pi) \mid a \in \alpha p \text{ and } \nexists q' \in Q : (q, a, q') \in \delta\}$.

The error LTS is complete, meaning each state other than the error state has outgoing transitions for every action in the alphabet. For example, Figure 3.4 describes the LTS of

property p and the 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 communication channel shown in Figure 3.1. 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}.

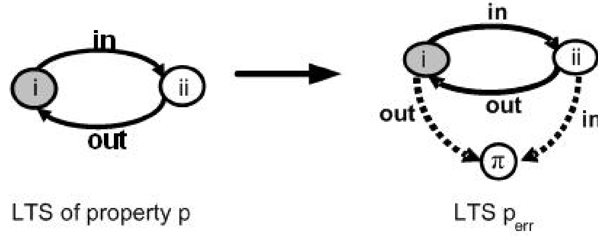


Figure 3.4: Order property.

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.

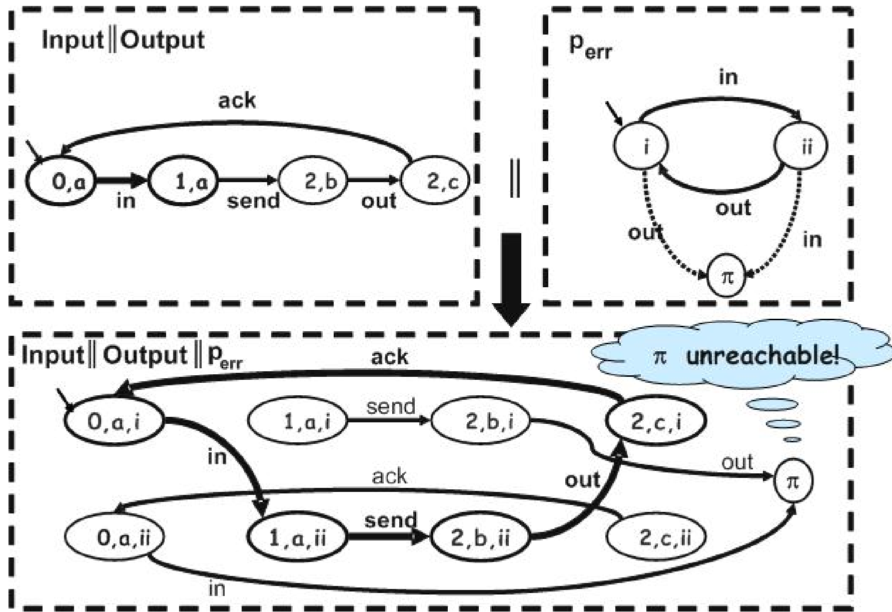


Figure 3.5: Computing the composition $Input \| Output \| p_{err}$.

For example, in order to verify the composition system $Input \| Output$ (illustrated in Figure 3.2) which satisfies the property p (illustrated in Figure 3.4), the parallel composition $Input \| Output \| p_{err}$ is computed in Figure 3.5. It's easy to check that the error state π is not reachable in this composition, so we conclude that the composition system $Input \| Output$ satisfies the property p.

3.1.2 Deterministic Finite State Automata

The approach in this thesis uses a learning algorithms called L* [1, 24] to generate an assumption from two components. A framework to generate assumption will be described in Section 3.2. In this framework (see Figure 3.10), at each iteration i, the *Learning* module produces a Deterministic Finite State Automata (DFA) M_i such that it is unique and minimal automata and $L(M_i) = L(A_W)$, where A_W is the weakest assumption under which F satisfies the property p [7]. The DFA M_i then is transformed into a candidate assumption A_i, where A_i is represented by a safety LTS. A Deterministic Finite State Automata is defined as follows:

Definition 3.5 (***Deterministic Finite State Automata***). 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.

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 $L(M) = \{\sigma \mid \delta(q_0, \sigma) \in F\}$.

Figure 3.6 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 $aaaab \notin L(M)$.

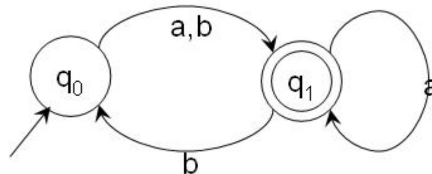


Figure 3.6: An illustration of DFA.

A DFA M is *prefix-closed* if $L(M)$ is prefix-closed, i.e., for every $\sigma \in L(M)$, every prefix of σ is also in $L(M)$. The DFAs returned by the learning algorithm in the proposed

approach are *complete, minimal, and prefix-closed.* These DFAs therefore contains a single non-accepting state *nas*.

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

Let a DFA $M = \langle Q \cup \{nas\}, \alpha M, \delta, q_0, Q\rangle$, the safety LTS $A = \langle Q, \alpha M, \delta \cap (Q \times \alpha M \times Q), q_0\rangle$. For example, Figure 3.7 describes an example to transform a DFA into a safety LTS.

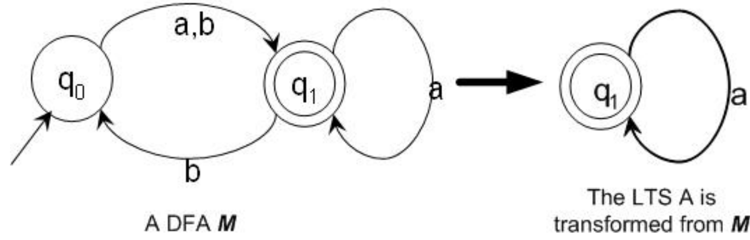


Figure 3.7: An illustration of getting a safety LTS from a DFA.

3.1.3 Component Refinement

Refinement is an important concept in software engineering. It's 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, refinement expresses the relationship between a specification and it's implementation. In this case, refinement means that more detailed information is added. The relation "A_I refines A_S" is intuitively meant to say that "component A_S has more behavioral options than component A_I," or equivalently, "every behavioral option realized by implementation A_I is allowed by specification A_S". In the object-oriented programming, refinement means adding some methods or attributes or constraints into a class. In the open incremental model checking (OIMC) approach [6, 17, 20], refinement means adding (or plugging) an new component (*extension*) into the *Base* component via compatible interface states.

In the proposed approach in this thesis, concept of *refinement* means adding some behavior into the component. Intuitively, component C_2 refines component C_1 meant to say that the component C_2 is created by adding some states and transitions into the component C_1, where C_1 and C_2 are represented by LTSs. Formally, we can define the refinement relation between C_1 and C_2 as follows:

Let $C_1 = \langle Q_1, \alpha C_1, \delta_1, q_0^1\rangle$ and $C_2 = \langle Q_2, \alpha C_2, \delta_2, q_0^2\rangle$ are two components. C_2 is a refinement of C_1 if and only if $Q_1 \subseteq Q_2$, $\delta_1 \subseteq \delta_2$, and $q_0^1 = q_0^2$. For these conditions, $L(C_1) \subseteq L(C_2)$.

For example, the component C_2 is a refinement of component C_1 illustrated in Figure 3.8. After some data is sent to C_1, it produces output using the action *out* and acknowledges that it has finished, by using the action *ack*. The refinement component C_2 is

created by adding the transition (b,*send*,b) into component C_1. It means that C_2 allows multiple *send* actions to occur before producing output.

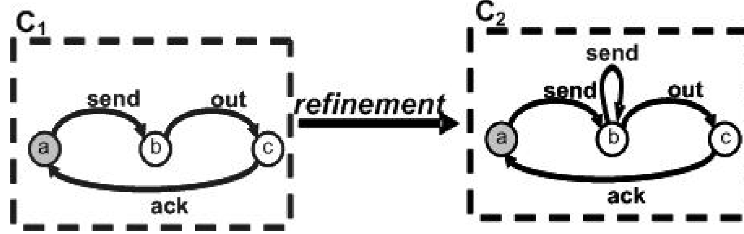


Figure 3.8: An illustration of component refinement.

3.1.4 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 $\langle A \rangle\ M\ \langle p \rangle$, where M is a component, p is a property and A is an assumption about M's environment. The formula is true if whenever M is part of a system satisfying A, then the system guarantees p. In the proposed approach, A, M, p are represented by LTSs.

Consider for simplicity a system that is made up of components M_1 and M_2. The main goal of assume-guarantee reasoning is to verify this system satisfy property p (i.e., $M_1 \| M_2 \models p$?) *without composing* M_1 with M_2. For this purpose, the simplest proof rule consists of showing that the following two premises hold: $\langle A \rangle\ M_1\ \langle p \rangle$ and $\langle \text{true} \rangle\ M_2\ \langle A \rangle$. From these, the rule infers that $\langle \text{true} \rangle\ M_1 \| M_2\ \langle A \rangle$ also holds. Formally, given LTSs M_1 , M_2 and p, assume-guarantee reasoning finds a LTS A such that $L(A \| M_1) \uparrow \alpha p \subseteq L(p)$ and $L(M_2) \uparrow \alpha A \subseteq L(A)$.

Note that for this rule to be useful, the assumption A must be more abstract than M_2, but still reflect M_2's behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for M_1 to satisfy p. Unfortunately, it is often difficult to find such an assumption.

3.1.5 LTSA Tool

The Labelled Transition Systems Analyzer (LTSA) [12] 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 behaviour of composite components based on the behaviour of their immediate children in the hierarchy. Abstraction consists of hiding the actions that do not belong to the interface of a component, and minimizing with respect to observational equivalence.

The input language "*Finite State Processes (FSP)*" of this tool is a process-algebra style notation with Labelled Transition Systems (LTS) semantics. A property is also expressed as an LTS, but with extended semantics, and is treated as an ordinary component during composition. Properties are combined with the components to which they refer. They do not interfere with system behaviour, 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 behaviour models, all helpful aids in both design and verification of system models.

This thesis uses LTSA tool to check correctness of the proposed approach by some illustrations. At each iteration i in framework for assumption generation and new assumption re-generation, this tool is used to check whether assumption A_i produced by L* Learner which satisfies the compositional rule.

3.2 Motivation

Despite significant advances in the development of model checking, it remains a difficult task in the hands of experts to make it scale to the size of industrial systems. A key step in achieving scalability is to "*divide-and-conquer*", i.e., breaking up the verification of a system into smaller tasks that involve the verification of its components. Assume-guarantee reasoning [21, 23] is a widespread "*divide-and-conquer*" approach that uses assumptions when checking individual components of a system. There are many successful researches [2, 4, 7, 8, 17, 19, 20] using this key step. However, there are many limited problems from these approaches (see Section 2.1, 2.2, 2.3) and many open problems which are under research (see Section 2.4).

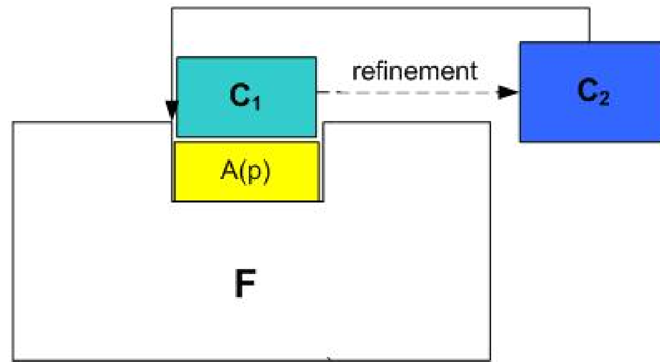


Figure 3.9: A framework for component refinement.

The main goal in this thesis is to find a faster approach for component-based software verification in the context of component refinement. The problem in this thesis is illustrated in Figure 3.9. Suppose that there is a fixed framework F and an extension C_1. The extension C_1 is plugged with the framework F via parallel composition operator (synchronizing the common actions and interleaving the remaining actions). Firstly, we

know that the system contains of F and C_1 satisfy the property p (i.e., $F \| C_1 \models p$). After that, C_1 is refined into C_2 by adding some states and transitions into C_1. The major goal of the proposed method is to verify whether the new composition system $F \| C_2$ satisfies p *without re-checking* it from the beginning. This technique tries to reuse the results of the previous verification (between F and C_1) in order to have an incremental verification manner to verify the new system.

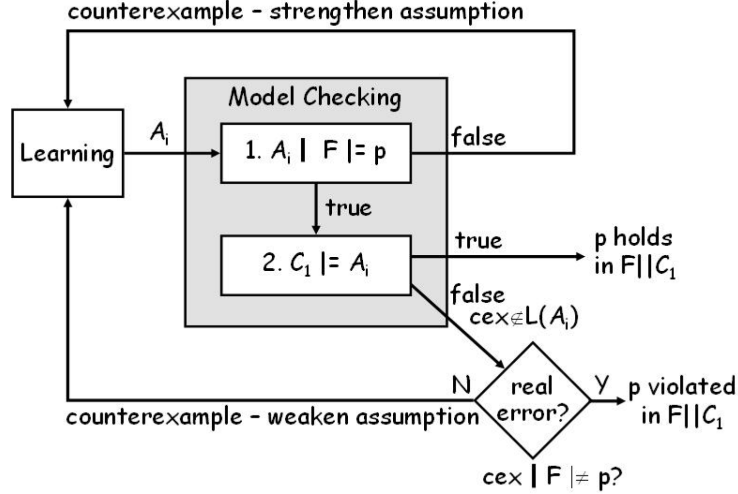


Figure 3.10: A framework to generate assumption [4, 8].

For above goal, this thesis proposes an approach to solve the problem. In a general view of the proposed approach, when we verify the old system $F \| C_1$ satisfying the property p, we generate an assumption $A(p)$ that is strong enough for F to satisfy p but weak enough to be discharged by C_1 (i.e., $\langle A(p) \rangle$ F $\langle p \rangle$ and $\langle \text{true} \rangle$ C_1 $\langle A(p) \rangle$ both hold). This approach uses the learning algorithms L* [4] to generate $A(p)$.

After that, when the component C_1 is refined into C_2, the proposed approach doesn't re-check the whole new system containing the framework F and the new component C_2. It only checks the assume-guarantee formula $\langle \text{true} \rangle$ C_2 $\langle A(p) \rangle$. We hope that if the difference between C_1 and C_2 is small, the formula still holds and the new system still satisfies the property p. There are two cases which may be occurred when checking the formula as follows:

- The first case: the formula $\langle \text{true} \rangle$ C_2 $\langle A(p) \rangle$ still holds. It means that the new system $F \| C_2$ satisfies the property p. In this case, we can seem the assumption $A(p)$ as the results of previous verification to check the new system with future changes in an incremental manner. In practice, if the difference between C_1 and C_2 is small then probability for C_2 to still satisfies p is great. We hope that the system will hold the property p with the assumption $A(p)$ for future changes.
- The second case: the formula $\langle \text{true} \rangle$ C_2 $\langle A(p) \rangle$ doesn't hold. In this case, the proposed approach then performs some analysis to determine whether p is indeed violated in the new system $F \| C_2$ or whether $A(p)$ is too strong for C_2 to satisfy. If

the assumption $A(p)$ is too strong, the the approach re-generates a new assumption $A_{new}(p)$ between F and C_2 *without re-generating* its from the beginning. In the framework to generate assumption using the L* learning algorithms illustrated in Figure 3.10, at the initial step, L* sets the observation table (S,E,T) as the empty table (i.e., L* sets S and E to $\{\lambda\}$). Therefore, the initial assumption A_0 as the strongest assumption (i.e., $A_0 = \lambda$, where λ presents the empty string). In the proposed approach, after generating the assumption $A(p)$ between F and C_1, it saves the observation table (S,E,T) as the results of previous verification for new assumption re-generation in an incremental manner. In order to re-generate new assumption $A_{new}(p)$ between F and C_2, this approach uses the initial observation table as (S,E,T) (not empty table). It means that the initial assumption in the proposed approach to re-generate new assumption is $A(p)$ (not λ). Because $A(p)$ is weaker than λ so much, therefore, we can compute $A_{new}(p)$ in the faster method illustrated in Figure 3.11.

After computing the new assumption $A_{new}(p)$, we can consider it as the intersection between C_1 and C_2 and use it as the results of previous verification for future changes.

Figure 3.11 illustrates the process for assumptions re-generation using the L* learning algorithms. In this figure, λ as strongest assumption. It is initial assumption in L* to generate old assumption $A(p)$ between F and C_1. In the case C_2 doesn't satisfy p (i.e., $C_2 \not\models p$), the assumption $A(p)$ as initial assumption to re-generate new assumption $A_{new}(p)$ between F and C_2. Intuitively, we can find $A_{new}(p)$ in faster manner with the initial assumption $A(p)$. It is difference between the proposed approach and these approaches in [2, 4, 7, 8].

3.3 Related works

Assume-guarantee reasoning leverages the observation that verification techniques can analyze the individual components of large software in isolation to improve performance. For verification of component-based software, there are many approaches proposed and some tools developed; see for example [2, 4, 7, 8].

Even though the proposed approach in this thesis is based on component-based modular model checking, there is a fundamental difference between the conventional modular verification works [10, 11, 22] and the approach including this thesis and in [2, 4, 7, 8]. Modular verification in the former works is rather closed. It 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 re-checked altogether. On the contrary, the proposed approach in this thesis and in [2, 4, 7, 8] verify global system properties by checking individual components in isolation. In its simplest form, it checks whether a component M guarantees a property p when its external environment satisfies an assumption A, and checks that the remaining components in the system (M's environment) indeed satisfy A. The proposed technique therefore is incremental modular verification.

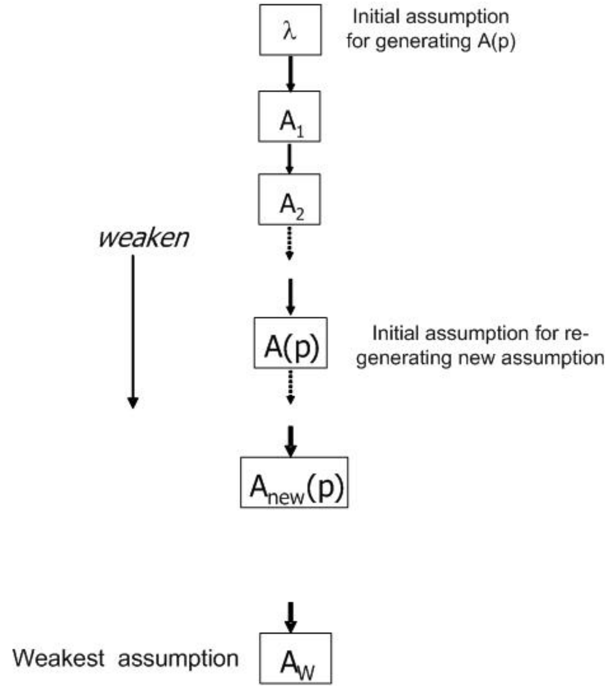


Figure 3.11: The new assumption re-generation process using L*.

The proposed approach introduced in this thesis is similar to [2, 4, 7, 8]. However, my work differs these approaches presented in [2, 4, 7, 8] in some key points. Firstly, my work presents a faster assume-guarantee approach to verify component-based systems in context of component refinement. There is a strong relationship between two components C_1 and C_2, where C_2 is refinement of C_1. Therefore, the proposed approach opens for future changes. On the contrary, component refinement was not mentioned in [2, 4, 7, 8]. Secondly, in the proposed approach, if the component C_1 is refined by component C_2 and if the formula $\langle \text{true} \rangle\ C_2\ \langle A(p) \rangle$ doesn't hold, an new assumption $A_{new}(p)$ is re-generated *without regenerating* its from the beginning. On the contrary, the approach in [2, 4, 7, 8] is viewed from a static perspective, i.e., the component and the external environment do not evolve. If the component changes after adapting some refinements, the assumption-generating approach is re-run on the whole component from beginning, i.e., the component model has to be re-constructed; and the assumption about the environment is then regenerated from that model.

Finally, my work in this thesis is close to the open incremental model checking in [6, 17, 20]. However, the proposed approach differs the approach in [6, 17, 20] in the concept of refinement. The concept of refinement in my approach means adding some states and transitions into the component C_1 whereas the concept of refinement in [6, 17, 20] means adding (or plugging) an new component (extension) into the Base component via compatible interface states. Moreover, in the technique in this thesis components, assumptions and the property are specified by LTSs. In [6, 17, 20] dynamic behavior, component consistency and property were written in the powerful logic CTL [5].

Chapter 4

Assumption Generation using Learning Algorithms - L*

Chapter 3 presented a general view of the proposed approach for component-based software verification in the context of component refinement. An important problem in this technique is that how to generate the assumption, specially, how to re-generate new assumption of the new systems in the case we have to re-generate it. This chapter introduces how to generate assumption between two components F and C_1. The important section in this chapter is the method to re-generate new assumption between F and C_2 in the case the formula $\langle \text{true} \rangle$ C_2 $\langle A(p) \rangle$ doesn't hold. The detailed information of the L* learning algorithms is also presented in this chapter.

4.1 The L* Learning Algorithms

The proposed approach uses the learning algorithms developed by Angluin [1] and later improved by Rivest and Schapire [24]. In this thesis, I refer to the improved version by the name of the original algorithm called L*. L* learns an unknown regular language and produces a DFA that accepts it. The main idea of the L* learning algorithms is based on the "*Myhill-Nerode Theorem*" [13] 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: $w \approx w'$ iff $\forall u \in \Sigma^*$: $wu \in U \iff w'u \in U$. 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 $\sigma \in \Sigma^*$; 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 algorithms believes to be identical to 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 4.1.

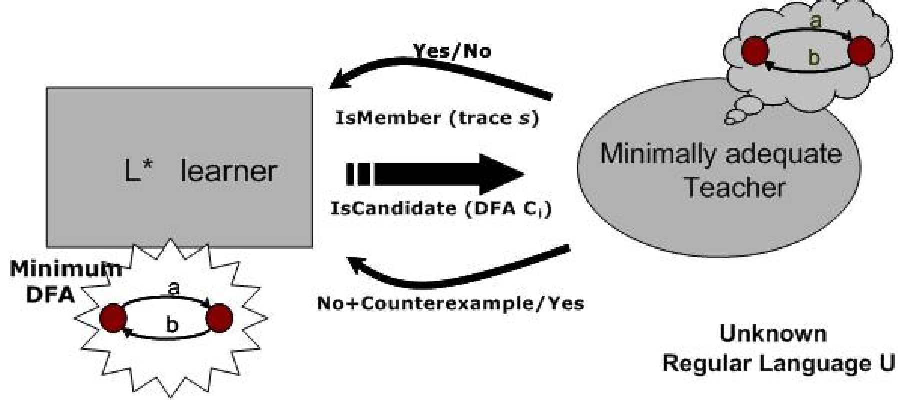


Figure 4.1: The interaction between L* Learner and the Teacher.

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), where:

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

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

The detailed information of the L* algorithms step by step is presented in Figure 4.2, 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 2). It then checks whether the observation table (S, E, T) is closed. 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 3). Because sa has been added to S, T must be re-updated by making membership queries (line 4). Line 3 and line 4 are repeated until the table (S,E,T) is closed.

When the observation table (S,E,T) is closed, a candidate DFA $M = \langle Q, \alpha M, \delta, q_0, F\rangle$ is constructed (line 5) from the 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 6). If the Teacher replies true (i.e., $L(M) = U$), L* returns M as correct (line 7), otherwise it receives an counterexample $c \in \Sigma^*$ from the Teacher.

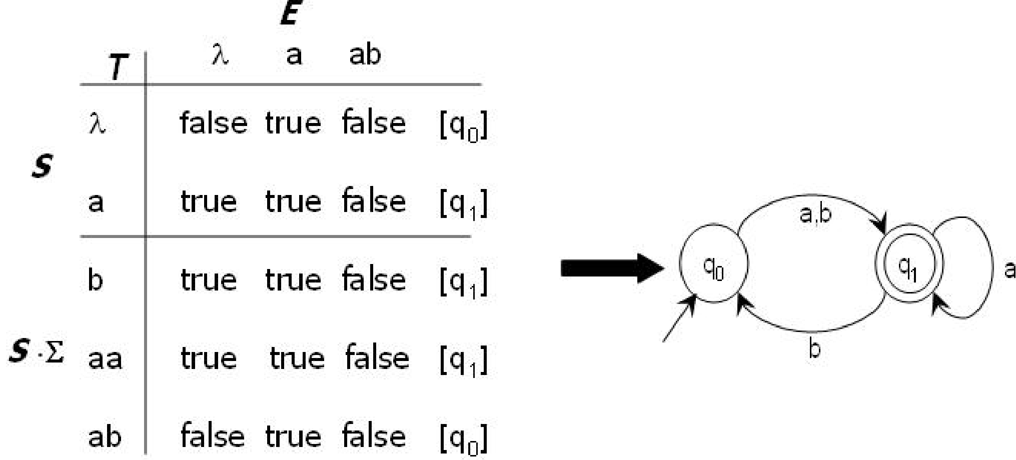
The counterexample c is analyzed by L* to find a suffix e of c that witnesses a difference between $L(M)$ and U. After that, e must be added to E (line 8). It will cause the next conjectured automaton to reflect this difference. When e has been added to E, L* iterates the entire process by looping around to line 2.

```
1.    Initially: S={λ}, E= {λ}
      Loop {
2.       update T using membership queries
         while (S,E,T) is not closed {
3.                 Add sa to S to make S closed where s∈S and a∈∑
4.                 Update T using membership queries
         }
5.       a closed (S,E,T) construct a candidate DFA M from (S,E,T)
6.       present an equivalence query: L(M) = U?
7.       if M is correct return M
8.       else add e∈∑* that witnesses the counterexample to E
      }
```

Figure 4.2: The L* Algorithms.

For example, Figure 4.3 presents a closed observation table and its candidate DFA constructed from this table. It's 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 this 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.



	T	λ	a	ab	
S	λ	false	true	false	[q0]
	a	true	true	false	[q1]
S·Σ	b	true	true	false	[q1]
	aa	true	true	false	[q1]
	ab	false	true	false	[q0]

Figure 4.3: An illustration of a closed observation table (S,E,T) and its candidate DFA.

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 [13], 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 $\mathcal{O}(kn^2 + nlogm)$, 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.

4.2 Assumption Generation using The L* Learning Algorithms

Supposing that there are two components; a framework F and an extension C_1. The extension C_1 is plugged with the framework F via parallel composition operator (synchronizing the common actions and interleaving the remaining actions). Figure 4.4 shows a general view of assume-guarantee verification. The main goal of assume-guarantee verification is to verify this system satisfy the property p (i.e., $F \| C_1 \models p$?) *without composing*

F with C_1. For this purpose, the the approach generates an assumption $A(p)$ that is strong enough for F to satisfy p but weak enough to be discharged by C_1 (i.e., $\langle A(p)\rangle$ F $\langle p\rangle$ and $\langle \text{true}\rangle$ C_1 $\langle A(p)\rangle$ both hold). Unfortunately, it is often difficult to find such an assumption. Formally, given LTSs F , C_1 and p, the main goal in this problem is to find a LTS $A(p)$ such that $L(A(p)\|F)\uparrow\alpha p \subseteq L(p)$ and $L(C_1)\uparrow\alpha A(p) \subseteq L(A(p))$.

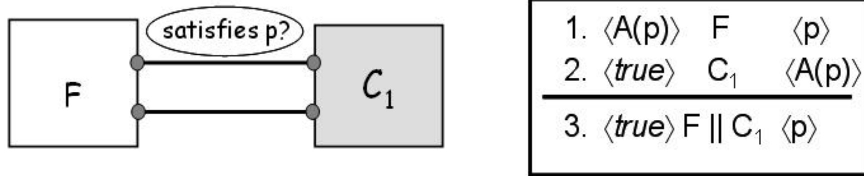


Figure 4.4: A general view of assume-guarantee verification.

Recently, there are two solutions in generating assumption $A(p)$ automatically. The first one is an algorithmic, non-incremental, generation of assumptions. It finds the weakest assumption A_W by taking the complement of paths in the product automata leading to error states. The A_W describes exactly those traces over $\Sigma = (\alpha F \cup \alpha p) \cap \alpha C_1$ which do not lead to the error state π in $F\|p_{err}$. For all component C_1', $F\|C_1' \models p$ iff $C_1' \models A_W$. The drawback in this solution is that if the computation runs out of memory, i.e., if component state space is too large, no assumption will be obtained as a result. The advantage is that it does not require knowledge of the environment. We want 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 solution - assumption generation using the learning algorithms called L* [1, 24]. It is an incremental approach, based on counterexamples and learning. Instead of finding A_W, it uses the L* learning algorithms to learn A_W. The advantage of this solution 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, it requires knowledge of the environment and is quite difficult to understand.

In order to obtain appropriate assumptions, the method applies the compositional rule in an iterative fashion illustrated in Figure 4.6. 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 rule are then applied. Step 1 is applied first, to check whether F satisfies p in environments that guarantee A_i by computing formula $\langle A_i\rangle$ F $\langle p\rangle$. 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. Therefore, the candidate assumption A_i must be strengthened, which corresponds to removing behaviors from it, with the help of the counterexample *cex* produced by step 1. In the context of the next candidate assumption A_{i+1}, component F should at least not exhibit the violating behavior reflected by this counterexample. If step 1 returns true, it means that A_i is strong enough for F to satisfy the property p. The step 2 is then applied to check component C_1 satisfying A_i by computing formula $\langle \text{true}\rangle$ C_1 $\langle A_i\rangle$. If step 2 returns true,

the property p holds in the composition system $F \| C_1$. In this case, the system $F \| C_1 \models p$ is verified. Otherwise, further analysis is required to identify whether p is indeed violated in $F \| C_1$ or whether A_i is too strong for C_1 to satisfy. Such analysis is based on the counterexample *cex* returned by step 2. It must checks that whether the counterexample *cex* belong to the unknown language $U = L(A_W)$ (i.e., $cex \in L(A_W)$?). For this purpose, this analysis checks whether p violated by F in the context of the counterexample *cex* by computing the composition system that contains of A_{cex} and F violate the property p (i.e., $A_{cex} \| F \not\models p$?), where A_{cex} is a LTS defined as follows:

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

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

Figure 4.5 illustrates the LTS A_{cex} is created from the the counterexample *cex*.

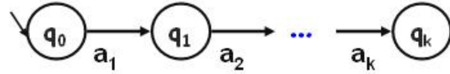


Figure 4.5: The LTS A_{cex} is created from the counterexample *cex*.

If the property p doesn't hold in the composition system $A_{cex} \| F$ (i.e., $A_{cex} \| F \not\models p$), it means that the property p doesn't hold in the composition system $F \| C_1$ (i.e., $F \| C_1 \not\models p$). Otherwise, A_i is too strong for C_1 to satisfy. The candidate assumption A_i therefore must be weakened (i.e., behaviors must be added) in iteration i+1. The result of such weakening will be that at least the behavior that the counterexample *cex* represents will be allowed by candidate assumption A_{i+1}. New candidate assumption may of course be too weak, and therefore the entire process must be repeated.

An important question in this section 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 4.6. In the assume-guarantee approach, L* learns the weakest assumption A_W. It means that L* learns the unknown language $U = L(A_W)$ over the alphabet $\Sigma = \alpha A_W = (\alpha F \cup \alpha p) \cap \alpha C_1$. It uses candidates produced by L* learning as candidate assumptions A_i for the assume-guarantee rule (compositional rule). In order to produce each candidate assumptions 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 3.4. For L* 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

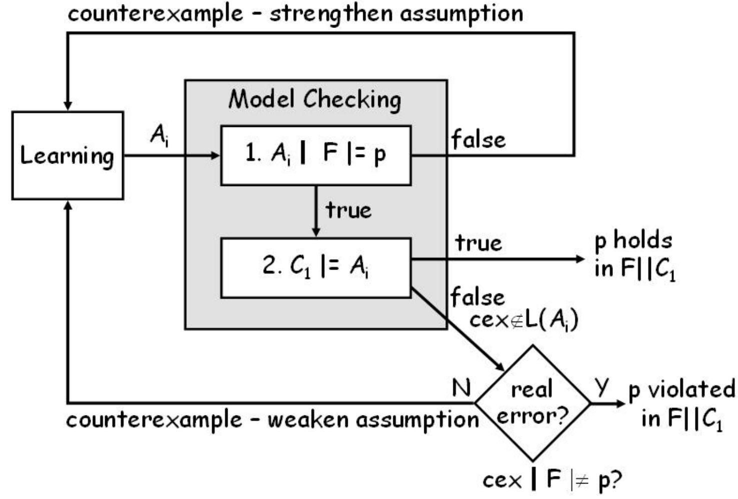


Figure 4.6: A framework to generate assumption using the L* learning algorithms [4, 8].

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 $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ whether in $\Sigma^* = L(A_W)$, the Teacher simulates the query on the composition $F \| p_{err}$. For the string σ, the Teacher first builds safety LTS $A_\sigma = \langle Q, \alpha A_\sigma, \delta, q^0 \rangle$, where $Q = \{q_0, q_1, \ldots, q_n\}$, $\alpha A_\sigma = \Sigma$, $\delta = \{(q_i, a_{i+1}, q_{i+1}) \mid 1 \leq i < n\}$, and $q^0 = q_0$. The Teacher then checks the formula $\langle A_\sigma \rangle\ F\ \langle p \rangle$ by computing the composition system $A_\sigma \| F \| p_{err}$. If the state error π is unreachable in this composition system (the formula returns true), it means that $\sigma \in L(A_W)$. Because F does not violate the property p in the context of σ, so the Teacher returns true. Otherwise, the answer to the membership query is false.

For the second type of questions, with each DFA M_i produced by L* from the observation table (S,E,T) at each iteration i, the Teacher must checks that whether the DFA M_i is a candidate DFA for the iteration i (i.e., $L(M_i) = L(A_W)$?) 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 rule. The Teacher applies two steps of the compositional rule and counterexample analysis to answer conjectures as follows:

- Step 1 illustrated in Figure 4.6 first is applied, the Teacher checks the formula $\langle A_i \rangle$ F $\langle p \rangle$ by computing the composition system $A_i \| F \| p_{err}$. If the state error π is reachable in this composition system, it means that this formula doesn't 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 $\langle \text{true} \rangle$ C_1 $\langle A_i \rangle$ illustrated in Figure 4.6. 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 composition system $F \| C_1$. Otherwise, this step returns a counterexample *cex*. The Teacher then performs some analysis to determine whether p is indeed violated in $F \| C_1$ or whether A_i is too strong for C_1 to satisfy.

- 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 Step 2. The Teacher first creates a safety LTS $A_{cex\uparrow\Sigma}$ from the counterexample *cex* illustrated in Figure 4.5. The Teacher then checks the formula $\langle A_{cex\uparrow\Sigma} \rangle$ F $\langle p \rangle$ by computing the composition system $A_{cex\uparrow\Sigma} \| F \| p_{err}$. If the state error π is unreachable, then the property p doesn't hold in the composition system $F \| C_1$ (i.e., $F \| C_1 \not\models p$). Otherwise, A_i is too strong for C_1 to satisfy in the context of *cex*. The $cex{\uparrow}\Sigma$ is returned as a counterexample for conjecture A_i.

4.3 New Assumption Re-generation

When the component C_1 is refined into new component C_2 by adding some states and transitions into C_1, the proposed approach must re-check the new system which contains of the framework F and the new component C_2 whether it satisfies the property p. For this purpose, the approach in this thesis only checks the formula $\langle \text{true} \rangle$ C_2 $\langle A(p) \rangle$. If it holds, the new system $F \| C_2$ satisfies the property p. Otherwise, the Teacher returns a counterexample $cex{\uparrow}\Sigma$ to witness this fact. The proposed approach then performs some analysis to determine whether p is indeed violated in the new system $F \| C_2$ or whether $A(p)$ is too strong for C_2 to satisfy. If $A(p)$ is too strong, a new assumption $A_{new}(p)$ between the framework F and the new component C_2 is re-generated that is strong enough for F to satisfy p but weak enough to be discharged by C_2 (i.e., $\langle A_{new}(p) \rangle$ F $\langle p \rangle$ and $\langle \text{true} \rangle$ C_2 $\langle A_{new}(p) \rangle$ both hold). How to re-generate new assumption in faster method is the major goal in this thesis.

The approach in [2, 4, 8] is viewed from a static perspective to re-generate new assumption. If the component changes after adapting some refinements, the assumption-generating approach is re-run on the whole component from beginning, i.e., the component model has to be re-constructed; and the assumption about the environment is then re-generated from that model. This approach therefore is not effective for re-checking the new system in the case it must re-generate a new assumption.

This thesis proposes a faster assume-guarantee approach for component-based software verification in the context of component refinement. The proposed approach re-generates a new assumption $A_{new}(p)$ between F and C_2 *without re-running* on these whole components from beginning. It tries to reuse the results of the previous verification (between F and C_1) in order to have an incremental manner to re-generate the new assumption.

In the algorithms L* illustrated in Figure 4.6, at the initial step, L* sets the observation table (S,E,T) as empty table (i.e., L* sets S and E to $\{\lambda\}$, where λ presents the empty string). Therefore, the initial assumption A_0 is the strongest assumption λ (i.e., $A_0 = \lambda$). In the proposed approach, after generating the assumption $A(p)$ between F and C_1, it saves the observation table (S,E,T) as the results of previous verification of the old system $F \| C_1$. To re-generate new assumption $A_{new}(p)$ between F and C_2, the approach in also uses the learning algorithms L* but with the initial observation table (S,E,T) (not empty table as in [2, 4, 8]), from now on called the old observation table (S_{old}, E_{old}, T_{old}). It means that the initial assumption in the proposed approach to re-generate new assumption is $A(p)$ (not λ). Because $A(p)$ is weaker than λ so much, our technique therefore can compute $A_{new}(p)$ in the faster method illustrated in Figure 3.11). In the proposed technique, the results of the previous verification (between F and C_1) is set to be the initial assumption $A(p)$ to re-generate new assumption $A_{new}(p)$ in incremental manner.

After re-generating the new assumption $A_{new}(p)$, we can consider $A_{new}(p)$ as the intersection between C_1 and C_2. In the future, if the component C_2 will be changed after adapting some refinements, we hope that the new system will hold the property p with the new assumption $A_{new}(p)$ (i.e., $\langle \text{true} \rangle$ C_2 $\langle A_{new}(p) \rangle$ still holds). If it will not hold, we will re-generate new assumption in the faster manner with initial assumption as $A_{new}(p)$.

The detailed information of the L* learning algorithms for new assumption re-generation step by step is presented in Figure 4.7, line numbers refer to L*'s illustration. 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}, T to T_{old}) (line 1). The counterexample $cex{\uparrow}\Sigma$ returned by the Teacher when it check the formula $\langle \text{true} \rangle$ C_2 $\langle A(p) \rangle$ is analyzed by L* 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 3). It then checks whether the observation table (S,E,T) is closed. 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 4). Because sa has been added to S, T must be re-updated by making membership queries (line 5). Line 4 and line 5 are repeated until the table (S,E,T) is closed.

When the observation table (S,E,T) is closed, a candidate DFA $M = \langle Q, \alpha M, \delta, q_0, F \rangle$ is constructed (line 6) from the table (S,E,T) by the same way in the section 5.2.

The candidate DFA M is presented as a conjecture to the Teacher (line 7). If the Teacher replies true (i.e., $L(M) = U$), L* returns M as correct (line 8), otherwise it receives an counterexample $cex \in \Sigma^*$ from the Teacher.

The counterexample cex is analyzed by L* to find a suffix e of c that witnesses a difference between $L(M)$ and U. After that, e must be added to E (line 9). 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.

1. Initially: $S=S_{old}$, $E=E_{old}$, $T=T_{old}$
2. Add $e \in \Sigma^*$ that witnesses the **counterexample** to E

Loop {

3. update T using **membership queries**

while (S,E,T) is **not closed** {

4. Add sa to S to make S closed where $s \in S$ and $a \in \Sigma$
5. Update T using **membership queries**

}

6. a closed (S,E,T) construct a **candidate** DFA M from (S,E,T)
7. present an **equivalence query**: $L(M) = U$?
8. if M is **correct** return M
9. else **add** $e \in \Sigma^*$ that witnesses the **counterexample** to E

}

Figure 4.7: The L* learning algorithms for new assumption regeneration.

Correctness and termination of the approach for new assumption re-generation is proved by following theorem.

Theorem 1. Given a framework F, a new component C_2 which is a refinement of the extension C_1, property p, and an assumption $A(p)$ which is strong enough for F to satisfy p but weak enough to be discharged by C_1, the algorithms for new assumption re-generation in our approach terminates and it returns true and a new assumption $A_{new}(p)$ if the new composition system $F \| C_2$ satisfy p and false otherwise.

Proof. The proposed approach uses two steps of the compositional rule to answer the question of whether the assumption A_i produced by L* Learner is an candidate assumption. It only returns true and new assumption $A_{new}(p)=A_i$ when both steps return true, and therefore correctness is guaranteed by the compositional rule in [4]. Our technique returns a real error when it detects a trace σ of C_2 which violates the property p when simulated on F. In this case, it implies that $F \| C_2$ violates p. The remaining problem is to prove that we always achieve the new assumption $A_{new}(p)$ from the old assumption $A(p)$ if $F \| C_2$ satisfy p. In the case the new assumption is re-generated, we know that $C_2 \not\models A(p)$ because $A(p)$ is too strong for C_2 to satisfy. We also know that $C_2 \models A_{new}(p)$ because $A_{new}(p)$ satisfies the compositional rule. From these, it means that $A_{new}(p)$ is weaker than $A(p)$. In the proposed approach, the assumption A_i produced by L* Learner at *iteration i* is always stronger than the assumption A_{i+1} at *iteration (i+1)*, so we always achieve the new assumption $A_{new}(p)$ from $A(p)$. About termination, at any iteration, the algorithm returns true or false (i.e., $F \| C_2 \not\models p$) and terminates or continues by providing a counterexample to L* Learner. By correctness of L* [1, 24], we are guaranteed that if L* Learner keep receiving counterexamples, in the worst case, our algorithms will eventually produce the weakest assumption A_W and terminates by definition of A_W.

Figure 4.8 describes the process for new assumptions re-generation by using the algorithms L* in the proposed approach in a general view. In this figure, λ is strongest assumption. It is initial assumption in L* to generate assumption $A(p)$ between F and C_1. In the case C_2 doesn't satisfy p (i.e., $C_2 \not\models p$), we use the assumption $A(p)$ as initial assumption to re-generate new assumption $A_{new}(p)$ between F and C_2. Because the old assumption $A(p)$ is too strong for C_2 to satisfy, so new assumption $A_{new}(p)$ is weaker than $A(p)$. It is very clear that new assumption re-generation by starting at the old assumption $A(p)$ is faster than by starting at the strongest assumption λ. Intuitively, we can find $A_{new}(p)$ in faster manner by initial assumption $A(p)$. It is difference between the proposed approach and approaches in [2, 4, 7, 8].

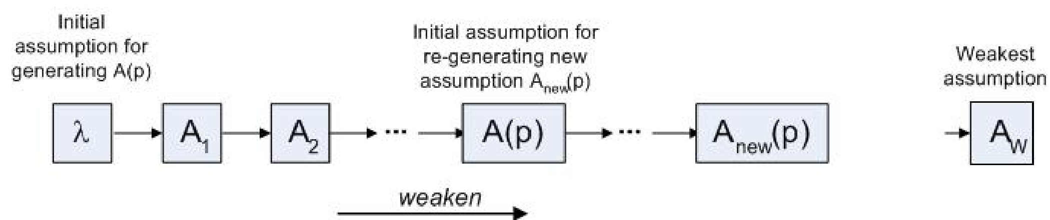


Figure 4.8: The process for new assumption regeneration using L*.

Chapter 5

A Case Study

Chapter 3,4 presented a faster assume-guarantee approach for component-based software verification in the context of component refinement. The contribution in this thesis is to propose an incremental method for new assumption re-generation. This chapter introduces a case study to illustrate the proposed technique, specially, assumption generation method and new assumption re-generation method will be presented step by step by concrete examples. The LTSA tool is used to check assumptions produced by L* Leaner that whether them satisfy the compositional rule.

5.1 System Specification

An illustration system which contains of the framework F and the component C_1 presented in Figure 5.1. In this illustration system, the LTS of the framework F as *Input* LTS, and the LTS of component C_1 as *Output* LTS. The initial state of *Input* LTS in this example is *state 0*. The initial state of *Output* LTS is *state a*. The extension C_1 is plugged with the framework F via parallel composition operator (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, *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 *out* action. The LTS p_{err} is created from LTS p by applying the definition 3.4. The dashed arrows illustrate the transitions to the error state that are added to the property p to obtain LTS p_{err}.

Formally, these components and order property are defined as follows:

Let $F = \langle Q_F, \alpha F, \delta_F, q_0^F \rangle$, where:

- $Q_F = \{0, 1, 2\}$,
- $\alpha F = \{in, send, ack\}$,

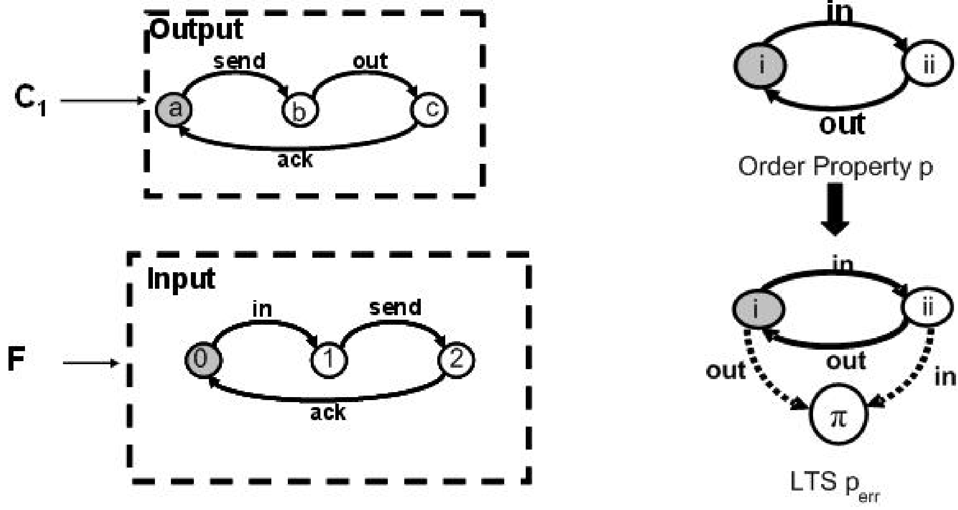


Figure 5.1: Components and order property of the illustration system.

- $\delta_F = \{(0, in, 1), (1, send, 2), (2, ack, 0)\}$, and
- $q_0^F = 0$.

Let $C_1 =< Q_{C_1}, \alpha C_1, \delta_{C_1}, q_0^{C_1} >$, where:

- $Q_{C_1} = \{a, b, c\}$,
- $\alpha C_1 = \{send, out, ack\}$,
- $\delta_{C_1} = \{(a, send, b), (b, out, c), (c, ack, a)\}$, and
- $q_0^{C_1} = a$.

Let $p = \langle Q_p, \alpha p, \delta_p, q_0^p \rangle$, where:

- $Q_p = \{i, ii\}$,
- $\alpha p = \{in, out\}$,
- $\delta_p = \{(i, in, ii), (ii, out, i)\}$, and
- $q_0^p = i$.

The LTS p_{err} is created from LTS p by applying the definition 3.4. Let $p_{err} = \langle Q_{p_{err}}, \alpha p_{err}, \delta_{p_{err}}, q_0^{p_{err}} \rangle$, where:

- $Q_{p_{err}} = \{i, ii\}$,
- $\alpha p_{err} = \{in, out, \pi\}$,

- $\delta_{p_{err}} = \{(i, in, ii), (ii, out, i), (i, out, \pi), (ii, in, \pi)\}$, and
- $q_0^{p_{err}} = i$.

In the proposed approach, we know that the illustration system described in Figure 5.1 satisfies the order property p (i.e., $F \| C_1 \models p$). It will generate an assumption $A(P)$ that is strong enough for F to satisfy p but weak enough to be discharged by C_1 (i.e., $\langle A(p) \rangle$ F $\langle p \rangle$ and $\langle \text{true} \rangle$ C_1 $\langle A(p) \rangle$ both hold) in section 5.2.

After that, when the component C_1 (*Output*) is refined into a new component C_2 (*Output'*), the proposed approach will check the formula $\langle \text{true} \rangle C_2$ $\langle A(p) \rangle$. It will not hold, it will re-generate a new assumption $A_{new}(p)$ between the framework F and the new component C_2 in section 5.3.

The component C_2 is a refinement of component C_1 illustrated in Figure 5.2. After some data is sent to C_1, it produces output using the action *out* and acknowledges that it has finished, by using the action *ack*. The refinement component C_2 is created by adding the transition (b,*send*,b) into component C_1. It means that C_2 allows multiple *send* actions to occur before producing output.

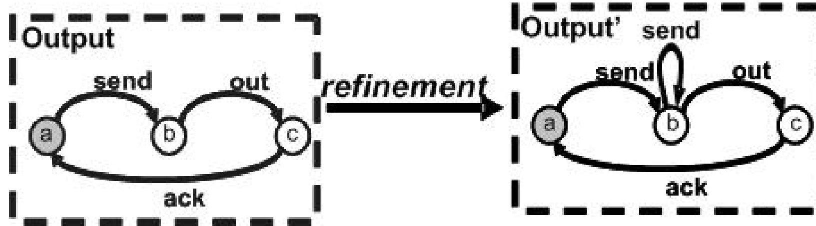


Figure 5.2: The component *Output* is *refined* into new component *Output'*.

5.2 Assumption Generation

The main goal in this section is to find an assumption $A(p)$ between the framework F and the component C_1 that is strong enough for F to satisfy p but weak enough to be discharged by C_1 (i.e., $\langle A(p) \rangle$ F $\langle p \rangle$ and $\langle true \rangle$ C_1 $\langle A(p) \rangle$ both hold). The assumption $A(p)$ will be used as initial assumption to re-generate new assumption $A_{new}(p)$ in section 5.3.

In order to generate assumption $A(p)$, L* learns the weakest assumption A_W. It means that L* learns the unknown language $U{=}L(A_W)$ over the alphabet $\Sigma = \alpha A_W = (\alpha F \cup \alpha p) \cap \alpha C_1 = \{send, out, ack\}$.

Initially, L* sets the observation table (S,E,T) with S and E to $\{\lambda\}$ illustrated in Figure 5.3, where λ presents the empty string. The observation table (S,E,T) is updated by making membership queries to the Teacher, i.e., $\lambda{\in}L(A_W)$?, $\langle \text{ack} \rangle{\in}L(A_W)$?, $\langle \text{out} \rangle{\in}L(A_W)$?, and $\langle \text{send} \rangle{\in}L(A_W)$?.

For example, by making the membership query $\lambda{\in}L(A_W)$? to the Teacher. The Teacher simulates the empty string λ on the composition system $Input \| p_{err}$ illustrated in Figure 5.4. In this case, the error state π is unreachable. It means that $\lambda{\in}L(A_W)$, the Teacher

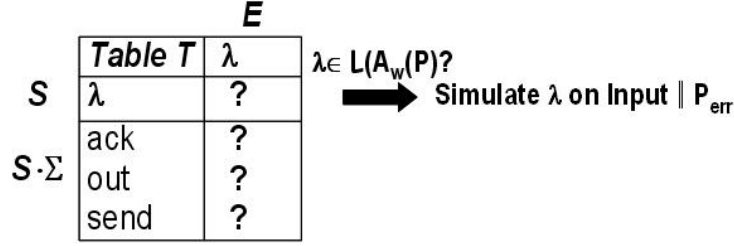


	Table T	λ
S	λ	?
S·Σ	ack	?
	out	?
	send	?

Figure 5.3: The empty observation table at initial step.

therefore replies *Yes* to L* Learner. The similarity with the others queries, the results are that ⟨ack⟩$\in L(A_W)$, ⟨out⟩$\notin L(A_W)$, and ⟨send⟩$\in L(A_W)$. The observation table (S,E,T) after updating illustrated in Figure 5.5.

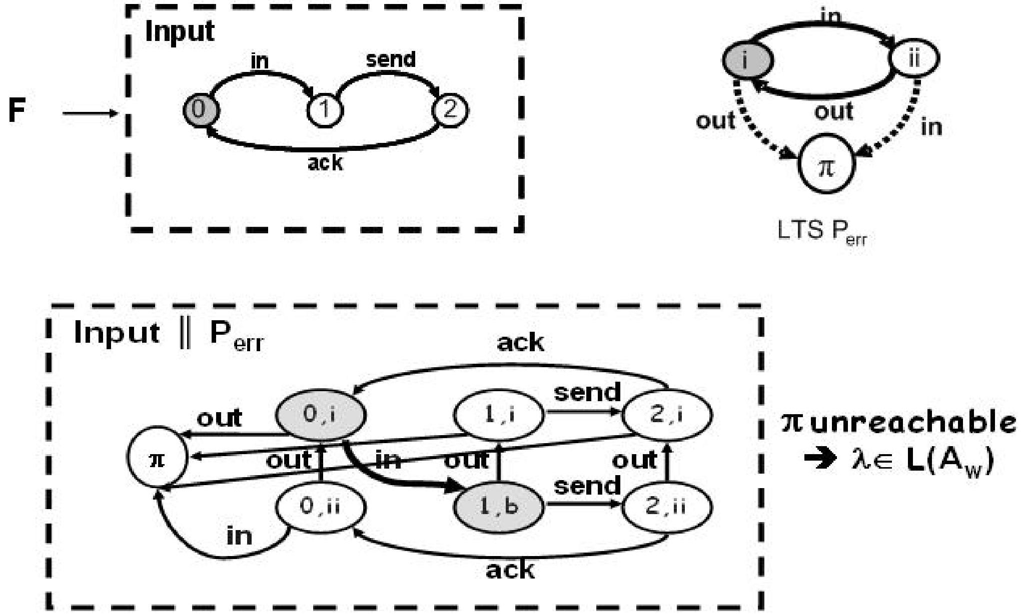


Figure 5.4: Simulation the empty string λ on the composition system $Input \| P_{err}$.

Because the row *out* in $S.\Sigma$ has no matching row in S, so the observation table (S,E,T) in Figure 5.5 is not closed. Adding *out* into S to make its closed. The observation table (S,E,T) after adding *out* into S illustrated in Figure 5.6. The observation table (S,E,T) is re-updated by making membership queries to the Teacher, i.e., ⟨out,ack⟩$\in L(A_W)$?, ⟨out,out⟩$\in L(A_W)$?, and ⟨out,send⟩$\in L(A_W)$?.

For example, by making membership query ⟨out,ack⟩$\in L(A_W)$? to the Teacher. The Teacher simulates the string ⟨out,ack⟩ on the composition system $Input \| P_{err}$ illustrated in Figure 5.4. In this case, the error state π is reachable. It means that ⟨out,ack⟩$\notin L(A_W)$, so the Teacher replies *No* to L* Learner. The similarity with the others queries, the results are that ⟨out,out⟩$\notin L(A_W)$, and ⟨out,send⟩$\notin L(A_W)$. The observation table (S,E,T) after re-updating illustrated in Figure 5.7. Because every row sa in $S.\Sigma$ ($s \in S$ and $a \in \Sigma$) has a matching row s' in S, so the observation table (S,E,T) in Figure 5.7 is closed. A candidate DFA M_1 is constructed from this closed observation table. We get a safety LTS

	Table T	λ (E)
S	λ	true
S·Σ	ack	true
	out	false
	send	true

Figure 5.5: The observation table after updating by making membership queries.

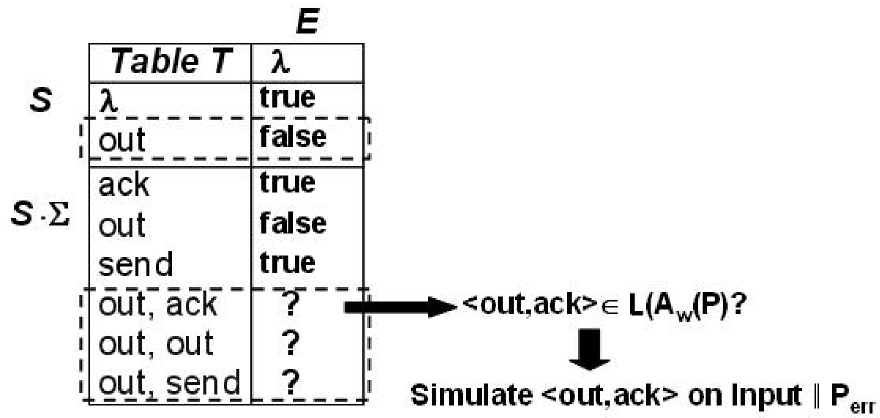


Figure 5.6: The observation table after adding *out* into S.

A_1 from the candidate DFA M_1 by removing the non-accepting state and all its ingoing transitions.

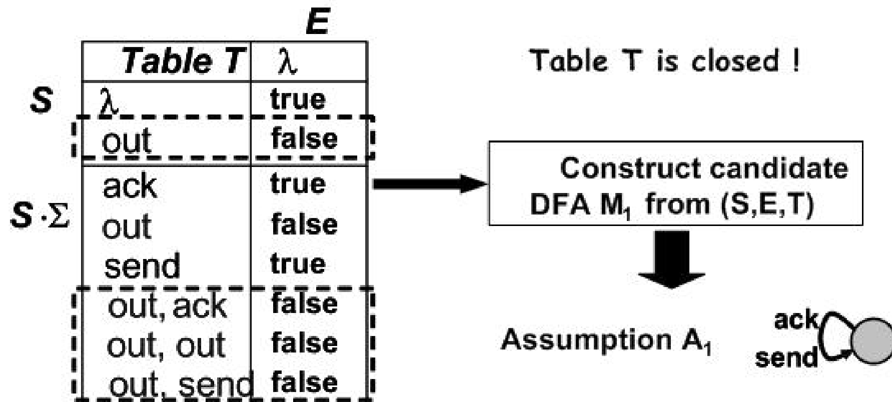


Figure 5.7: The observation table after re-updating by making membership queries.

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

Step 1 first is applied, the Teacher checks the formula $\langle A_1 \rangle$ *Input* $\langle p \rangle$ by computing the composition system $A_1 \| Input \| p_{err}$ illustrated in Figure 5.8. It is easy to check that the error state π is reachable in this composition system, so the 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.

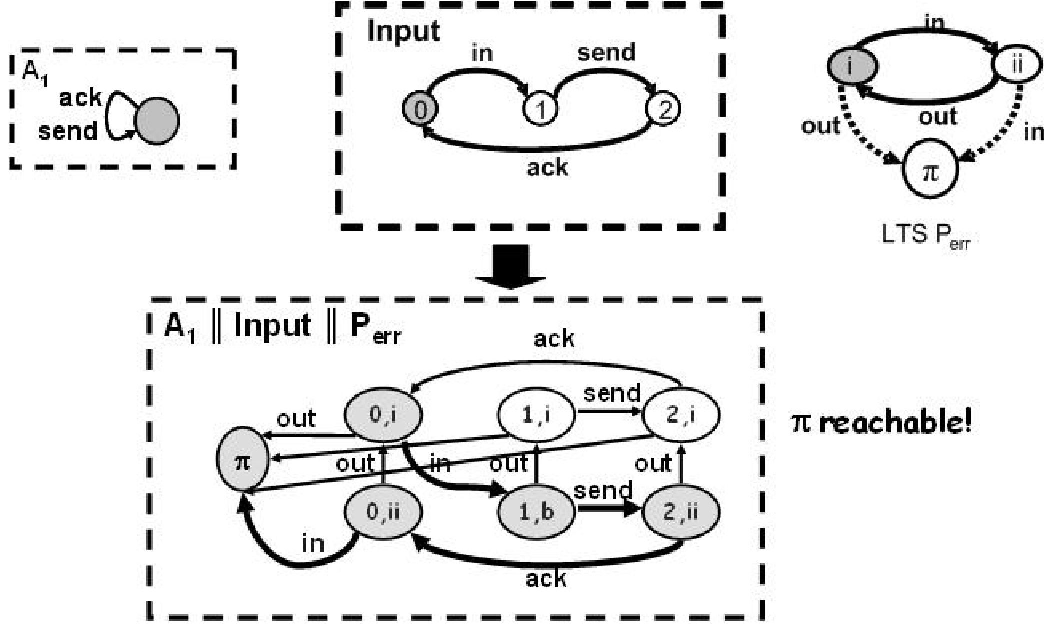


Figure 5.8: Computing the composition system $A_1 \| Input \| P_{err}$.

The counterexample $cex{\uparrow}\Sigma$=⟨send,ack⟩ is analyzed by L* to find a suffix e of cex that witnesses a difference between $L(M_1)$ and U. In this case, L* analyzes and sets e to *ack*. Adding *ack* into E and re-updating the observation table (S,E,T) by making membership queries. The observation table (S,E,T) after adding *ack* into E illustrated in Figure 5.9. The observation table (S,E,T) is re-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 Teacher simulates these strings on the composition system $Input \| p_{err}$ illustrated in Figure 5.4. The results which the Teacher reply to L* are that ⟨ack⟩$\in L(A_W)$, ⟨out,ack⟩$\notin L(A_W)$, ⟨ack,ack⟩ $\in L(A_W)$, ⟨out,ack⟩$\notin L(A_W)$, ⟨send,ack⟩$\notin L(A_W)$, ⟨out,ack,ack⟩$\notin L(A_W)$, ⟨out,out,ack⟩$\notin L(A_W)$, and ⟨out,send,ack⟩$\notin L(A_W)$. The observation table (S,E,T) after re-updating illustrated in Figure 5.10.

Because the row *send* in $S.\Sigma$ has no matching row in S, so the observation table (S,E,T) in Figure 5.10 is not closed. Adding *send* into S to make S closed. The observation table (S,E,T) after adding *send* into S and re-updating by making membership queries to the Teacher illustrated in Figure 5.11. For every row sa in $S.\Sigma$ ($s\in S$ and $a\in\Sigma$) has a matching row s' in S, so the observation table (S,E,T) in Figure 5.11 is closed. A candidate DFA M_2 is constructed from this closed observation table. We get a safety LTS A_2 from the candidate DFA M_2 by removing the non-accepting state and all its ingoing transitions.

The Teacher then uses the safety LTS A_2 as candidate assumption for the compositional rule. The Teacher applies two steps of the compositional rule and counterexample analysis to answer conjectures from L* Learner.

	Table T	E: λ	E: ack
S	λ	true	
	out	false	
S·Σ	ack	true	
	out	false	
	send	true	
	out, ack	false	
	out, out	false	
	out, send	false	

Figure 5.9: The observation table after adding *ack* into *S*.

	Table T	E: λ	E: ack
S	λ	true	true
	out	false	false
S·Σ	ack	true	true
	out	false	false
	send	true	false
	out, ack	false	false
	out, out	false	false
	out, send	false	false

Figure 5.10: The observation table after re-updating by making membership queries.

Step 1 first is applied, the Teacher checks the formula $\langle A_2 \rangle$ *Input* $\langle p \rangle$ by computing the composition system $A_2 \| Input \| p_{err}$ illustrated in Figure 5.12. It is easy to check that the error state π is unreachable in this composition system, so the the Teacher then returns true. It means that the formula $\langle A_2 \rangle$ *Input* $\langle p \rangle$ holds. The Teacher forwards A_2 to Step 2.

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

5.3 New Assumption Regeneration

When the component *Output* is refined into the new component *Output'* illustrated in Figure 5.2. The main goal of the proposed approach is to verify the new system which contains of component *Input* and new component *Output'*. In order to verify the new system, the approach only checks the formula $\langle \text{true} \rangle$ *Output'* $\langle A(p) \rangle$ by computing the composition system $Output' \| A_{2_{err}}$, where the error LTS $A_{2_{err}}$ is created from the safety LTS A_2 by applying the definition 3.4.

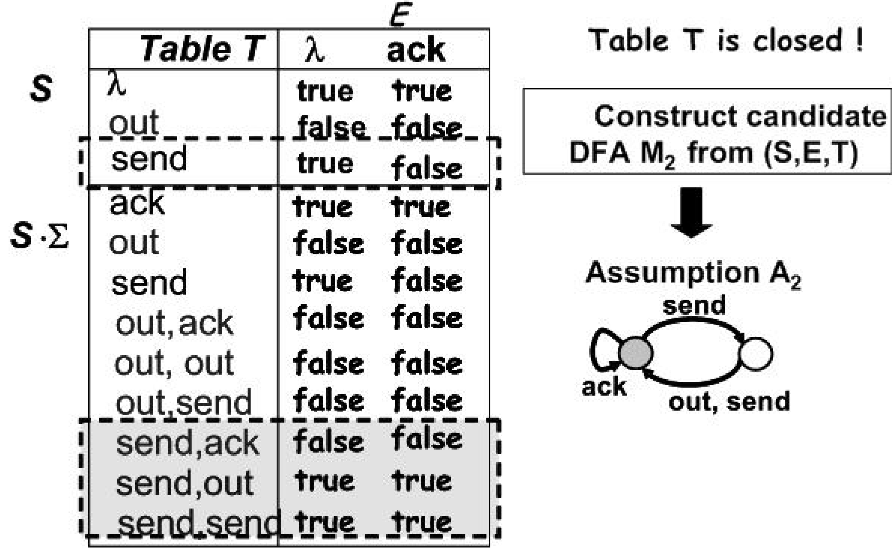


Table T	λ	ack
λ	true	true
out	false	false
send	true	false
ack	true	true
out	false	false
send	true	false
out,ack	false	false
out, out	false	false
out,send	false	false
send,ack	false	false
send,out	true	true
send,send	true	true

Figure 5.11: The observation table after adding *send* into *S*.

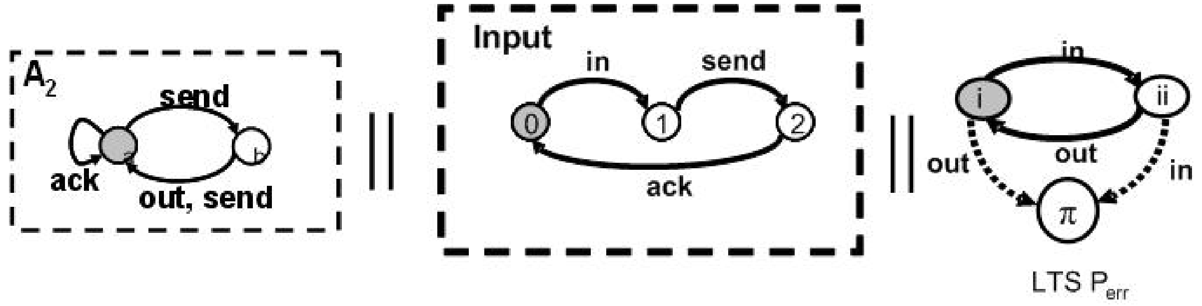


Figure 5.12: Computing the composition system $A_2 \| Input \| p_{err}$.

Figure 5.13 illustrates the computing the composition system *Output'*$\| A_{2_{err}}$. In this composition, the error state π is reachable with trace ⟨send,send,out⟩. It means that the formula ⟨true⟩ *Output'* $\langle A(p) \rangle$ doesn't hold. In this case, the Teacher returns a counterexample cex=⟨send,send,out⟩. The Teacher then performs some analysis to determine whether p is indeed violated in $F \| C_2$ ($Input \|$ *Output'*) or whether A_2 is too strong for *Output'* to satisfy by simulating the counterexample $cex{\uparrow}\Sigma$=⟨send,send,out⟩ on the composition system $Input \| p_{err}$.

Figure 5.14 illustrates simulating the counterexample $cex{\uparrow}\Sigma$ on the composition system $Input \| P_{err}$. In this composition, the error state π is unreachable. It means that the assumption A_2 is too strong for *Output* to satisfy in the context of $cex{\uparrow}\Sigma$=⟨send,send,out⟩. The $cex{\uparrow}\Sigma$ is returned as a counterexample for conjecture A_2.

The approach in this thesis must re-generate a new assumption $A_{new}(p)$ between F and C_2. In order to re-generate new assumption, it also uses the learning algorithms L* but beginning from the old assumption A_2 (not from λ).

The counterexample $cex{\uparrow}\Sigma$=⟨send,send,out⟩ is analyzed by L* to find a suffix e of cex that witnesses a difference between $L(M_2)$ and U. In this case, L* analyzes and sets e to *out*. Adding *out* into E and re-updating the observation table (S,E,T) by making

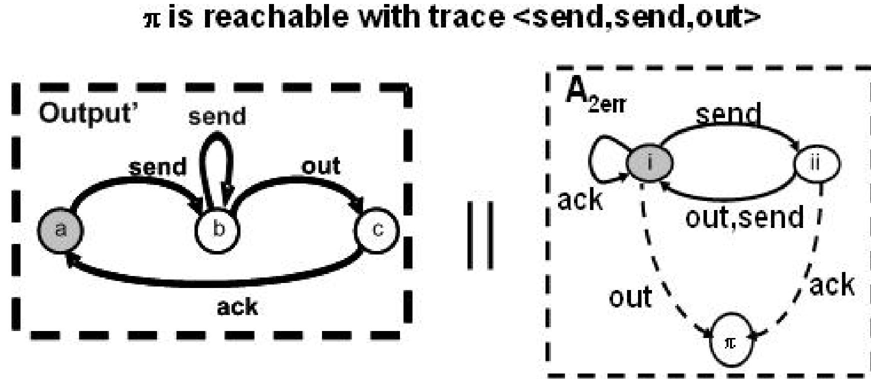


Figure 5.13: Computing the composition system *Output'*$\|A_{2_{err}}$.

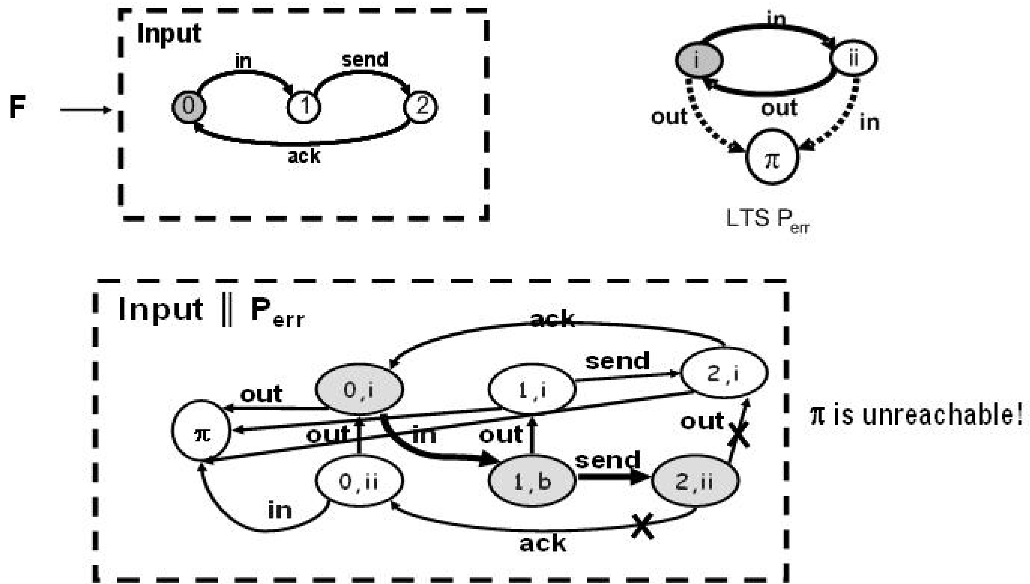


Figure 5.14: Simulating the $cex\uparrow\Sigma$ on the composition system $Input\|p_{err}$.

membership queries to the Teacher. After re-updating, the observation table (S,E,T) is closed. A candidate DFA M_3 is constructed from this closed observation table. We get a safety LTS A_3 from the candidate DFA M_3 by removing the non-accepting state and all its ingoing transitions. The safety LTS A_3 illustrated in Figure 5.15.

The Teacher uses the safety LTS A_3 as candidate assumption for the compositional rule. The Teacher applies two steps of the compositional rule and counterexample analysis to answer conjectures from L* Learner.

Step 1 first is applied, the Teacher checks the formula $\langle A_3\rangle$ *Input* $\langle p\rangle$ by computing the composition system $A_3\|Input\|p_{err}$ illustrated in Figure 5.16. It is easy to check that the error state π is reachable in this composition system, so the the Teacher then returns false and a counterexample cex=⟨in,send,out,ack,out⟩. The Teacher informs L* Learner that its conjecture A_3 is not correct and provides $cex\uparrow\Sigma$=⟨send,out,ack,out⟩ to witness this fact.

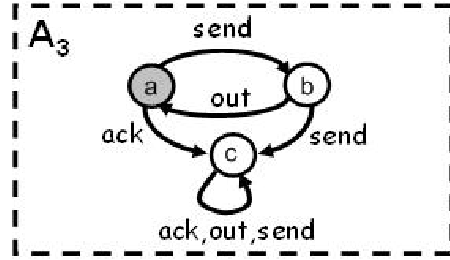


Figure 5.15: The safety LTS A_3 is created from the closed table (S,E,T).

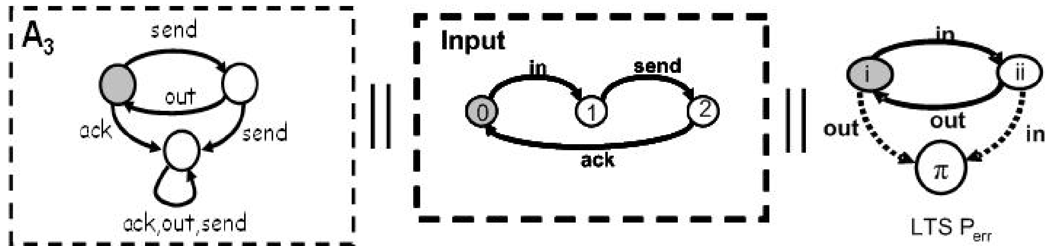


Figure 5.16: Computing the composition system $A_3\|Input\|p_{err}$.

The counterexample $cex\uparrow\Sigma$=⟨send,out,ack,out⟩ is analyzed by L* to find a suffix e of cex that witnesses a difference between $L(M_3)$ and U. In this case, L* analyzes and sets e to *out*. Adding *out* into E and re-updating the observation table (S,E,T) by making membership queries to the Teacher. After re-updating, the observation table (S,E,T) is closed. A candidate DFA M_4 is constructed from this closed observation table. We get a safety LTS A_4 from the candidate DFA M_4 by removing the non-accepting state and all its ingoing transitions. The safety LTS A_4 illustrated in Figure 5.17.

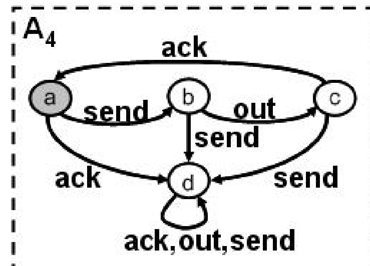


Figure 5.17: The safety LTS A_4 is created from the closed table (S,E,T).

The Teacher then uses the safety LTS A_4 as candidate assumption for the compositional rule. The Teacher applies two steps of the compositional rule and counterexample analysis to answer conjectures from L* Learner.

At Step 1, the Teacher checks the formula $\langle A_4\rangle$ *Input* $\langle p\rangle$ by computing the composition system $A_4\|Input\|p_{err}$ illustrated in Figure 5.18. It is easy to check that the error state π is unreachable in this composition system, so the the Teacher then returns true. It means that the formula $\langle A_4\rangle$ *Input* $\langle p\rangle$ holds. The Teacher forwards A_4 to Step 2.

Step 2 is applied by checking the formula ⟨true⟩ *Output'* $\langle A_4\rangle$. In order to check this formula, the Teacher computing the composition *Output'*$\|A_{4_{err}}$ illustrated in Figure 5.19.

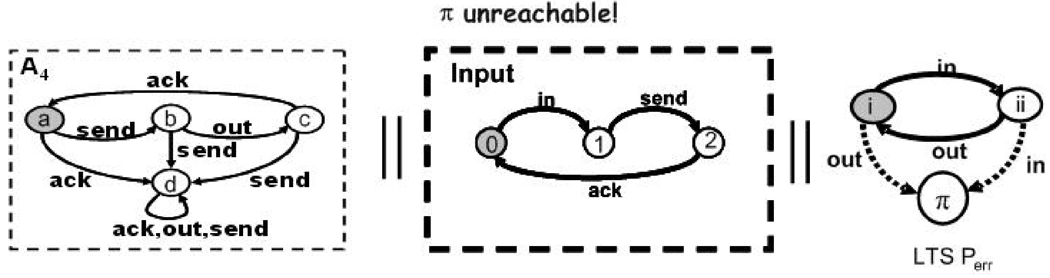


Figure 5.18: Computing the composition system $A_4\|Input\|p_{err}$.

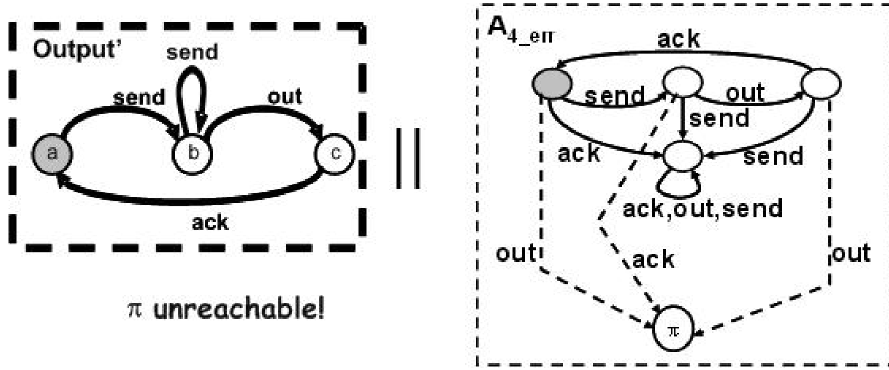


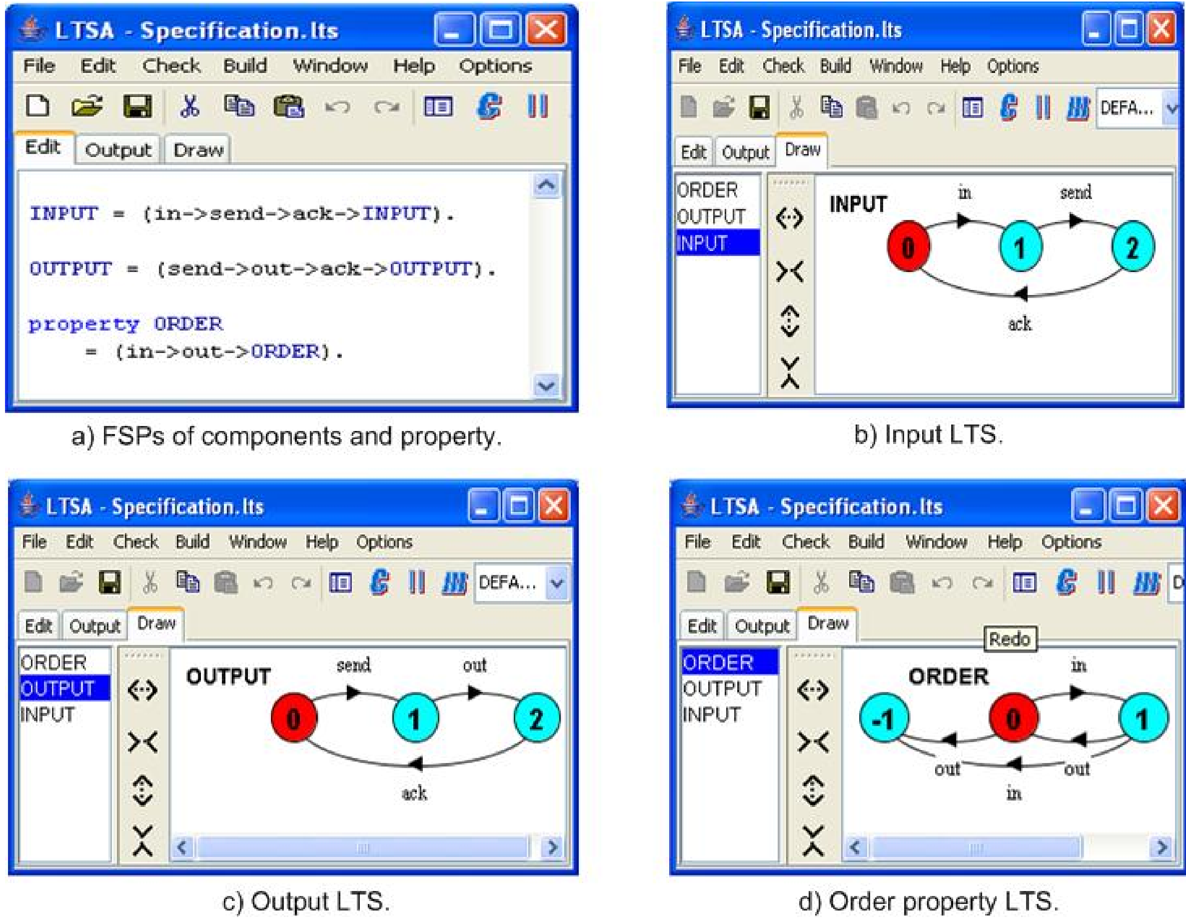
Figure 5.19: Computing the composition system $Output'\|A_{4_{err}}$.

It is easy to check that the error state π is unreachable in this composition system, so the the Teacher then returns true. It means that the order property p holds in the composition system $Input\|Output'$ (i.e., $Input\|Output' \models p$). The learning algorithms L* terminates and returns the new assumption $A_{new}(p) = A_4$. The new system is verified that it satisfies the property p.

By using the old assumption A_2 as the initial assumption to re-generate new assumption $A_{new}(p)$, the proposed approach doesn't generate candidate assumptions A_1, A_2 passing many steps. Therefore, this approach can re-generate new assumption in the faster manner and reduce the computing time to generate these candidate assumptions. In practice, the effect of this approach will be more efficient by verifying the complex systems. It is very significant to verify the component-based systems in the context of component refinement.

5.4 Experiments

This thesis uses the LTSA tool [12] to check correctness of the proposed approach by concrete examples illustrated in section 5.2 & 5.3. Because currently well-known model checker do not support assumption model checking and assumption generation, at each iteration i in frameworks for assumption generation and new assumption re-generation, we use assumption A_i produced by L* Learner and check that whether it satisfies the compositional rule (i.e., $\langle A_i \rangle$ F $\langle p \rangle$ and $\langle \text{true} \rangle$ C_i $\langle A_i \rangle$ both hold) by checking composition systems $A_i\|F\|p_{err}$ and $C_i\|A_{i_{err}}$ in the LTSA tool, where i=1,2.



a) FSPs of components and property.

b) Input LTS.

c) Output LTS.

d) Order property LTS.

Figure 5.20: FSPs and LTSs of illustration system in LTSA tool.

Firstly, LTSs of all components and property of the illustration system (illustrated in Figure 5.1) are translated into the input language "*Finite State Processes (FSPs)*" of this tool showed in Figure 5.20. In this figure, sub-figure a) presents FSPs of components and the order property p and the others present correlative LTSs of them. In the LTSA tool, state -1 is the error state.

For generating the old assumption $A(p)$ between the framework *Input* and the component *Output*, at iteration *1*, L* Learner produced the candidate assumption A_1 illustrated in Figure 5.7. We use LTSA tool to check that A_1 satisfies the compositional rule.

At Step 1, the formula $\langle A_1 \rangle$ *Input* $\langle p \rangle$ is checked by computing the composition system $A_1 \| Input \| p_{err}$. FSPs of these LTSs and checking result is presented in Figure 5.21. In this figure, sub-figure a) presents FSPs of component *Input*, assumption A_1, the order property p, and the composition system. The checking result is presented in sub-figure b). It means that the composition system is violated.

At iteration *2*, L* Learner produced the candidate assumption A_2 illustrated in Figure 5.11. We use LTSA tool to check that whether A_2 satisfies the compositional rule.

a) FSPs of Input, A1 and P

b) Checking result

Figure 5.21: The checking result of composition system $A_1 \| Input \| p_{err}$.

a) FSPs of Input, A2 and P

b) Checking result

Figure 5.22: The checking result of composition system $A_2 \| Input \| p_{err}$.

Step 1 is applied first to check the formula $\langle A_2 \rangle$ *Input* $\langle p \rangle$ by computing the composition system $A_2 \| Input \| p_{err}$. FSPs of these LTSs and checking result is presented in Figure 5.22. In this figure, sub-figure a) presents FSPs of the component *Input*, assumption A_2, the order property p, and the composition system. The checking result is presented in sub-figure b). It means that the formula holds.

Step 2 is applied by checking formula $\langle \text{true} \rangle$ *Output* $\langle A_2 \rangle$. The LTSA tool computes the composition system $Output \| A_{2_{err}}$ illustrated in Figure 5.23. Sub-figure a) presents FSPs of the component *Output*, assumption A_2, and the composition system, where A_2 is order property. The checking result is presented in sub-figure b). It means that the formula also holds. From these, A_2 satisfies the compositional rule. L* terminates and returns assumption between *Input* and *Output* as $A(p)$=A_2.

When the component *Output* is *refined* into the new component *Output'* illustrated

a) FSPs of Output and property A2

b) Checking result

Figure 5.23: The checking result of composition system $Output \| A_{2_{err}}$.

a) FSPs of components and property

b) Checking result

Figure 5.24: The checking result of composition system $Output' \| A_{2_{err}}$.

in Figure 5.2, the LTSA tool checks the formula $\langle \text{true} \rangle$ *Output'* $\langle A_2 \rangle$ by computing the composition system $Output' \| A_{2_{err}}$ presented in Figure 5.24. Sub-figure a) showed FSPs of the new component *Output'*, assumption A_2, and the composition system. The checking result is presented in sub-figure b). The result means that the formula does not hold. A new assumption between *Input* and *Output'* is re-generated.

At iteration *1* of the algorithm for new assumption regeneration, L* Learner produced the candidate assumption A_3 illustrated in Figure 5.15. The LTSA tool is used to check that whether A_3 satisfies the compositional rule.

Step 1 is applied first to check the formula $\langle A_3 \rangle$ *Input* $\langle p \rangle$ by computing the composition system $A_3 \| Input \| p_{err}$ showed in Figure 5.25. In this figure, sub-figure a) presents FSPs of the component Input, assumption A_3, the order property p, and the composition system. The checking result is presented in sub-figure b). It means that the formula does not hold.

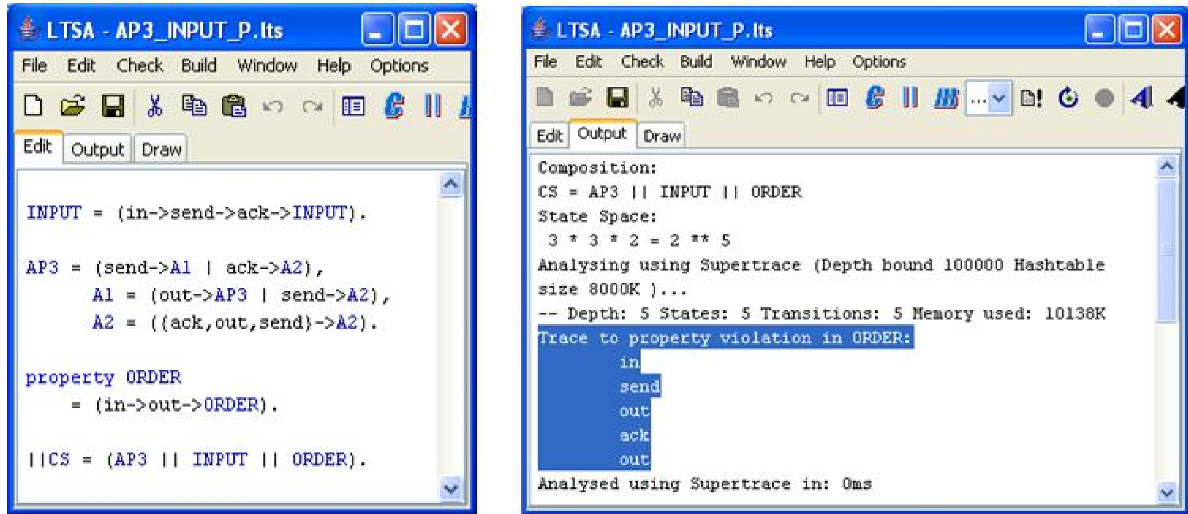


Figure 5.25: The checking result of composition system $A_3\|Input\|p_{err}$.

At iteration *2* of the algorithm for new assumption regeneration, L* Learner produced the candidate assumption A_4 illustrated in Figure 5.17. We use LTSA tool to check that whether A_4 satisfies the compositional rule.

At Step 1, the formula $\langle A_4 \rangle$ *Input* $\langle p \rangle$ is checked by computing the composition system $A_4\|Input\|p_{err}$. FSPs of these LTSs and checking result is presented in Figure 5.26. In this figure, sub-figure a) presents FSPs of the component *Input*, assumption A_4, the order property p, and the composition system. The checking result is presented in sub-figure b). It means that the formula holds.

Step 2 is applied by checking formula $\langle \text{true} \rangle$ *Output'* $\langle A_4 \rangle$. The LTSA tool computes the composition system $Output'\|A_{4_{err}}$ illustrated in Figure 5.27. Sub-figure a) presents FSPs of the component *Output'*, assumption A_4, and the composition system, where A_4 is order property. The checking result is presented in sub-figure b). It means that the formula also holds. From these, A_4 satisfies the compositional rule. The algorithm for new assumption regeneration terminates and returns new assumption between *Input* and *Output'* as $A_{new}(p)$=A_4.

LTSA - AP4_INPUT_P.lts

```
INPUT = (in->send->ack->INPUT).

AP4 = (send->A1 | ack->A2),
      A1 = (out->A3 | send->A2),
      A2 = ({ack,out,send}->A2),
      A3 = (ack->AP4 | send->A2).

property ORDER
    = (in->out->ORDER).

||CS = (AP4 || INPUT || ORDER).
```

a) FSPs of components and property

```
Composition:
CS = AP4 || INPUT || ORDER
State Space:
 4 * 3 * 2 = 2 ** 5
Analysing using Supertrace (Depth
bound 100000 Hashtable size 8000K
)...
-- Depth: 0 States: 4 Transitions: 4
Memory used: 10001K
No deadlocks/errors
Analysed using Supertrace in: 10ms
```

b) Checking result

Figure 5.26: The checking result of composition system $A_4 \| Input \| p_{err}$.

LTSA - N_OUTPUT_AP4.lts

```
N_OUTPUT = (send->OUT),
OUT = (send->OUT| out->ack->N_OUTPUT)

property AP4 = (send->A1 | ack->A2),
      A1 = (out->A3 | send->A2),
      A2 = ({ack,out,send}->A2),
      A3 = (ack->AP4 | send->A2).

||CS = (N_OUTPUT || AP4).
```

a) FSPs of components and property

```
Composition:
CS = N_OUTPUT || AP4
State Space:
 3 * 4 = 2 ** 4
Analysing using Supertrace (Depth
bound 100000 Hashtable size 8000K
)...
-- Depth: 0 States: 6 Transitions: 8
Memory used: 10054K
No deadlocks/errors
Analysed using Supertrace in: 0ms
```

b) Checking result

Figure 5.27: The checking result of composition system $Output' \| A_{4_{err}}$.

Chapter 6

Conclusion and Future Works

Verification of software has received a lot of attentions of the software engineering community, specially modular verification of component-based software. Currently there are many approaches was proposed [10, 11, 22, 2, 4, 7, 8] and some successful applications were developed [2, 4, 7, 8]. However, there are many limited problems and many open problems which are under research from these approaches.

This thesis proposed a faster assume-guarantee verification approach to verify component-based software in the context of component refinement. In this technique, if a component is *refined* into a new component, the whole system of many existing components and the new component is not required to re-check altogether. It only checks the new component satisfying the assumption of the old system. If yes then the new system is verified. Otherwise, the proposed approach performs some analysis to determine whether the property is indeed violated in the new system or whether the assumption of the old system is too strong for the new component to satisfy. If the assumption is too strong, the new assumption is re-generated. The approach in this thesis tries to reuse the results of the previous verification in order to have an incremental manner to re-generate new assumption. It does not re-generate new assumption from beginning. After re-generating the new assumption, we can consider it as the results of previous verification for future changes. A case study is presented to illustrate my approach step by step. The LTSA [12] tool also is used to check correctness of the technique by some concrete examples.

However, this thesis does not evaluate the effectiveness of the proposed approach in formal way. My future works will complete this limitation. Moreover, in future works will focus on studying following problems:

- The *refinement* concept in this thesis means adding some states and transitions into the old component. In practice, it not only adding some behaviors but also removing some behaviors. Finding an approach for new assumption generation in faster manner with the new concept of refinement is an interesting problem in my future works.

- Applying my approach for some bigger illustration systems.

- Because currently model checkers do not support assume-guarantee verification, in my future works, i will implement a tool supporting about that.

- The problem in this thesis will be more complex if the component C_1 contains of many sub-components. What are the interaction between these sub-components for the component C_1 to satisfy the assumption? The algorithms to generate assumption will also be more complex.

- In the proposed approach, the property P is a LTS global safety property. The problem will be more difficult if the property P is represented by the powerful temporal logic CTL.

- The problem in this thesis has a strong relationship between C_1 and C_2 (i.e., C_2 is a refinement of C_1). If C_1 and C_2 are independent together, the problems will be difficult to find a faster approach to verify the new system.

Bibliography

[1] D. Angluin: "Learning regular sets from queries and counterexamples", Information and Computation, 75(2):87106, Nov. 1987.

[2] C. Blundell, D. Giannakopoulou, C. Pasareanu: "Assume-Guarantee Testing", Microsoft Research - Specification and Verification of Component-Based Systems (SAVCBS 2005) Workshop: 7-14

[3] L. Brim, B. Zimmerova, I. Cerna, P.Varekova: "Component-Interaction Automata as a Verification-Oriented Component-Based System Specification", Microsoft Research - Specification and Verification of Component-Based Systems (SAVCBS 2005) Workshop: 31-38.

[4] J. Cobleigh, D. Giannakopoulou, C. Pasareanu: "Learning Assumptions for Compositional Verification", TACAS 2003: 331-346.

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

[6] K. Fisler and S. Krishnamurthi: "Modular verification of collaboration-based software designs", In Proc. Symposium on the Foundations of Software Engineering, September 2001.

[7] D. Giannakopoulou, C. Pasareanu, H. Barringer: "Assumption Generation for Software Component Verification", ASE 2002: 3-12.

[8] D. Giannakopoulou, C. Pasareanu, J. Cobleigh: "Assume-Guarantee Verification of Source Code with Design-Level Assumptions", ICSE 2004: 211-220.

[9] Robert M. Keller: "Formal verification of parallel programs", Communications of the ACM, 19(7):371384, July 1976.

[10] O. Kupferman and M. Y. Vardi: "Modular model checking", In Compositionality: The Significant Difference, volume 1536 of Lecture Notes in Computer Science. Springer-Verlag, 1998.

[11] K. Laster, O. Grumberg: "Modular model checking of software", Conference on Tools and Algorithms for the Constructions and Analysis of Systems, 1998.

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

[13] A. Nerode: "Linear automaton transformations", In In Proc. of the American Mathematical Society 9, pages 541544, 1958.

[14] T.T. Nguyen, T. Katayama: "A Framework for Unanticipated Software Changes", Proc. Unanticipated Software Evolution (USE), European Joint Conferences on Theory and Practice of Software (ETAPS' 2003).

[15] T.T. Nguyen, T. Katayama: "Towards a Sound Modular Model Checking of Collaboration-Based Software Designs", IEEE Computer Asia-Pacific Software Engineering Conference APSEC, pp. 88-97 (2003).

[16] T.T. Nguyen, T. Katayama: "Handling Consistency of Software Evolution in an Efficient Way", IEEE Computer Proc. International Workshop on Principles of Software Evolution (IWPSE), pp. 121-130, IEEE RE'2004.

[17] T.T. Nguyen, T. Katayama: "Open Incremental Model Checking", Microsoft Research - Specification and Verification of Component-Based Systems (SAVCBS) Workshop, ACM FSE'04.

[18] T.T. Nguyen, T. Katayama: "A Formal Approach Facilitating the Evolution of Component-Based Software", IEEE Computer Proc. International Workshop on Principles of Software Evolution (IWPSE), ACM SIGSOFT ESEC/FSE'2005.

[19] T.T. Nguyen, T. Katayama: "Constructing Open Systems via Consistent Components", International Colloquium on Theoretical Aspects of Computing (ICTAC), Springer-Verlag LNCS 2005.

[20] T.T. Nguyen, T. Katayama: "Specification and Verification of Inter-Component Constraints in CTL", Microsoft Research - Specification and Verification of Component-Based Systems (SAVCBS) Workshop, ACM SIGSOFT ESEC/FSE'05.

[21] C. B. Jones: "Tentative steps toward a development method for interfering programs", ACM Trans, on Prog. Lang. and Sys., 5(4):596-619, Oct. 1983.

[22] C. S. Pasareanu, M. B. Dwyer, and M. Huth: "Assume-guarantee model checking of software: A comparative case study", In Theoretical and Practical Aspects of SPIN Model Checking, volume 1680 of Lecture Notes of Computer Science, Springer-Verlag, 1999.

[23] A. Pnueli: "In transition from global to modular temporal reasoning about programs", In Logics and models of concurrent systems, pages 123-144, 1985.

[24] R. L. Rivest and R. E. Schapire: "Inference of finite automata using homing sequences", Information and Computation, 103(2):299347, Apr. 1993.