การทวนสอบความสอดคล้องของเครื่องจักรสถานะจำกัดโดยการอนุมานเครื่องและการตรวจสอบโมเดล

นายวราวุฒิ ผ้าเจริญ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2555

CONFORMANCE VERIFICATION OF FINITE STATE MACHINES BASED ON MACHINE INFERENCE AND MODEL CHECKING

Mr.Warawoot Pacharoen

A Dissertation Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2012

Thesis Title	CONFORMANCE VERIFICATION OF FINITE STATE MACHINES BASED ON MACHINE INFERENCE AND MODEL CHECKING
By	Mr.Warawoot Pacharoen
Field of Study	Computer Engineering
Thesis Advisor	Assistant Professor Athasit Surarerks, Ph.D.
Thesis Co-advisor	Assistant Professor Pattarasinee Bhattarakosol, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree

..... Dean of the Faculty of Engineering
(Associate Professor Boonsom Lerdhirunwong, Dr.Ing.)

THESIS COMMITTEE

..... Chairman
(Professor Prabhas Chongstitvatana, Ph.D.)

..... Thesis Advisor
(Assistant Professor Athasit Surarerks, Ph.D.)

..... Thesis Co-advisor
(Assistant Professor Pattarasinee Bhattarakosol, Ph.D.)

..... Examiner
(Associate Professor Wiwat Vatanawood, Ph.D.)

..... External Examiner
(Assistant Professor Panjai Tantasanawong, Ph.D.)

..... External Examiner
(Assistant Professor Arnon Rungsawang, Ph.D.)

วราวุฒิ ผ้าเจริญ: การทวนสอบความสอดคล้องของเครื่องจักรสถานะจำกัดโดยการอนุมานเครื่องและการตรวจสอบโมเดล. (CONFORMANCE VERIFICATION OF FINITE STATE MACHINES BASED ON MACHINE INFERENCE AND MODEL CHECKING) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : ผศ. ดร. อรรถสิทธิ์ สุรฤกษ์, อ.ที่ปรึกษาวิทยานิพนธ์ร่วม : ผศ. ดร. ภัทรสินี ภัทรโกศล, 69 หน้า.

ปัญหาความสอดคล้องได้รับความสนใจอย่างมากในงานวิจัยทางด้านการทวนสอบซอฟต์แวร์ เนื่องจากซอฟต์แวร์ที่ไม่สามารถทำงานได้ตรงตามที่ระบุไว้ในข้อกำหนดซอฟต์แวร์อาจทำให้เกิดปัญหาต่างๆ เช่น การหยุดชะงักในระหว่างการสื่อสารกับซอฟต์แวร์อื่น แต่สมมติฐานเบื้องต้นในงานวิจัยด้านการทวนสอบความสอดคล้องในปัจจุบัน คือ เราสามารถมองเห็นโครงสร้างภายในของซอฟต์แวร์ที่จะนำมาทวนสอบได้ ข้อจำกัดดังกล่าวทำให้เราไม่สามารถทวนสอบซอฟต์แวร์บางประเภท เช่น ซอฟต์แวร์ที่พัฒนาด้วยภาษา .NET หรือ Java ได้ เนื่องจากไม่สามารถมองเห็นการทำงานภายในได้ ในวิทยานิพนธ์นี้ผู้วิจัยได้เสนอวิธีการใหม่ในการทวนสอบความสอดคล้องของข้อกำหนดซอฟต์แวร์กับซอฟต์แวร์ที่สามารถมองเห็นได้แต่พฤติกรรมภายนอก โดยผู้วิจัยใช้ขั้นตอนวิธีในการอนุมานเครื่องเพื่อให้ได้เครื่องจักรสถานะจำกัดจากซอฟต์แวร์ที่ไม่สามารถมองเห็นโครงสร้างภายในได้ หลังจากได้เครื่องจักรสถานะจำกัดของซอฟต์แวร์จากขั้นตอนวิธีดังกล่าวแล้วจึงนำไปแปลงเป็นภาษารูปนัย LTS เพื่อใช้การตรวจสอบโมเดล LTSA ตรวจสอบความสอดคล้องของโมเดลที่ได้และโมเดลของข้อกำหนดซอฟต์แวร์ต่อไป จากผลการทดลองกับการประกอบเว็บเซอร์วิสผู้วิจัยสามารถตรวจพบการทำงานของเว็บเซอร์วิสที่ไม่ตรงตามที่ระบุไว้ในข้อกำหนดซอฟต์แวร์ได้ และเนื่องจากขั้นตอนวิธีที่นำมาใช้สามารถอนุมานเครื่องจักรสถานะจำกัดเชิงกำหนดได้เพียงอย่างเดียว ผู้วิจัยจึงได้เสนอขั้นตอนวิธี L_{NM^*} ที่สามารถอนุมานได้ทั้งเครื่องจักรสถานะจำกัดเชิงกำหนดและเชิงไม่กำหนด

ภาควิชา ..วิศวกรรมคอมพิวเตอร์.. ลายมือชื่อนิสิต .....

สาขาวิชา ..วิศวกรรมคอมพิวเตอร์.. ลายมือชื่ออ.ที่ปรึกษาวิทยานิพนธ์หลัก .....

ปีการศึกษา .....2555..... ลายมือชื่ออ.ที่ปรึกษาวิทยานิพนธ์ร่วม .....

5071822621: MAJOR COMPUTER ENGINEERING
KEYWORDS: FINITE STATE MACHINE / CONFORMANCE VERIFICATION / MACHINE INFERENCE / MODEL CHECKING

WARAWOOT PACHAROEN : CONFORMANCE VERIFICATION OF FINITE STATE MACHINES BASED ON MACHINE INFERENCE AND MODEL CHECKING. ADVISOR : ASST. PROF. ATHASIT SURARERKS, Ph.D., CO-ADVISOR : ASST. PROF. PATTARASINEE BHATTARAKOSOL, Ph.D., 69 pp.

The conformance problem has attracted much interest in the research field of software verification, since the implementation that doesn't conform to the specification may cause some errors such as the communicating interruption in the composited application. However, the earlier works in conformance verification assume that the internal structures of the implementation are explicit. It may not always be the case such as the applications that implemented in .NET or Java. In this dissertation, we propose an alternative approach for verifying a conformance between the specification and the implementation whose only external behaviors can be observed. We use an adapted version of Angluin's algorithm to infer a deterministic Finite State Machine (FSM) from the implementations. By transforming the obtained model to the modeling formalism LTS, the model checker LTSA can be used for checking the conformance criterion in our framework. From the experiment based on Web services composition, we can detect the execution trace of the implemented Web service that does not conform to the choreography specification. Moreover, since the assumption that the implementations have to be deterministic may be too restricted in some applications (such as, a communication system or a component-based system), we also present a novel learning algorithm, namely $L_{NM}{}^{*}$, which can be applied to infer both deterministic and non-deterministic FSMs.

Department : ... Computer Engineering ...	Student's Signature .....
Field of Study : .. Computer Engineering ..	Advisor's Signature .....
Academic Year : 2012	Co-advisor's Signature .....

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Asst. Prof. Dr. Athasit Surarerks, for his guidance, and support. He has advised me for many years since my Master thesis, and then my Ph.D. dissertation. Also, I am grateful to Asst. Prof. Dr. Pattarasinee Bhattarakosol, my co-supervisor, who has given me invaluable advice in both academic and personal life since my undergraduate, my master, and my doctorate. I am truly proud that I was under their supervision.

I greatly appreciate Prof. Dr. Prabhas Chongstitvatana, Assoc. Prof. Dr. Wiwat Vatanawood, Asst. Prof. Dr. Arnon Rungsawang, and Asst. Prof. Dr. Panjai Tantatsanawong, for being my dissertation committees and giving several useful comments and suggestions to improve this dissertation.

I am warmly thankful to Assoc. Prof. Dr. Toshiaki Aoki, who has provided a great opportunity for one year research at Japan Advanced Institute of Science and Technology (JAIST), Japan. Without his inspiration and encouragement, I would be lost. I also thank Thai's friends at JAIST, who have given me a memorable time which we spent together.

I appreciate the University Development Commission (UDC) Scholarship from Department of Mathematics, Statistics and Computer, Faculty of Science, Ubon Rajathanee University that supports the tuition fee and some living expenses during my master and Ph.D. program.

I thank all ELITE laboratory's members for the great time we lived together within the same research room.

I would like to express my deeply gratitude to my parents for their endless support, and I sincerely thank my dearest Miss Apaporn Champa for her unconditional understanding. It is because of them I have attained this position. I dedicate this dissertation to them.

Contents

Chapter Page

List of Tables

List of Figures

CHAPTER I

INTRODUCTION

To promote a system's reliability, there are many attempts in the software verification domain to determine whether an implementation *conforms* to its specification which can be a program specification, a communication protocol, etc. The reason is that unexpected behaviors of an implementation may cause the system errors or the communicating interruption in the composited application. This motivates the study of *conformance verification* that will be investigated in this dissertation.

Typically, the considered implementations are software components, services, or modules that assume to be modeled as *finite state machines* which are widely used to model systems in diverse areas. Given a formal model which acts as specification, a number of studies have been presented to ensure the correct implementations. The proposed methodologies can be categorized into two main categories based on the knowledge of the internal structures of the implementation: *conformance verification* and *conformance testing*.

The works in conformance verification approach assume that the implementation is white-box, i.e., its internal structures can be directly observed and mapped to a formal language. Then the formal model of the implementation can be used to check the conformance criteria against the given specification. To the best of our knowledge, notations and semantics of the formal languages which are used in these works are more or less based on Petri net, process algebra, or automata. Moreover, different notions of the conformance relation between the specification and the implementation have been formally defined with respect to their formalism. However, the assumption that the implementations are white-box does not need to be the case, since some applications can be only observed their external behavior, e.g., third party applications or the applications implemented by Java or .NET.

For checking the conformance of a black-box implementation, the technique so-called *model-based testing* has been used for a long time in both academic and industrial section. In this approach, it is assumed that a *correct* formal model which acts as specification is given, typically as *finite state machine (FSM)* or *labelled transition system (LTS)*.

Then we want to generate a set of tests (each test is a pair of input sequence and expected output sequence), or *test suite*, from the given model in order to compare with the real system. By applying each test, if the actual output sequence differs from the expected output sequence, then a fault has been detected. Otherwise, it can be concluded that an implementation conforms to the specification under some assumptions, e.g., the number of states of the implementation is smaller than a given bound m. Interesting source of techniques and tools of model-based testing can be found in the book by Broy et al. (2005). Further detailed of conformance testing can be found in (Lee and Yannakakis, 1996; Fujiwara et al., 1991; Hierons, 2004) for FSM-based specification and (Tretmans, 2008; Frantzen et al., 2009) for LTS-based specification.

Learning in the context of model checking has been used for several purposes in the software verification literature. For example, Chaki et al. (2008) used the approach for checking component substitutability of evolving software systems. The approach was also used by Cobleigh et al. (2003) to automatically generate assumptions for assume-guarantee verification of systems. Black box checking (Peled et al., 1999) and adaptive model checking (Groce et al., 2002) are other example applications of this approach in order to verify correctness of the system in the case of specification is not available or incomplete, respectively. The most well-known learning algorithm in this field was proposed by Angluin (1987), namely L^*, for deterministic finite-state automata (DFA) inference. Moreover, recently Shahbaz and Groz (2009) proposed an algorithm ${L_M}^*$ that adopt from Angluin's work for Mealy machine inference.

In this dissertation, an automata learning technique is introduced in order to solve the conformance verification problem of black box implementations. Since the practical limitation of current works in conformance verification is that the internal structures of implementations have to be explicit, our approach can be used without this limitation by observing behavior of the implementation through its interface. Moreover, unlike testing our approach is applicable regardless of preset test environment, such as characterizing set, state cover set, transition cover set, etc. Given a black box implementation, our method synthesizes a behavioral model by asking queries to test (observe) its internal behavior. As a result, the tests used here are generated online while constructing the model.

1.1 Our Method for a Conformance Verification

Suppose we are given a Labelled Transition System (LTS) specification S_i^{Spec} of an implementation i in the protocol. First we use a learning algorithm to infer an implementation model of the implementation as a Finite State Machine (FSM). Then, in order to check the conformance criterion, we transform the FSM to the corresponding LTS S_i^{Imp} and use model checking to perform the conformance analysis based on a relation between the LTSs of specification model and the learning model, i.e., S_i^{Spec} and S_i^{Imp}.

For the notion of conformance used in this dissertation, we assume that the implementation conforms to the specification if it implements all and only the observable behaviors allowed by the specification. The reason is if the implementation can perform the actions which are not foreseen by the given specification or cannot perform the actions which are desired by the given specification, this may cause the interruption in the communication. This conformance criterion is called a *trace equivalence* relation of LTS and will be formally defined in the next chapter.

1.2 Objectives of Research

- To propose an alternative approach for conformance verification that can be applied with black-box implementations.
- To propose a novel active learning algorithm that can be extended to non-deterministic finite state machines inference.
- To propose the use of a model checking to answer the equivalence query of the learning algorithm using compositional reachability analysis of LTSs.

1.3 Scope and Assumption

The scope of this dissertation is limited to the following:

- This dissertation considers the stateful implementations.
- This dissertation focuses only functional properties of the system. Other aspects, such as, timing, performance, security, etc. are not considered.
- The proposed method does not rely on specific implementation languages.

Additionally, in this dissertation, we assume the following:

- The protocol specification is given in terms of *Finite State Process (FSP)* notations that are the textual representations of LTS.
- The input/output interfaces of the implementations are known and observable.
- The implementations can be modeled as deterministic FSMs, i.e., they exhibit regular and deterministic behaviors. Later this assumption will be relaxed to non-deterministic case in Chapter 5.
- An abstraction technique is applied for variables in the specification model and the learned model.

1.4 Summary of Contributions

This dissertation provides an alternative approach for conformance verification of a black-box implementation for which we can only observe its external behavior. The proposed technique is applicable regardless of a preset test suite; therefore, it can endure with changes in the specification and in the application. The learning algorithm requires only information of i/o interface of the implementation. This property is suitable for the practical application such as one that developed by third party.

Furthermore, we also propose the novel learning algorithm that can infer behavioral model of the implementation as a non-deterministic finite state machine (NFSM). This relaxes the assumption that the implementation have to deterministic, since it may be too restricted in some applications such as a concurrent communication system.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows. The next chapter recalls the basic definitions and notions which are used globally in the dissertation. Chapter 3 overviews the background work and surveys the state-of-the-art in the domain of conformance verification, conformance testing, and automata learning. Chapter 4 describes our proposed method for conformance verification based on the Web services case study. Chapter 5 presents the extended work towards inferring NFSMs. Finally, a discussion and conclusions of the dissertation are presented in Chapter 6.

CHAPTER II

DEFINITIONS AND NOTATIONS

This chapter briefly recalls the basic definitions and notations that will be used later in the forthcoming chapters. The definitions and most of the notations follow (Broy et al., 2005; Lee and Yannakakis, 1996; Fujiwara et al., 1991; Hierons, 2004).

2.1 Labelled Transition System

Let L be the universal set of observable actions (or labels), and let $Act = L \cup \{\tau\}$, where τ denotes an internal action that is *unobservable* by the environment. Moreover, we use π to denote a special *error state*, which models the state that has no outgoing transitions.

Definition 2.1 (Labelled Transition System) *A Labelled Transition System (LTS) P is a 4-tuple $\langle S, \Sigma, \Delta, s_0 \rangle$ where S is the finite non-empty set of states including the error state π, $\Sigma \subseteq L$ is the finite non-empty set of observable actions called the alphabet of P, $\Delta \subseteq S - \{\pi\} \times \Sigma \cup \{\tau\} \times S$ is the transition relation that maps from a state and an action onto another state, and $s_0 \in S$ is the initial state.*

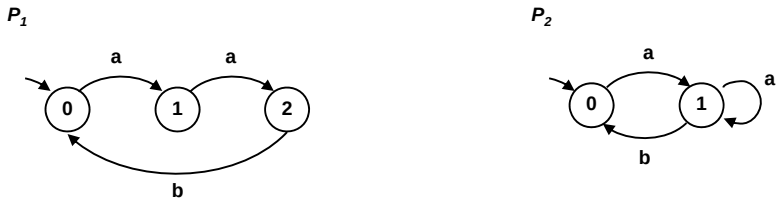

Figure 2.1: Example of LTS $P_1 = \langle S^1, \Sigma^1, \Delta^1, s_0^1 \rangle$ (left) and LTS $P_2 = \langle S^2, \Sigma^2, \Delta^2, s_0^2 \rangle$ (right) with $\Sigma^1 = \Sigma^2 = \{a, b\}$.

To note that, an LTS P is *non-deterministic* if it contains τ-transitions or if there exist $(s, a, s'), (s, a, s'') \in \Delta$ but $s' \neq s''$. Otherwise, P is *deterministic*.

Definition 2.2 (Transit) *An LTS $P = \langle S, \Sigma, \Delta, s_0 \rangle$ transits into an LTS P' with action $a \in Act$, denoted as $P \xrightarrow{a} P'$, if:*

(a) $P' = \langle S, \Sigma, \Delta, s_0' \rangle$, where $s_0' \neq \pi$ and $(s_0, a, s_0') \in \Delta$, or

(b) $P' = \Pi$, *where* $s'_0 = \pi$

We use Π to denote the LTS that is allowed to have the error state π as its initial state (i.e., LTS $\langle\{\pi\}, \Sigma, \emptyset, \pi\rangle$).

Definition 2.3 (Parallel Composition) *Given two LTSs* $P_1 = \langle S^1, \Sigma^1, \Delta^1, s_0^1\rangle$ *and* $P_2 = \langle S^2, \Sigma^2, \Delta^2, s_0^2\rangle$, *the parallel composition of* P_1 *and* P_2, *donoted by* $P_1 \parallel P_2$, *is defined as follows:*

(a) If $P_1 = \Pi$ *or* $P_2 = \Pi$, *then* $P_1 \parallel P_2 = \Pi$.

(b) Otherwise, $P_1 \parallel P_2$ *is an LTS* $P = \langle S, \Sigma, \Delta, s_0\rangle$, *where* $S = S^1 \times S^2$, $\Sigma = \Sigma^1 \cup \Sigma^2$, $s_0 = (s_0^1, s_0^2)$, *and* Δ *is the smallest relation satisfying the following rules, where* a *is either observable action or* τ, *i.e.,* $a \in \Sigma \cup \{\tau\}$*:*

$$\frac{P_1 \xrightarrow{a} P_1', a \notin \Sigma^2}{P_1 \parallel P_2 \xrightarrow{a} P_1' \parallel P_2}$$

$$\frac{P_2 \xrightarrow{a} P_2', a \notin \Sigma^1}{P_1 \parallel P_2 \xrightarrow{a} P_1 \parallel P_2'}$$

$$\frac{P_1 \xrightarrow{a} P_1', P_2 \xrightarrow{a} P_2', a \neq \tau}{P_1 \parallel P_2 \xrightarrow{a} P_1' \parallel P_2'}$$

According that the parallel composition is both commutative and associative, the order in which LTSs are composed is not significant, e.g., $P_1 \parallel P_2$ is equivalent to $P_2 \parallel P_1$.

Definition 2.4 (Trace and Language) *A trace* σ *of an LTS* P *is a sequence of observable actions that* P *can perform, starting from the initial state. We use* $Tr(P)$, *called the language of* P, *to mean the set of all traces of* P.

Definition 2.5 (Trace Inclusion) *For any two LTSs* $P_1 = \langle S^1, \Sigma^1, \Delta^1, s_0^1\rangle$ *and* $P_2 = \langle S^2, \Sigma^2, \Delta^2, s_0^2\rangle$ *where* $\Sigma^1 \subseteq \Sigma^2$, *we write* $P_1 \leq_{tr} P_2$ *to denote the trace inclusion relation that means all traces of* P_1 *are also in* P_2. *Formally,* $P_1 \leq_{tr} P_2$ *if and only if* $Tr(P_1) \subseteq Tr(P_2)$.

In practice, when checking $P_1 \leq_{tr} P_2$, firstly an *error* LTS of P_2 denoted $P_{2_{err}}$ is created. It can be created by adding the error state π and traps possible violations with this state. For instance, Figure 2.2 shows the error LTSs of both LTSs in Figure 2.1.

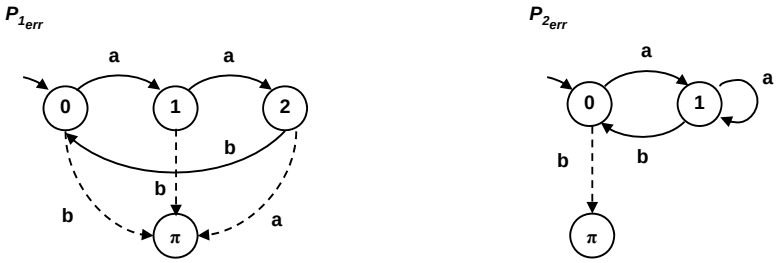

Figure 2.2: Error LTSs of the LTSs in Figure 2.1.

Note that the error LTS is *complete*, meaning each state other than the error state has outgoing transitions for every action in the alphabet. Formally, the error LTS of LTS $P = \langle S, \Sigma, \Delta, s_0 \rangle$ is $P_{err} = \langle S \cup \{\pi\}, \Sigma, \Delta', s_0 \rangle$, where $\Delta' = \Delta \cup \{(s, a, \pi) | a \in \Sigma \text{ and } \nexists s' \in S : (s, a, s') \in \Delta\}$.

After that, the parallel composition $P_1 \parallel P_{2_{err}}$ is computed. If the π state is reachable in $P_1 \parallel P_{2_{err}}$, then $P_1 \not\leq_{tr} P_2$; that means there exists at least one trace σ that occurs in P_1 but does not occur in P_2, i.e., $\exists \sigma, \sigma \in Tr(P_1) \wedge \sigma \notin Tr(P_2)$.

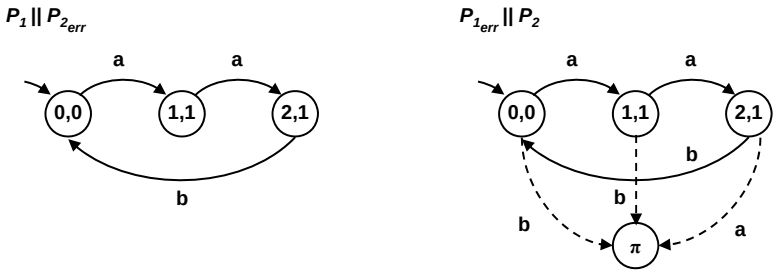

Figure 2.3: The parallel composition between $P_1 \parallel P_{2_{err}}$ (left) and $P_{1_{err}} \parallel P_2$ (right).

For example, in Figure 2.3 (left), since the π state cannot be reached in $P_1 \parallel P_{2_{err}}$, then that means $P_1 \leq_{tr} P_2$. However, since the π state is reachable in $P_{1_{err}} \parallel P_2$ as shown in Figure 2.3 (right), then that means $P_2 \not\leq_{tr} P_1$ (e.g., a $\cdot b \in Tr(P_2) \wedge a \cdot b \notin Tr(P_1)$).

The trace inclusion relation is a preorder and can be extended to define an equivalence:

Definition 2.6 (Trace Equivalence) *The trace equivalence relation between two LTSs*

P_1 and P_2, written $P_1 =_{tr} P_2$, holds iff $P_1 \leq_{tr} P_2$ and $P_2 \leq_{tr} P_1$.

In other words, two equivalent LTSs have the same observable behaviors, formally $Tr(S_1) = Tr(S_2)$.

2.2 Finite State Machine

From this point forward, the term "finite state machines" (FSMs) will be referred to as "Mealy machines", which represent outputs on their transitions. Moreover, both terms are used interchangeably in this dissertation.

Definition 2.7 (Finite State Machine) *A finite state machine (FSM) M is a 5-tuple $\langle Q, I, O, \delta, q_0 \rangle$, where Q, I, O are the non-empty finite sets of states, input symbols, and output symbols, respectively; $q_0 \in Q$ is the initial state; and $\delta : Q \times I \to 2^{Q \times O} \setminus \{\emptyset\}$ is the transition function, where $2^{Q \times O}$ is the power set of $Q \times O$ and $\emptyset$ is the empty set.*

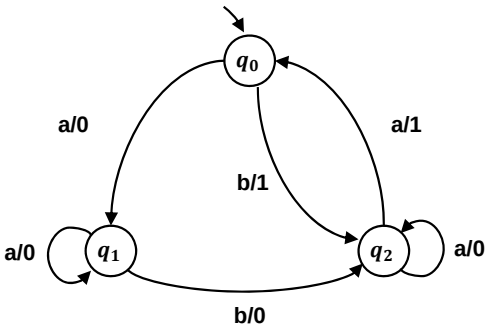

Figure 2.4: Example of an finite state machine.

At any point in time, the machine is currently at state $q \in Q$. It is possible to give inputs to the machine. By receiving an input $i \in I$, the machine may produce an output $o \in O$ and change to the next state $q' \in Q$ if and only if $(q', o) \in \delta(q, i)$. For example, Figure 2.4 shows an FSM with state $Q = \{q_0, q_1, q_2\}$, input symbols $I = \{a, b\}$, and output symbols $O = \{0, 1\}$. Applying a starting in state q_0 produces output 0 and moves to the next state q_1, since $(q_1, 0) \in \delta(q_0, a)$.

As usual, the function δ can be extended to take an input sequence, i.e., $\delta : Q \times I^* \to 2^{Q \times O^*}$. For example, here $\delta(q_0, a \cdot b) = \{(q_2, 0 \cdot 0)\}$.

Recall from (Hierons, 2004) that the projections of the function δ, called δ_Q and δ_O, are possible to define such that $\delta_Q : Q \times I^* \to 2^Q$ and $\delta_O : Q \times I^* \to 2^{O^*}$ will give *the states reached* and *the output sequences produced*, respectively, from a state and a given input sequence. For example, in Figure 2.4, $\delta_Q(q_2, a) = \{q_0, q_2\}$ and $\delta_O(q_2, a) = \{0, 1\}$.

Suppose that $\bar{x}$ denotes an *input sequence* $x_1 \cdot x_2 \cdot \ldots \cdot x_k$ of input symbols from I, and that $\bar{y}$ denotes an *output sequence* $y_1 \cdot y_2 \cdot \ldots \cdot y_k$ of output symbols from O. An *input/ output sequence* is a sequence $\bar{x}/\bar{y} = x_1/y_1 \cdot x_2/y_2 \cdot \ldots \cdot x_k/y_k$ for some $x_1, \ldots, x_k \in I$ and $y_1, \ldots, y_k \in O$. Next, the following properties are usually referred to in the model:

Property 1 (Deterministic). An FSM is *deterministic* if, for all states $q \in Q$ and all input $i \in I, |\delta(q, i)| \leq 1$. Otherwise, an FSM is *non-deterministic*.

Property 2 (Initially Connected). An FSM is *initially connected* if every state $q \in Q$ can be reached from the initial state q_0, i.e., $\forall q \in Q, \exists \bar{x} \in I^*$ such that $q \in \delta_Q(q_0, \bar{x})$.

Property 3 (Completely Specified). An FSM is *completely specified* if, for all of the states, it has transitions for every input. Formally, $\forall q \in Q, \forall i \in I, |\delta(q, i)| \geq 1$.

However, if the machine is not completely specified, called a *partially specified FSM*, it can be transformed to a completely specified FSM by adding either a *sink state* s_Ω or *loop back transition*, with a designated *error symbol* Ω for all inputs that do not occur in the original machine.

Property 4 (Observable). An FSM is *observable* if, for every state $q \in Q$, input $i \in I$, and output $o \in O$, it has at most one transition leaving q with input i and output o, that is, $|\{q' \in Q \mid (q', o) \in \delta(q, i)\}| \leq 1$.

This property ensures that, with the same input, the machine will never move to different states with the same output. This scenario aids us in determining the target state of the machine by observing only its output.

Property 5 (Reduced). An FSM is *reduced* (or *minimized*) if it is initially connected and no two states are equivalent. In other words, there always exists an input sequence that can distinguish between any two states, i.e., $\forall q, q' \in Q$ and $\exists \bar{x} \in I^*$, $\delta_O(q, \bar{x}) \neq \delta_O(q', \bar{x})$.

Definition 2.8 (Language) *Given an FSM* $M = (Q, I, O, \delta, q_0)$*, an associated language of* M *from a state* $q \in Q$*, denoted by* $\mathcal{L}_M(q)$*, is the set of input/output sequences allowed by* M *from* q*. More formally,* $\mathcal{L}_M(q) = \{\bar{x}/\bar{y} \mid \bar{x} \in I^* \wedge \bar{y} \in \delta_O(q, \bar{x})\}$*. We use* $\mathcal{L}(M)$*, called the language of* M*, to mean the set* $\mathcal{L}_M(q_0)$*.*

Definition 2.9 (Reduction) *Given two FSMs* $M_1 = (Q^1, I^1, O^1, \delta^1, q_0^1)$ *and* $M_2 = (Q^2, I^2, O^2, \delta^2, q_0^2)$*, where* $I^1 = I^2$*, FSM* M_1 *is a reduction of FSM* M_2*, denoted by* $M_1 \preceq M_2$*, if and only if* $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$*.*

Definition 2.10 (Equivalence) *The equivalence relation between the two FSMs* M_1 *and* M_2*, written* $M_1 = M_2$*, holds if and only if* $M_1 \preceq M_2$ *and* $M_2 \preceq M_1$*, i.e.,* $\mathcal{L}(M_1) = \mathcal{L}(M_2)$*.*

2.3 General Notations

Some of general notations used in the dissertation are summarized in the Table 2.1 and Table 2.2 as follows.

Table 2.1: Notations for labelled transition system.

Notation	Meaning
τ	an internal action
π	a special error state
Π	an LTS$\langle\{\pi\}, \Sigma, \emptyset, \pi\rangle$
$P \xrightarrow{a} P'$	transit (See Definition 2.2)
$P_1 \parallel P_2$	parallel composition (See Definition 2.3)
P_{err}	the error LTS of LTS P (See Definition 2.5)
$Tr(P)$	a language of LTS P (See Definition 2.4)
$P_1 \leq_{tr} P_2$	the trace inclusion relation (See Definition 2.5)
$P_1 =_{tr} P_2$	the trace equivalence relation (See Definition 2.6)

Table 2.2: Notations for finite state machine.

Notation	Meaning
Ω	an error output symbol
s_Ω	a sink state
δ_Q	the projection of the function δ to give the states reached
δ_O	the projection of the function δ to give the output sequences produced
$\bar{x}/\bar{y}$	input/output sequence
$\mathcal{L}(M)$	a language of FSM M (See Definition 2.8)
$M_1 \preceq M_2$	the reduction relation (See Definition 2.9)
$M_1 = M_2$	the equivalence relation (See Definition 2.10)

CHAPTER III

BACKGROUND AND RELATED WORK

This chapter overviews the background work and surveys the state-of-the-art in the domain of Web service choreography, conformance verification, conformance testing, and automata learning techniques.

3.1 Web Services Choreography Preliminaries

Web Services Choreography is a specification protocol defining the order of the observable message exchanges among the participating services in a business process. Starting from this global description, each party can then extract out the local descriptions for building own Web services independently. There are several languages have been proposed to specify a choreography such as WS-CDL, Let's Dance (Zaha et al., 2006), and MAP (Barker et al., 2009). A basic building block of these choreography languages is an activity, either basic or structured one. The basic activity corresponds to an atomic action such as: a *request* action, a *response* action, a *request-response* action, a *variable assignment* action. On the other hand, the structured activity defines the control flow among activities that includes: a *sequence* activity, a *non-deterministic* activity, a *choice* activity, a *parallel* activity. However, it is not within the scope of this work to provide a further detailed formalization of the languages.

For simplicity, this dissertation uses the Message Sequence Chart (MSC) to graphically represented a choreography between Web services as shown in Figure 3.1. More precisely we assume that formal specification of a Web service in choreography is given in terms of *Finite State Process (FSP)* notations that are the textual representations of LTS.

Figure 3.1 illustrates a Request For Quotation (RFQ) example of choreography between three participating services, namely *buyer*, *seller*, and *shipper*. They can interact by sending messages to each other and the choreography specifies the sequence of the interactions.

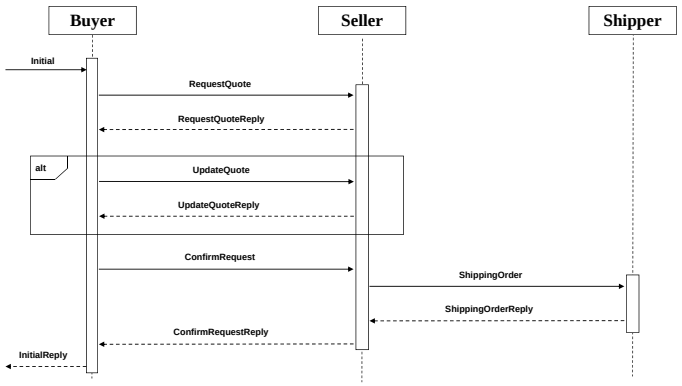

Figure 3.1: An RFQ example of Web service choreography specified by Message Sequence Chart.

To note that, FSP specifications that are used as formal specification in our work will be translated to the corresponding finite LTSs in order to analyze and verify some properties. For example, from Figure 3.1, we can extract an LTS model of seller Web service as shown in Figure 3.2.

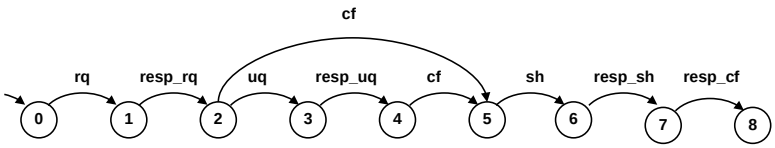

Figure 3.2: LTS S_{Seller}^{Spec} of seller Web service.

3.2 Conformance Verification

Formal methods are introduced to the Web service design and specification in order to promote the system reliability by analyzing and verifying some required properties. Given formal models of choreography specification and a set of implementations, there are many efforts in the literature for checking a conformance between both models with respect to some conformance criteria. For example, Busi et al. (2005) propose formal languages for describing choreography and orchestration based on process algebra where a notion of conformance takes the form of bisimulation-like relation. In (Kazhamiakin and Pistore, 2006) the authors present a formal framework for conformance verification that allows for modeling the data-flow and control-flow of web service composition. The conformance criterion is the behaviors of the implemented composition and the choreography are the same. The work in (Yeung, 2006) formalizes WS-CDL and BPEL using process algebra CSP, where conformance can be carried out based on the concept of traces-refinement in CSP. Foster et al. (2006) translate WS-CDL specification and BPEL4WS,

the early version of BPEL, to the FSP process algebra model, then a conformance relation is defined by equivalent of the interaction traces. Furthermore, the conformance in this work can be checked using the model checker LTSA.

However, the aforementioned works verify a conformance based on the behavior of composition of local implementations against the choreography specification. Obviously, using model checking to verify in such a case may face with state explosion problem. There are several works apply the projection techniques in order to extract the behavior of the considered role in a choreography, hence the conformance checking procedure can be done locally without considering other services. Among the works on this technique, Zhao et al. (2006) introduce a formal model for WS-CDL, and a simple projection from choreography to orchestration is given. Li et al. (2007) propose two formal languages for describing choreography and orchestration. In order to do the verification, the definition of endpoint projection and process refinement are presented in this work. Moreover, the work in (Tasharofi and Sirjani, 2009) uses Reo and Constraint Automata with State Memory (CASM) as a unified formalism for describing both WS-CDL and BPEL. By using endpoint projection on CASM, the behavior of an interested party in the choreography is obtained and can be used to compare with the CASM model of BPEL of the interested party based on the *simulation relation* in CASM.

3.3 Conformance Testing

The problem of conformance testing has been investigated by many works in the literature. Given a formal model which acts as specification and a black box implementation, called *implementation under test (IUT)*, for which we can only observe its external behavior, we want to check whether the IUT conforms to the given specification. Therefore, in this approach it is assumed that a specification model of the system is given, typically in the form of FSM or LTS. Based on these models, there are many attempts proposing techniques and algorithms to construct a test suite in order to test the IUT. In this work, we classify the existing works of model-based testing into two categories, FSM-based and LTS-based testing, with respect to the specification model.

For FSM-based testing, various test sequence generation techniques have been published to either a deterministic finite state machine (DFSM) or a non-deterministic finite state machine (NFSM) specification. The well-known methods based on DFSM specifica-

tion are Transition tour (TT) method, W-method, Wp-method, Distinguishing sequence (DS) method, Unique input/output (UIO) method, and UIOv-Method. These methods generate the test suites which have different length and fault detection coverage. Further details and comparisons of the DFSM-based techniques can be found in the Chapter 4 of (Broy et al., 2005). On the other hand, most approaches for selecting a test suite from NFSM are based on the method so-called *state counting*, which can be applied to the case in which the IUT is known to be deterministic (Hierons, 2004), or non-deterministic. Last but not least, the necessary assumptions of all referred FSM-based test generation methods are (*i*) the specification FSM is assumed to be reduced or minimal, initially connected, completely specified and observable, and (*ii*) there is an upper bound on the number of states in the IUT.

For LTS-based testing, there are a large number of methods have been proposed in the literature. Among these works, the most of well-known methods are based on the construction of a *canonical tester* from, e.g., LOTOS specification, refusal graphs, or the Compulsory and Options sets (CO-OP method). However, some drawbacks of these methods from the practical used have been reported, such as the LTS models do not distinguish between inputs and outputs. To the best of our knowledge, the well-accepted testing relation for LTS (with inputs and outputs) is ioco (Tretmans, 2008).

According to the recent surveys of testing Web services composition (Canfora and Penta, 2009), several approaches exist to deal with conformance problem of the Web service implementations to the specification models (e.g., a finite state machine, a graph grammar). For example, Bertolino and Polini (2005) propose an approach to test that a black-box Web service conforms to a specification before the service will be added into an UDDI registry. The behavior of the service in this work is described by a finite state machine. In (Heckel and Mariani, 2005) the authors use graph grammars to represent the mutually agreed behavior between two partners, called *contract*, in order to enable the automatic derivation of the test suite.

The closest idea to our approach is (Frantzen et al., 2009) in which the authors propose the verification framework, called JAMBITION, to test whether an implemented Web service conforms to a design specification. This framework uses a Symbolic Transition System (STS) diagram as a specification model and tests the Java Web services, and then

the test cases will be generated on-the-fly, i.e., the next input will be guided by the observed output from the IUT. Furthermore, test generation method can be based on full state and/or transition coverage criteria with respect to sioco testing relation (i.e., a sound and complete adaptation of ioco for STS). Unfortunately, test cases generated based on state or transition coverage cannot detect some faults in the IUT.

3.4 Automata Learning Preliminaries

In this section we start by reviewing the automata learning approach in general, and then we focus on the learning algorithms that usually used in software engineering domain and the algorithm used in our approach.

The field of automata learning has been studied for decades as one branch of *grammatical inference*. The early research in grammatical inference is to solve the problems in three main areas: computational linguistics, inductive inference, and pattern recognition. Later, it has been introduced to other domains such as bio-informatics and automata learning. The good source of techniques and applications of grammatical inference can be found in the book of de la Higuera (2010).

Given an unknown language U over the alphabet Σ, the goal of automata learning is to infer an automaton that can recognize the language U. In the following subsections, we describe the concepts of the two general categories of automata learning techniques: *passive learning* and *active learning*.

3.4.1 Passive Learning Approaches

In this approach, we are given a finite set of the observations (or informants) that are the strings from the alphabet Σ of an unknown language U. Then the algorithms in this paradigm try to estimate a model of the language from these observations. The challenge of this concept is that the learning process is bounded to identify the automaton in the limit of samples. If the strings are the words of the language, they are called *positive samples*. Otherwise, they are called *negative samples*. From a set of positive and negative samples, Gold (1978) proved that finding a minimum machine, more precisely a minimum deterministic finite automaton (DFA), of the targeted language from the given set of both samples is hard (i.e., NP-Complete). However, the research was continued to find *probably*

approximately correct (PAC) model that could learn concepts with a high probability but in low complexity.

3.4.2 Active Learning Approaches

Instead of using given samples, the algorithms in active learning paradigm have an ability to collect observations by asking queries. It is typically assumed that there exists an Oracle who knows the language and can correctly answer some queries in this setting.

The most well-known algorithm in active learning approach is provided by Angluin, namely L^*, which can efficiently learn an unknown regular language U and produce a minimum DFA that accepts U in polynomial time. In the setting of L^*, the learning algorithm is called a *Learner* and an Oracle is called a *minimally adequate Teacher* (or *Teacher* for short).

There are two types of queries in L^*. The first type is a *membership query*, consisting of a string $\sigma \in \Sigma^*$. The response is "0" (in case $\sigma \notin U$) or "1" (in case $\sigma \in U$) that will be recorded in an *observation table*. The Learner asks the membership queries and recorded the responses from the Teacher until some conditions on the observation table have been met. These conditions will be described further in the next subsection together with the case of Mealy machine inference. Then, the Learner constructs a conjecture that is a candidate DFA M whose language the algorithm L^* believes to be identical to U. The second type of question is called an *equivalence query* that consists of the conjecture. The Teacher's answer is true if $\mathcal{L}(M) = U$. Otherwise the Teacher returns a counterexample showing a discrepancy between the machine $\mathcal{L}(M)$ and U.

3.4.3 Learning Algorithms for Mealy Machine Inference

The learning algorithm that plays an important role in our approach was proposed by Shahbaz and Groz (2009), namely ${L_M}^*$. It adopts the Angluin's algorithm L^* (Angluin, 1987) to Mealy machine inference. Even though L^* can learn an unknown regular language in polynomial time, its adaptations to Mealy Machines are not obviously efficient as discussed in (Shahbaz and Groz, 2009). For example, the direct adaptations can be performed through model transformation techniques by mapping from inputs I and outputs O of the Mealy Machine to letters of a DFA's alphabet Σ, such that $\Sigma = I \cup O$

(Hungar et al., 2003) or $\Sigma = I \times O$ (Mäkinen and Systä, 2001). However, these methods are confronted by complexity problems because the cost of L^* is polynomial based on the size of Σ. Shahbaz and Groz (2009) observed that, by slightly modifying the structure of the observation table and the way in which the counterexample is processed, their proposed method, namely ${L_M}^+$, can learn deterministic Mealy machines more effectively.

Similar to L^*, in order to infer the Mealy machine from a black box machine, the algorithm ${L_M}^*$ needs to ask two types of questions to a Teacher that is assumed to correctly answer the questions. Let M_U be a black box machine whose input set I is known. The first type is an *output query*, consisting of a string $\sigma \in I^+$. This is similar to the concept of membership queries in L^*. The different is that instead of "0" or "1", the Teacher replies with the complete output strings which will be recorded in an observation table.

The second type of question is an equivalence query, consisting of a candidate Mealy machine M whose language the algorithm ${L_M}^*$ believes to be identical to M_U. The answer is true if $\mathcal{L}(M) = \mathcal{L}(M_U)$. Otherwise the Teacher returns a counterexample which is a string σ showing a discrepancy between the machine M and M_U. The detailed description of an observation table and the procedure of ${L_M}^*$ will be defined as follows.

3.4.4 Observation Table

The structure of an observation table, denoted by (S, E, T), of the algorithm ${L_M}^*$ consists of three parts: S, E, and T where

- S, E are the non-empty finite sets of prefix-closed and suffix-closed, respectively, input strings from I^+.
- T is a finite function that map $(S \cup S \cdot I) \times E$ to the output string from O^+.

Intuitively, an observation table can be visualized as a two-dimensional array with rows labelled by elements of $(S \cup S \cdot I)$ and columns labelled by elements of E. The entry corresponding to row s in $(S \cup S \cdot I)$ and column e in E equals to $T(s \cdot e)$ containing the last output symbol of the output string as $suf^{|1|}(\lambda(q_0, s \cdot e))$, where $suf^{|k|}(\omega)$ denotes the suffix of a string ω of length k.

Definition 3.1 (Row Equivalence) *Let $s, t \in S \cup S \cdot I$ be two rows in the table (S, E, T), then s and t are row equivalence, denoted by $s \cong_E t$, if and only if $T(s, e) = T(t, e)$, for all $e \in E$. Moreover, we used $[s]$ to denote the equivalence class of rows that are row equivalence to s.*

Definition 3.2 (Closed and Consistent Observation Table) *An observation table is called closed if and only if for each $t \in S \cdot I$, there existed an $s \in S$ such that $s \cong_E t$. An observation table is called consistent if and only if for each $s_1, s_2 \in S$ such that $s_1 \cong_E s_2$, then $s_1 \cdot i \cong_E s_2 \cdot i$, for all $i \in I$.*

If the table (S, E, T) is closed and consistent, we can construct a Mealy machine conjecture as defined in (Shahbaz and Groz, 2009).

3.4.5 The Algorithm $L_M{}^*$

As shown in (Shahbaz and Groz, 2009), the Algorithm $L_M{}^*$ starts by initializing the observation table (S, E, T) with $S = \{\epsilon\}$ and $E = I$. Then $L_M{}^*$ asks the output queries constructed from the table to determine T. For each row $s \in S \cup S \cdot I$ and column $e \in E$, a query is constructed as $s \cdot e$. The corresponding output string from the query $s \cdot e$ is recorded to the entry (s, e), i.e., row s and column e, of the table with the help of function T where $T(s, e) = suf^{|1|}(\lambda(q_0, s \cdot e))$.

After filling the initial table with the result of the queries, $L_M{}^*$ starts the loop for testing whether the current table is closed and consistent. If it is not closed that means $\exists t \in S \cdot I$ such that $s \ncong_E t$, for all $s \in S$. Then $L_M{}^*$ finds and moves t to S and $T(t \cdot i, e)$ is determined for all $i \in I, e \in E$ in $S \cdot I$. If the table is not consistent that means $\exists s_1, s_2 \in S, i \in I$, and $e \in E$ such that $s_1 \cong_E s_2$, but $T(s_1 \cdot i, e) \neq T(s_2 \cdot i, e)$. Then $L_M{}^*$ finds and adds the string $i \cdot e$ to E and extends the table by asking the output queries for the missing elements.

These two operations are performed repeatedly until the table is closed and consistent. From the closed and consistent table (S, E, T), $L_M{}^*$ makes a Mealy machine conjecture M and asks it to the Teacher. The Teacher replies either *"yes"*, showing that the conjecture is correct, or with a counterexample σ. If the Teacher replies *yes*, $L_M{}^*$ terminates with the correct Mealy machine M. Otherwise, the Teacher replies with a

counterexample σ, then ${L_M}^*$ adds σ and all its prefixes to S and extends (S, E, T) by the output queries. Then the algorithm repeats the main loop until (S, E, T) is closed and consistent, followed by making a new conjecture.

CHAPTER IV

CONFORMANCE VERIFICATION FOR WEB SERVICE COMPOSITION

In this chapter, we provide the detailed explanation of our research methodology based on the Web services composition.

4.1 The Approach

In a sense, Web service can be considered as a reactive system that can be invoked by the environment, i.e., service consumer. Typically, Web service receives a request message from the environment, performs some computations or invokes other Web services, takes decisions on its internal transitions, and finally sends the corresponding response message back to the environment. For this reason, the behaviors of Web service can be naturally modeled by a Mealy machine that is specifically designed for an I/O based system.

As mentioned in Chapter 1, in order to analyze a conformance between the specification and the implementation, we verify whether the implementation has the same observable behaviors as described in the specification, i.e., the conformance criterion is the trace equivalence between the specification model and the implementation model, as described by Definition 2.6.

To obtain the model of the implementation, our framework uses the learning algorithm $L_M{}^*$ in an iterative fashion as shown in Figure 4.1. After each iteration, the algorithm provides a Mealy machine conjecture of the implementation Web service that can be used as a source for performing model checking. However, without any assumptions the learning from the implementation is impossible.

- We assume that the Web service is available and accessible, e.g., no network connection problem, no message loss.
- The inferred Web service is input deterministic. This means the Web service will produce the same output sequence from each input sequence that is sent to it.

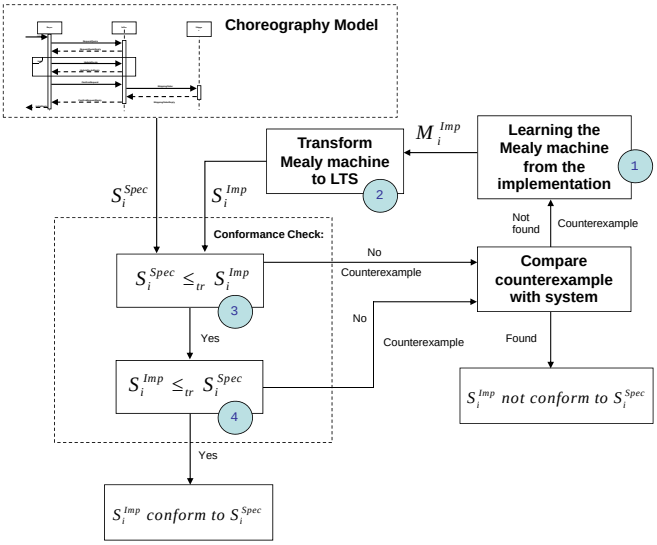

Figure 4.1: Web service conformance verification framework.

Suppose we have a specification LTS S_i^{Spec} of Web service i that is assumed to be correctly specified. Our framework consists of the following four steps:

4.1.1 Step 1: Learning the Implementation model

The first step is to learn the behavioral model from the actual implementation of Web service using $L_M{}^*$. Table 4.1 is an example of an observation table that is inferred from the seller Web service in RFQ example; moreover, a Mealy machine conjecture M_{Seller}^{Imp} constructed from the table is shown in Figure 4.2.

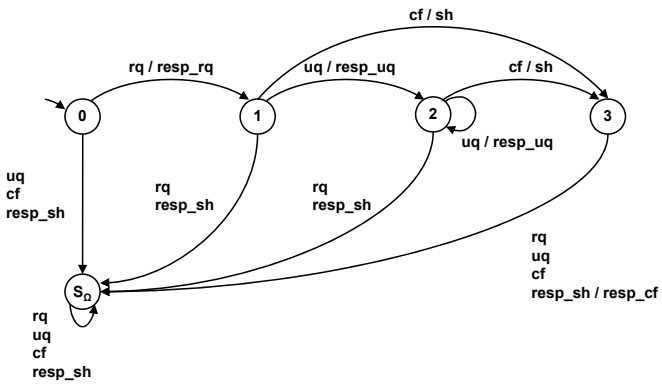

Figure 4.2: Mealy machine conjecture of seller Web service from Table 4.1.

Note that, the error outputs are recorded as Ω in the table. Furthermore, for simplicity, we do not show Ω on the transition of Mealy machine, such as uq is used

Table 4.1: Closed and Consistent Observation Table T_1 for learning *seller* service.

T_1		E			
		rq	uq	cf	resp_sh
S	ϵ	resp_rq	Ω	Ω	Ω
	rq	Ω	resp_uq	Ω	Ω
	uq	Ω	Ω	Ω	Ω
	rq, uq	Ω	resp_uq	sh	Ω
	rq, uq, cf	Ω	Ω	Ω	resp_cf
$S \cdot I$	cf	Ω	Ω	Ω	Ω
	resp_sh	Ω	Ω	Ω	Ω
	rq, rq	Ω	Ω	Ω	Ω
	rq, cf	Ω	Ω	Ω	Ω
	rq, resp_sh	Ω	Ω	Ω	Ω
	uq, rq	Ω	Ω	Ω	Ω
	uq, uq	Ω	Ω	Ω	Ω
	uq, cf	Ω	Ω	Ω	Ω
	uq, resp_sh	Ω	Ω	Ω	Ω
	rq, uq, rq	Ω	Ω	Ω	Ω
	rq, uq, uq	Ω	resp_uq	sh	Ω
	rq, uq, resp_sh	Ω	Ω	Ω	Ω
	rq, uq, cf, rq	Ω	Ω	Ω	Ω
	rq, uq, cf, uq	Ω	Ω	Ω	Ω
	rq, uq, cf, cf	Ω	Ω	Ω	Ω
	rq, uq, cf, resp_sh	Ω	Ω	Ω	Ω

instead of uq/Ω.

4.1.2 Step 2: Transforming Mealy Machine to LTS

An intention of this step is to transform the Mealy machine conjecture, from the previous step, to the model checking formalism LTS. The LTS for a given Mealy machine model in the trace semantics, called the corresponding LTS, is defined as follows:

Definition 4.1 (Corresponding LTS) *Given a Mealy machine* $M = \langle Q, I, O, \delta, q_0 \rangle$ *where* Q, I, O *are the non-empty finite sets of states, input symbols, and output symbols respectively,* $\delta : Q \times I \to 2^{Q \times O} \setminus \{\emptyset\}$ *is the transition function,* $q_0 \in Q$ *is the initial state. The corresponding LTS with regard to* M *is an LTS* $P = \langle S, \Sigma, \Delta, s_0 \rangle$*, where*

- $s_0 = q_0$
- $\Sigma = I \cup O$
- *For all* $q_i \in Q$ *and all* $a \in I$*, there exists a one-to-one mapping* $\psi : Q \to S_1$ *and* $\gamma : Q \times I \to S_2$ *where* $S = S_1 \cup S_2$ *and* $S_1 \cap S_2 = \emptyset$ *such that*
  - $(\psi(q_i), a, \psi(q_j)) \in \Delta$ *if and only if* $(q_j, \Omega) \in \delta(q_i, a)$
  - $(\psi(q_i), a, \gamma(q_i, a)), (\gamma(q_i, a), b, \psi(q_j)) \in \Delta$ *if and only if* $(q_j, b) \in \delta(q_i, a)$ *where* $b \in O$ *and* $b \neq \Omega$

– $\psi(s_\Omega) = \pi$, *where* s_Ω *is the sink state, i.e., the state that has no outgoing transition to other states*

For example, Figure 4.3 shows the corresponding LTS $S^{Imp}_{Seller_{err}}$ of the Mealy machine in Figure 4.2. To note that, the corresponding LTS is not only an error LTS, but also a deterministic LTS.

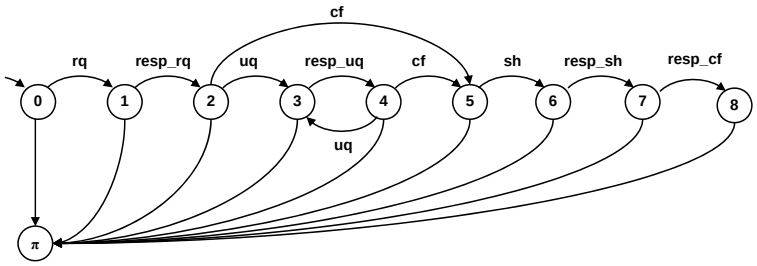

Figure 4.3: The corresponding LTS of the Mealy machine in Figure 4.2.

4.1.3 Step 3: Verifying $S_i^{Spec} \leq_{tr} S_i^{Imp}$

The two following steps intend to check whether the implementation model of Web service conforms to the choreography specification. In this step our framework verifies that all observable traces from the specification have been implemented, i.e., $S_i^{Spec} \leq_{tr} S_i^{Imp}$. As mention in Definition 2.5, in order to check $S_i^{Spec} \leq_{tr} S_i^{Imp}$, we check whether the π state is reachable in $S_i^{Spec} \parallel S^{Imp}_{i_{err}}$.

For instance, using the LTS S^{Spec}_{Seller} (Figure 3.2) as specification model and LTS $S^{Imp}_{Seller_{err}}$ (Figure 4.3) as an error LTS of implementation model, we verify that whether an error state π is reachable in $S^{Spec}_{Seller} \parallel S^{Imp}_{Seller_{err}}$.

4.1.4 Step 4: Verifying $S_i^{Imp} \leq_{tr} S_i^{Spec}$

In this phase we verify that $S_i^{Imp} \leq_{tr} S_i^{Spec}$ to guarantee that the actual system has implemented only the traces foreseen by the protocol. To do this, we create the error LTS of the specification model, i.e., $S^{Spec}_{i_{err}}$, which traps all possible traces that do not occur from the specification with the error state.

For example, Figure 4.4 shows the error LTS $S^{Spec}_{Seller_{err}}$ of seller Web service specification. In addition, we transform the error LTS $S^{Imp}_{i_{err}}$ of the implementation to the LTS

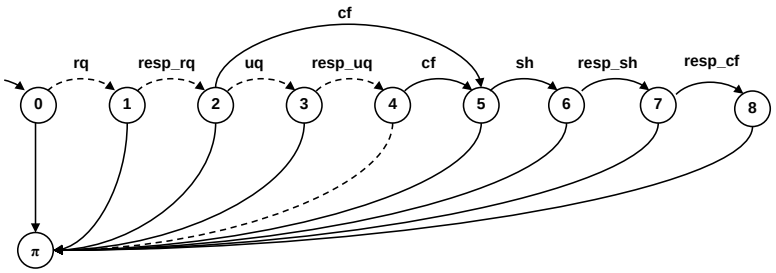

Figure 4.4: Error LTS of S^{Spec}_{Seller} in Figure 3.2.

S^{Imp}_i. This can be done easily by removing the error state and all transitions that lead to it. Then we check whether the π state is reachable in $S^{Spec}_{i_{err}} \parallel S^{Imp}_i$.

Finally, the verification process is terminated and returns that the implementation model conforms to choreography specification, if a counterexample is not found from both step 3 and step 4. Otherwise, we have a counterexample from either step 3 or step 4 which shows the different behavior between the implementation model and specification. Then we check whether the counterexample is spurious or not with the real implementation. If the counterexample trace is found in an actual execution, we can conclude that the implementation does not conform to the specification by using the counterexample as evidence. On the other hand, this situation implies that the current inferred model of the implementation is inaccurate. As a result, the counterexample is fed back to the learning algorithm to construct a new better conjecture in the next iteration.

4.2 Preliminary Experiment

In order to demonstrate the feasibility of applying our framework to a practical system, we used a modified version of the Request For Quotation (RFQ) (Kazhamiakin and Pistore, 2006) as a case study. In this experiment, we applied our framework to verify whether the implementation of the seller Web service conforms to the role of seller that was defined by the given choreography model.

4.3 Description

There are three participating services, namely *buyer*, *seller*, and *shipper*, in the RFQ case study. The MSC diagram in Figure 3.1 presents the choreography model that specifies the sequence of interactions among these Web services. First, the buyer accepts

an initial message from its environment to activate the conversation. After that the buyer sends a request requestQuote message to the seller. The seller replies a requestQuoteReply offer back to the buyer. In this situation, the buyer can make a decision to either accept the offer by sending a confirmRequest message, or request a new offer by sending an updateQuote message to the seller. If the seller receives the confirmRequest message from the buyer, the seller will send a shippingOrder request to the shipper. Otherwise, the seller replies an updateQuoteReply to offer a new quotation to the buyer. Finally, when the seller received the shippingOrderReply message, it sends the confirmRequestReply reply to the buyer and the buyer sends a reply to the environment.

4.4 Implementation

We have implemented the seller service using JAX-WS (2012) to handle Web service in Java. Thanks to the help of NetBeans IDE (2012), when we compiled and deployed the seller Web service, the tool automatically generated and installed the Web service and its WSDL file to the application server, e.g., GlassFish (2012). According to the choreography, the implemented seller Web service has three public operations, namely requestQuote, updateQuote, and confirmRequest, which are specified by tag `<operation name>` in the WSDL. However, unlike BPEL, we had to manually handle states and instances of the stateful Web service in JAX-WS. Furthermore, in order to reduce the time used in the verification process, we expect that the implemented Web service should throw an exception whenever an explicit error occurs. For instance, if the requestQuote operation of the seller Web service has already been invoked and the same operation is invoked again, the service should throw an exception immediately.

4.5 Learning Algorithm in Practice

We have also implemented a Learner and a Teacher of ${L_M}^*$ in Java. The Learner is simply implemented with respect to the algorithm. On the other hand, an implementation of a Teacher is quite complicated. For the reason that we have to provide a Teacher that is able to answer both kinds of questions: “membership queries” and “equivalence queries”.

4.5.1 Membership Queries

The membership queries are asked by the Learner in step 1 of the verification process. In order to answer this kind of question, we created the Teacher which is composed of two parts, main part and monitoring part, based on the given WSDL of a targeted Web service.

Suppose we know the set of input symbols and the set of output symbols of the Learner. The main part implements the methods to invoke the public operations of the targeted Web service. When the main part has received the sequence of input symbols from the Learner, it maps each symbol to the public operation, creates a SOAP message corresponding to the operation, and sends the message to the Web service.

Simultaneously, while the main part sends the SOAP message to the implementation, the monitoring part is used to monitor the output that will be sent from it. If the output message is received on time, the Teacher maps the message back to the corresponding output symbol and replies it to the Learner. Otherwise, if either the error message is received or the timeout has expired, the Teacher replies with the symbol Ω. Our implementation of this part was modified from TcpMon (2012).

4.5.2 Equivalence Queries

The conjectures are constructed and presented to the Teacher in steps 2-4 of the verification process. To answer the question, the model checker *Labelled Transition System Analyser* LTSA (Magee and Kramer, 2006) is employed to check the trace equivalence relation between the LTSs of implementation S_i^{Imp} and specification S_i^{Spec}. In step 2, the Mealy machine conjecture is automatically transformed to the corresponding LTS according to Definition 4.1. After that we performed a model checking based on a trace equivalence relation. Step 3 of our framework verifies that all observable traces from the specification have been implemented, i.e., $S_i^{Spec} \leq_{tr} S_i^{Imp}$. Next, in step 4, we verify that $S_i^{Imp} \leq_{tr} S_i^{Spec}$ to guarantee that the actual system has implemented only the traces foreseen by the protocol. If we have a counterexample provided by LTSA from either step 3 or step 4, the Teacher replies with the counterexample. Otherwise, the Teacher replies *"yes"* to the Learner.

4.6 Experimental Result

Given the set of input symbols $I = \{rq, uq, cf, resp\_sh\}$ for requestQuote, updateQuote, confirmRequest, and shippingOrderReply message respectively, and the set of output symbols $O = \{resp\_rq, resp\_uq, resp\_cf, sh, \Omega\}$ for requestQuoteReply, updateQuoteReply, confirmRequestReply, shippingOrder message and the special symbol Ω representing the error message respectively. From step 1, the observable table (S, E, T) is initialized and the algorithm is applied until the table is closed and consistent. Table 4.1, in the previous section, is the closed and consistent table from the first iteration. Figure 4.2 presents the Mealy machine conjecture M^{Imp}_{Seller} from Table 4.1.

After transforming the Mealy machine conjecture to the corresponding LTS model $S^{Imp}_{Seller_{err}}$ (step 2), we checked that every observable traces from the specification have been implemented by verifying $S^{Spec}_{Seller} \parallel S^{Imp}_{Seller_{err}}$ (step 3). Using LTSA, we found that the π state is not reachable.

However, the trace $\sigma = rq \cdot resp\_rq \cdot uq \cdot resp\_uq \cdot uq$, as shown by dotted line in Figure 2.2, is a counterexample from verifying $S^{Spec}_{Seller_{err}} \parallel S^{Imp}_{Seller}$ (step 4) with LTSA. After checking with the implementation of seller Web service, we found that the counterexample is the actual run of the service. As a result, we can conclude that the current implemented seller Web service does not conform to the given choreography specification; moreover, the counterexample trace can be used to refine either the specification or the implementation of the service.

4.7 Summary

Although the notions of conformance between Web service choreography and orchestration has been formally defined in the literatures, the practical limitation of the verification process is that the internal structures of implemented orchestration have to be explicit such as BPEL. In this chapter, we presented a framework for verifying the conformance between choreography specification and the black box implementation of Web service using the learning algorithm $L_M{}^*$ and the model checker LTSA. As an application to a practical system, we conducted an experiment to verify the seller Web service implemented in Java of the RFQ case study. From the experimental result, we were able to detect the execution trace of the seller service that does not conform to the specification.

CHAPTER V

NON-DETERMINISTIC FINITE STATE MACHINES INFERENCE

5.1 Motivation

In the former chapters, we assume that the behaviors of the implementation are deterministic and can be described by DFSMs. On the other hand, *nondeterminism* in specifications and practical applications is not unusual in certain systems that are composed of a number of components that participate concurrently, such as a communication system, a component-based system, and a service-oriented system. Such nondeterminism could arise from the asynchronous communication between different components, as well as from unpredictable activities such as interleaving between components. A *nondeterministic finite state machine* (NFSM) is preferred for specifying such a system in a more neutral manner because it has both an input/output structure and nondeterminism. Even though the NFSM has received much attention in a wide range of test generation methods (see, for example, (AboElFotoh et al., 1993; Hierons, 2004; Ipate, 2006; Luo et al., 1994)), it has received less examination in the automata learning literature. Nevertheless, the relationship between model-based testing and automata learning can be found in (Berg et al., 2005a; Lee and Yannakakis, 1996).

Most of the studies on automata inferencing of nondeterministic machines, both the active and passive approaches, are based on a class of *non-deterministic finite automata* (NFAs). For example, Yokomori (1995) presented an active algorithm for inferring NFAs using *contradiction backtracking*. In (Denis et al., 2000, 2004), the authors introduced a subclass of NFAs, namely *residual finite state automata* (RFSAs), and provided passive learning algorithms for RFSAs that operate in a manner similar to that of the classical RPNI algorithm (Oncina and Garcia, 1992). Moreover, they argued in (Denis et al., 2004) that the resulting NFAs of (Yokomori, 1995) are actually RFSAs. Another subclass of NFAs, called *unambiguous finite automata* (UFAs), was defined and studied in (Coste and Fredouille, 2003). A recent survey and comparisons of passive learning for NFAs can be

found in (García et al., 2008). In addition, for active procedure, the most recent RFSAs' learning algorithm, called NL^*, is presented in (Bollig et al., 2009).

Inferencing in a class of NFSMs, i.e., finite automata with outputs and with nondeterminism, has been studied in (Berg et al., 2006; Shahbaz et al., 2007), for example. The closest idea to our approach can be found in (Shahbaz et al., 2007), in which the authors extend the concept of L^* to a more expressive model called a *parameterized finite state machine* (PFSM), which has nondeterministic structures. For the same deterministic input, i.e., the same input symbol but with *different input parameters*, a PFSM could produce different outputs. In contrast, we consider in this study an NFSM model that, for the identical inputs, i.e., the same input symbol and *the same input parameter*, could produce different outputs. Thus, we have studied a model with a higher degree of nondeterminism.

There are two main reasons as to why we cannot use an algorithm to determinise an NFSM to an equivalent *deterministic* FSM (DFSM), and then apply the DFSM's learning algorithms. First, unlike finite automata, there are NFSMs for which there is no equivalent DFSM (Hierons, 2004). Second, even though some NFSMs can be converted into equivalent DFSMs, the corresponding DFSM could have exponentially more states than the original NFSM, which would be an obstacle in the learning process because its efficiency depends on this factor. Thus, specific learning algorithms for NFSMs are needed.

In this chapter, we propose an active-style learning algorithm called ${L_{NM}}^*$ for NFSM inferencing. To apply the algorithm, an *input/output query* (i/o query) that extends the concept of an output query (Shahbaz and Groz, 2009) is introduced. Because some NFSMs might be partially specified, as studied in (Petrenko and Yevtushenko, 2006) for conformance testing approach, we present an optimization that can optionally be invoked in this specific case of NFSMs. We also provide a time complexity analysis together with a proof of correctness because our algorithm can be terminated by an incorrect conjecture that has more states than the original machine when the *complete testing assumption* (Hierons, 2004) does not hold. We implemented the algorithm and studied its efficiency using a suite of experiments. From the experimental results, we draw conclusions regarding the applicability and scalability of our algorithm, as well as

the effect of our implemented optimization. Moreover, these results allow us to perform a tighter analysis of the worst-case time complexity of $L_{NM}{}^{*}$.

5.2 Inference of Non-deterministic FSMs

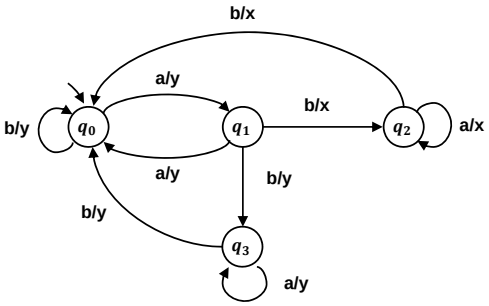

Figure 5.1: Example of a non-deterministic finite state machine.

Even though Angluin's algorithm L^* can efficiently learn an unknown regular language U and produce a minimal DFA that accepts U in polynomial time, its adaptations to FSMs are not obviously efficient, as discussed in (Shahbaz and Groz, 2009). For example, the direct adaptations can be performed through model transformation techniques by mapping from inputs and outputs of the FSM to letters of a DFA's alphabet Σ, such that $\Sigma = I \cup O$ (Hungar et al., 2003) or $\Sigma = I \times O$ (Mäkinen and Systä, 2001). However, these methods are confronted by complexity problems because the cost of L^* is polynomial, based on the size of Σ. Shahbaz and Groz (2009) observed that, by slightly modifying the structure of the observation table and the way in which the counterexample is processed, their proposed method, namely $L_M{}^{+}$, can learn deterministic FSMs, specifically Mealy machines, more effectively.

As usual in L^*-based algorithms' settings, a learning algorithm, called *Learner*, needs to ask two types of questions to a *Minimally Adequate Teacher*, called *Teacher* for short, which is assumed to correctly answer the questions. The first type of question is called a *membership query* in L^*, which consists of a string σ from Σ^*. The Teacher replies either *true* if $\sigma \in U$ or *false* otherwise. Later, this concept is adapted to the *output query* in $L_M{}^{+}$, which consists of a string σ from I^+. The difference is that, instead of true or false, the Teacher of $L_M{}^{+}$ replies with the output string from O^+, which will be processed and recorded in an *observation table*.

The second type of question is called an *equivalence query*, which consists of a

candidate DFA M, whose language the Learner believes to be identical to U (i.e., $\mathcal{L}(M) = U$) in the case of L^*, or a candidate Mealy machine M, whose language the Learner believes to be identical to the language of an unknown Mealy machine M_U (i.e., $\mathcal{L}(M) = \mathcal{L}(M_U)$) in the case of ${L_M}^+$. The answer is true if it is a correct conjecture; otherwise, the Teacher returns a counterexample, which is a string in the symmetric difference of $\mathcal{L}(M)$ and U in L^* or $\mathcal{L}(M)$ and $\mathcal{L}(M_U)$ in ${L_M}^+$.

In our setting, the algorithm asks *input/output queries* (i/o queries) that are input/output sequences $\bar{x}/\bar{y}$, where $\bar{x}$ is in I^* and $\bar{y}$ is in O^*, followed by an input sequence $\bar{z}$ in I^+, and obtains the corresponding answer from the Teacher. This concept is similar to that of the output queries of ${L_M}^+$. However, we modify the Teacher to answer the i/o queries with the output sequences from O^+ that have the output sequence $\bar{y}$ as their prefix. Note that, after each query, the unknown machine must be returned to the initial state by a *reset* input.

Moreover, we need to ask the same i/o query k times for each input/output sequence to observe every possible output sequence from an unknown NFSM (if the complete testing assumption holds for k).

Let $M = (Q, I, O, \delta, q_0)$ be an unknown NFSM that is initially connected, completely specified, observable, and reduced. A detailed description of an observation table and the procedure of ${L_{NM}}^*$ will be described in this section.

5.2.1 Observation Table

At a higher level, the observation table is composed of two parts: an upper and a lower part. Each row in the upper part represents a candidate state of the unknown machine, while each row in the lower part represents the target state of a candidate state and an input. Formally, the structure of an observation table (denoted by (S, E, T)) of the algorithm ${L_{NM}}^*$ consists of three parts: S, E, and T, where

- S is the non-empty finite set of prefix-closed input/output sequences $\bar{x}/\bar{y}$, where $\bar{x} \in I^*, \bar{y} \in O^*$, and S always contains the *empty sequence* ϵ.
- E is the non-empty finite set of suffix-closed input sequences from I^+.

- T is a finite function that maps $(S \cup S \cdot I/O) \times E$ to a set of output sequences from $2^{O^{|E|}}$.

Intuitively, an observation table can be visualized as a two-dimensional array with rows labelled by elements of S and $S \cdot I/O$ (i.e., $S \cup S \cdot I/O$) and columns labelled by elements of E. The entry corresponding to row s in $(S \cup S \cdot I/O)$ and column e in E equals to $T(s, e)$, which contains the set of the output sequences from $\mathit{suff}^{|e|}(\delta_O(q_0, s \cdot e))$, where $\mathit{suff}^k(\mathcal{S})$ denotes the set of k-length suffixes of every sequence from a set $\mathcal{S}$. For example, let $\mathcal{S} = \{y \cdot x \cdot x, y \cdot x \cdot y, y \cdot y \cdot y\}$, $\mathit{suff}^1(\mathcal{S}) = \{x, y\}$ and $\mathit{suff}^2(\mathcal{S}) = \{x \cdot x, x \cdot y, y \cdot y\}$.

Table 5.1: Example of an observation table.

T_1		E	
		a	b
S	ϵ	$\{y\}$	$\{y\}$
$S \cdot I/O$	a/y	$\{y\}$	$\{x, y\}$
	b/y	$\{y\}$	$\{y\}$

Definition 5.1 (Row Equivalence) *Let s, t be two rows in the table (S, E, T), i.e., $s, t \in S \cup S \cdot I/O$. Then, s and t are row equivalent, denoted by $s \cong_E t$, if and only if $T(s, e) = T(t, e)$ for all $e \in E$. Moreover, we used $[s]$ to denote the equivalence class of rows that are row equivalent to s.*

For example, Table 5.1 is an example of the observation table used for learning the NFSM M_0 in Figure 5.1. From this table, the row ϵ is equivalent to b/y (i.e., $\epsilon \cong_E b/y$) but is not equivalent to a/y (i.e., $\epsilon \ncong_E a/y$).

Definition 5.2 (Closed Observation Table) *An observation table is called closed if and only if for each $t \in S \cdot I/O$, there exists an $s \in S$ such that $s \cong_E t$.*

For example, Table 5.1 is not closed because $a/y \in S \cdot I/O$ but $\forall s \in S, s \ncong_E a/y$. However, Table 5.2 is a closed observation table because, for each row t in $S \cdot I/O$, there exists a row s in S such that $s \cong_E t$.

From the closed observation table, we can construct an NFSM conjecture as follows:

Table 5.2: Closed observation table.

T_1		E	
		a	b
S	ϵ	$\{y\}$	$\{y\}$
	a/y	$\{y\}$	$\{x, y\}$
	$a/y \cdot b/x$	$\{x\}$	$\{x\}$
$S \cdot I/O$	b/y	$\{y\}$	$\{y\}$
	$a/y \cdot a/y$	$\{y\}$	$\{y\}$
	$a/y \cdot b/y$	$\{y\}$	$\{y\}$
	$a/y \cdot b/x \cdot a/x$	$\{x\}$	$\{x\}$
	$a/y \cdot b/x \cdot b/x$	$\{y\}$	$\{y\}$

Definition 5.3 (NFSM Conjecture) *Given a closed observation table* (S, E, T)*,* ${L_{NM}}^*$ *obtains an NFSM conjecture* $M = (Q, I, O, \delta, q_0)$*, where*

- $q_0 = [\epsilon]$,
- $Q = \{q_i \mid q_i = [s] \text{ for } 1 \leq i \leq |S| - 1, \forall s \in S \wedge s \neq \epsilon\}$,
- $\delta(q, i) = \{(q', o) \mid q = [s], q' = [s \cdot i/o], \forall s \in S, \forall i \in I, \forall o \in T(s, i)\}$.

The conjecture ${M_0}^{(1)}$ shown in Figure 5.2 is constructed from Table 5.2, which is a closed observation table, according to Definition 5.3.

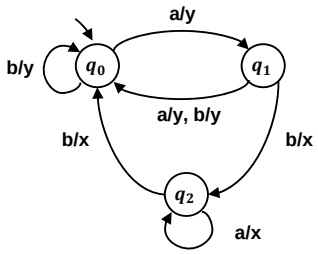

Figure 5.2: The NFSM conjecture ${M_0}^{(1)}$ from Table 5.2.

Theorem 5.4 *Let* (S, E, T) *be a closed (and consistent) observation table, and let* M *be the NFSM conjecture that is constructed from* (S, E, T)*. The conjecture* M *is consistent with the finite function* T*. Any other NFSM that is consistent with* T *but not equivalent to* M *must have more states.*

Proof. Since the observation table (S, E, T) in the setting of ${L_{NM}}^*$ preserves the prefix-closed and suffix-closed properties of S and T, respectively, the conjecture is proven to be consistent with the observation table that has been given by Niese (2003). Moreover,

because the conjecture M is the reduced NFSM by construction, any other NFSM that consistent with T but not equivalent to M must have at least one more state. □

5.2.2 The Algorithm

We now describe $L_{NM}{}^*$, which takes a set of input symbols I and a time to query k as input. Its pseudocode is given in Algorithm 1.

Algorithm 1: The algorithm $L_{NM}{}^*$.

```
   input  : A set of input symbols I, time to query k
   output: NFSM conjecture M
   // Construct the initial observation table (S, E, T)
1  set S = {ϵ}, E = I, and update T using i/o queries ;
2  add ϵ · i/o to S · I/O for all i ∈ I, o ∈ T(ϵ, i), and update T using i/o queries ;
3  repeat
       // Check whether the table is closed
4      while found t ∈ S · I/O such that t ≇_E s, for all s ∈ S do
5          move t to S;
6          add t · i/o to S · I/O for all i ∈ I, o ∈ T(t, i), and update T using i/o queries ;
7      end
8      make the NFSM conjecture M from (S, E, T) ;
9      if the Teacher replies with a counterexample ce then
10         if any prefix of ce has been recorded in T with a different value then terminate ;
11         else
12             find the longest u ∈ S ∪ S · I/O such that ce = u · v ;
13             add the input sequence of v and all of its suffixes to E, and update T using i/o
               queries ;
14         end
15     end
16 until the Teacher replies "yes";
17 return the conjecture M;
```

The algorithm starts by initialising an observation table (S, E, T) with $S = \{\epsilon\}$ and $E = I$. Then, it asks the i/o queries to fill the upper part of the table, i.e., $T(\epsilon, e), \forall e \in E$ (line 1). Next, it uses the observed outputs from the upper part to construct the i/o queries to fill the lower part, namely $S \cdot I/O$, of the table, i.e., $T(\epsilon \cdot i/o, e), \forall i \in I, \forall e \in E, \forall o \in T(\epsilon, i)$ (line 2).

After initialising the table, $L_{NM}{}^*$ repeatedly checks whether the current table is closed (line 4). If it is not closed that means there exists row $t \in S \cdot I/O$ such that $t \ncong_E s$ for all $s \in S$. Then, $L_{NM}{}^*$ finds and moves row t to S (line 5). Next, $t \cdot i/o$ is added to $S \cdot I/O$, and $T(t \cdot i/o, e)$ is determined by the i/o queries for all $i \in I, e \in E, o \in T(t, i)$ (line 6).

When the table is closed, $L_{NM}{}^*$ makes an NFSM conjecture M from the table

according to Definition 5.3 and verifies it with the Teacher by equivalence query (line 8). The Teacher replies either *yes*, acknowledging that the conjecture is correct, or with a counterexample. If the Teacher says *yes*, then ${L_{NM}}^*$ terminates with the correct NFSM M (line 16). Otherwise, the Teacher replies with a counterexample. The counterexample is analysed as to whether it is false (line 10). If it is a false counterexample, then the procedure terminates; otherwise, it will be used for extending the table accordingly (lines $12 - 13$). The method for processing a counterexample will be described in the next subsection. With the extended table, the algorithm repeats the checking loop (lines $4 - 6$) again until the table is closed, followed by making a new conjecture.

Note that, according to the complete testing assumption, for each i/o query, ${L_{NM}}^*$ needs to ask the same query k times to explore every possible output from the machine. Therefore, an answer of an i/o query is a set of output sequences.

5.2.3 Counterexample

To the best of our knowledge, the crucial improvement in the methods for processing counterexamples of the original Angluin's algorithm L^* was proposed by Rivest and Schapire (1993). They observed that the handling of counterexamples as in L^* could lead to *inconsistency* in an observation table (S, E, T). Informally, the table is inconsistent if two (or more) rows in the upper part of the table that represent the same *potential state in the conjecture* have different target states when applied to some inputs. More precisely, $\exists s, t \in S$ and $\exists i \in I$, such that $s \cong_E t$ but $s \cdot i \ncong_E t \cdot i$. This scenario implies that the rows s and t must be distinguished. Fortunately, Rivest and Schapire suggested that, by adding a distinguishing sequence from the counterexample to the set E, inconsistency will never occur. The reason is that the method will never directly add a new row to S, and consequently, the rows in S will remain inequivalent. Furthermore, this condition will always hold trivially. However, the method requires a relaxation on the prefix-closed and suffix-closed properties of the table. For more details and proofs of the method, interested readers can refer to the original paper (Rivest and Schapire, 1993).

In (Shahbaz and Groz, 2009), the authors modified the method for processing the counterexample based on Rivest and Schapire's idea. Their method starts by finding the longest prefix of the counterexample that has already been observed in the table, i.e., $S \cup S \cdot I$. Then, the remaining string and all of its suffixes are added to E. Unlike

the previous methods, the observation table preserves the prefix-closed and suffix-closed properties, and, therefore, the constructed conjecture is proved to be consistent with the table.

Our treatment of counterexamples is adapted straight from (Shahbaz and Groz, 2009). Let ce be the counterexample for the current conjecture. We find the longest prefix $u \in S \cup S \cdot I/O$ of ce such that $ce = u \cdot v$, and $v = \bar{x}/\bar{y}$ is the remaining input/ output sequence of ce. Then, we add the input sequence of v (i.e., $\bar{x}$) and all of its suffixes to E.

We have observed that, when processing a counterexample in this setting, the table preserves the prefix-closed and suffix-closed properties of S and E, respectively. Thus, the output conjecture is proved to be consistent with the observation table.

5.2.4 Correctness

As usual in an active learning procedure, our algorithm asks increasingly longer i/ o queries to the Teacher to observe all of the possible states of an unknown machine, and the corresponding sets of output sequences are recorded in an observation table (S, E, T). According to the structure of the observation table, the set S contains uniquely potential states of the conjecture, and the set E contains the sequences that can be used to distinguish these states from each other. This scenario means that every row in S can be distinguished when applying some $e \in E$. In other words, any rows in the table (i.e., $S \cup S \cdot I/O$) that represent the same state must not be distinguished by any sequences in E.

In the case of a deterministic machine, when the same states are applied by any distinguishing sequences, the machine always responds with the same set of output sequences. However, this scenario is not always the case for a non-deterministic machine. The reason is that if the complete testing assumption does not hold, the Learner may observe a different set of output sequences when the state is applied more than once by the same input/output sequence. This situation could lead the Learner to infer an incorrect conjecture. Nevertheless, the learning procedure will always terminate, which will be proved as follows.

Proposition 5.5 *Suppose that $M_U = (Q, I, O, \delta, q_0)$ is an unknown NFSM. Let q_i be a state in Q, and let $O_{q_i,a}$ be a set of possible outputs of a state q_i under an input a in I. Clearly, $O_{q_i,a} \subseteq O$. Let $L_{q_i,a}$ be the set of all combinations of outputs of the state q_i under the input a. Then, $L_{q_i,a} = 2^{O_{q_i,a}} \setminus \{\emptyset\}$ and $|L_{q_i,a}| = 2^{|O_{q_i,a}|} - 1$.*

Proposition 5.5 claims that the number of possible distinct output sets that can be observed and added to the observation table is finite.

Theorem 5.6 *Given an unknown NFSM $M_U = (Q, I, O, \delta, q_0)$, $L_{NM}{}^*$ will eventually provide a closed table (S, E, T) in each iteration, regardless of the complete testing assumption.*

Proof. Now assume that s is a row in S and t is a row in $S \cdot I/O$ and that they represent the same state q_i in Q. Therefore, $s \cong_E t$, which means that $T(s, e) = T(t, e)$ must hold for all e in E with respect to Definition 5.1. If the complete testing assumption holds, then we know that $\delta_O(q_i, a) = O_{q_i,a}$ for $\forall a \in I$ and $\forall q_i \in Q$. Since e is I^+, we have $T(s, e) = T(t, e) = O_{q_i,e}$. Thus, the table is closed.

In contrast, when the complete testing assumption does not hold, we know that $\delta_O(q_i, a) \subseteq O_{q_i,a}$. Thus, there may exist an e in E such that $T(s, e) \neq T(t, e)$, i.e., the different subsets of $O_{q_i,a}$ have been observed as outputs for $T(s, e)$ and $T(t, e)$. This leads the Learner to consider moving row t, which represents a spurious state, to S. Thus, there are two possible cases, as follows:

- If t is not a new row in S, then the table is now closed.
- Otherwise, row t is moved to S, and the learning process can continue. In this case, the number of the remaining elements in $L_{q_i,a}$ must *decrease* by at least one for each iteration.

By Proposition 5.5, because the set $L_{q_i,a}$ is finite, the maximum number of spurious states for a state q_i for all inputs is bounded by $\sum_{a \in I} |L_{q_i,a}|$, which is also finite. As a result, from (i) and (ii), the learning process eventually provides a closed table. □

Theorem 5.7 *Given an unknown NFSM $M_U = (Q, I, O, \delta, q_0)$, let M be a corresponding conjecture that is constructed from a closed table (S, E, T) in each iteration. When ${L_{NM}}^*$ terminates, if the complete testing assumption holds, then M is guaranteed to be isomorphic with M_U.*

Proof. Theorem 5.6 ensures that ${L_{NM}}^*$ always provides a closed table in each iteration. Whenever the table is closed, the corresponding NFSM conjecture is constructed based on Definition 5.3. Since, by Theorem 5.4, the conjecture M is consistent with the finite function T. For the case in which the complete testing assumption holds, according to the correctness of the Teacher's answer for the equivalence query, we either obtain a counterexample from the conjecture for extending the table, or the learning procedure terminates with a correct conjecture that is isomorphic to M_U. □

Note that, when the complete testing assumption does not hold, there may exist some rows, which represent spurious states, are recorded in the table. Thus, the conjecture M, which is consistent with the table, may have these spurious states. With respect to the correctness of the answer for the equivalence query, if M contains the spurious states, then ${L_{NM}}^*$ terminates. In summary, our algorithm does not necessarily provide an NFSM that is isomorphic to M_U in this case.

5.2.5 Complexity

We analysed a theoretical upper bound for the number of i/o queries asked by ${L_{NM}}^*$. Similar to the membership queries of L^* or the output queries of ${L_M}^+$, the maximum number of i/o queries also corresponds to the worst-case size of the observation table.

Let $|I|$ and $|O|$ be the sizes of the input set I and the output set O, respectively. Let n be the number of states of the NFSM, and let m be the maximum length of any counterexamples that are provided by the Teacher for equivalence queries. The size of the table has at most $n + n|I||O|$ rows (n rows in the upper part + their successors) and $|I| + m(n-1)$ columns because E contains $|I|$ elements initially, and at most m suffixes of the maximum $n-1$ counterexamples are added. In addition, with respect to complete testing assumption, each query must be asked k times to observe every possible output. Thus, ${L_{NM}}^*$ produces a correct conjecture by asking a maximum of

$k(S \cup S \cdot I/O) \times E = O(kn|I|^2|O| + kmn^2|I||O|)$ i/o queries.

5.2.6 Optimization

As mentioned in (Shahbaz and Groz, 2009), reactive systems can be naturally modelled as (non-deterministic) finite state machines. These models are very useful for checking some properties before implementing the system or testing whether the implementation conforms to the specification models. However, these models might come up with *partial* transition relations (Petrenko and Yevtushenko, 2006). To apply our method, one necessary assumption is that the FSM models must be *completely specified*.

Any (non-deterministic) FSM can be transformed into a completely specified FSM by adding a sink state that loops itself for all inputs, i.e., a state that has no outgoing transition to other states, and adding transitions for the missing inputs from any states in the original FSM to the sink state, with a designated error output symbol.

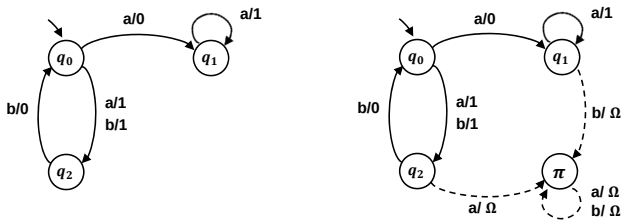

Figure 5.3: A partially specified NFSM (left) and the corresponding completely specified NFSM (right).

Consider an NFSM example, shown in Figure 5.3, in which the input symbols are $\{a, b\}$ and the output symbols are $\{0, 1\}$. Here, an NFSM on the left side is partially specified since it is missing input b of state q_1 and input a of state q_2. Thus, a sink state π is introduced, and new transitions are added between state q_1 under input b and state q_2 under input a to the sink state, as shown in Figure 5.3 (right). In this figure, an error output symbol is represented by Ω.

Thus, if we know that any sequence t will lead the machine to enter the sink state, then every sequence that has t as its prefix will also lead the machine to enter the sink state. We can then use this characteristic to reduce the number of i/o queries asked to the Teacher. Before asking each query, the Learner must first test whether it is an extension of an input/output sequence that has already been observed with an error output. If so,

the Learner can then immediately record the result of the query as an error in the table.

Note that, when we obtain a correct conjecture, which is a completely specified FSM with a sink state, from $L_{NM}{}^*$, it can be transformed back to the original machine easily by removing the sink state and all of the transitions that lead to it.

5.2.7 Example

We illustrate the algorithm $L_{NM}{}^*$ on the NFSM M_0 given in Figure 5.1. The algorithm initializes (S, E, T) with $S = \{\epsilon\}$ and $E = I = \{a, b\}$. Moreover, we set $k = 10$ in this example. Then, $L_{NM}{}^*$ asks the i/o queries to fill the upper part of the table, i.e., $T(\epsilon, a) = y$ and $T(\epsilon, b) = y$. Next, it uses the known outputs to construct the queries to fill the lower part of the table. The initial table is shown in Table 5.1.

When the initial table is filled, $L_{NM}{}^*$ repeatedly tests whether the table is closed. Table 5.1 is not closed because the row a/y in $S \cdot I/O$ is not equivalent to any row in S. Therefore, the algorithm moves the row a/y to S and extends the table by adding $a/y \cdot a/y, a/y \cdot b/x$, and $a/y \cdot b/y$ to $S \cdot I/O$. Then, the queries are constructed for the missing elements of the observation table.

The new table is closed, as shown in Table 5.2; thus, $L_{NM}{}^*$ makes a conjecture $M_0{}^{(1)}$ from it, which is shown in Figure 5.2. Since the conjecture $M_0{}^{(1)}$ is not correct, the Teacher replies with a counterexample *ce*. In this case, we assume that the counterexample *ce* is $a/y \cdot b/y \cdot a/y \cdot b/x$ ($a/y \cdot b/y \cdot a/y \cdot b/x \in \mathcal{L}(M_0{}^{(1)})$ but $a/y \cdot b/y \cdot a/y \cdot b/x \notin \mathcal{L}(M_0)$).

Table 5.3: Processing the counterexample $a/y \cdot b/y \cdot a/y \cdot b/x$ for $M_0{}^{(1)}$.

(a) Adding the suffixes of $a \cdot b$ to E.

T_2		E		
		a	b	$a \cdot b$
S	ϵ	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$
	a/y	$\{y\}$	$\{x, y\}$	$\{y \cdot y\}$
	$a/y \cdot b/x$	$\{x\}$	$\{x\}$	$\{x \cdot x\}$
$S \cdot I/O$	b/y	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$
	$a/y \cdot a/y$	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$
	$a/y \cdot b/y$	$\{y\}$	$\{y\}$	$\{y \cdot y\}$
	$a/y \cdot b/x \cdot a/x$	$\{x\}$	$\{x\}$	$\{x \cdot x\}$
	$a/y \cdot b/x \cdot b/x$	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$

(b) Moving the rows $a/y \cdot b/y$ to S.

T_2		E		
		a	b	$a \cdot b$
S	ϵ	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$
	a/y	$\{y\}$	$\{x, y\}$	$\{y \cdot y\}$
	$a/y \cdot b/x$	$\{x\}$	$\{x\}$	$\{x \cdot x\}$
	$a/y \cdot b/y$	$\{y\}$	$\{y\}$	$\{y \cdot y\}$
$S \cdot I/O$	b/y	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$
	$a/y \cdot a/y$	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$
	$a/y \cdot b/x \cdot a/x$	$\{x\}$	$\{x\}$	$\{x \cdot x\}$
	$a/y \cdot b/x \cdot b/x$	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$
	$a/y \cdot b/y \cdot a/y$	$\{y\}$	$\{y\}$	$\{y \cdot y\}$
	$a/y \cdot b/y \cdot b/y$	$\{y\}$	$\{y\}$	$\{y \cdot x, y \cdot y\}$

According to the method for processing the counterexample, $L_{NM}{}^*$ adds $a \cdot b$, which

is the remaining input sequence of ce, and all of its suffixes, i.e., b and $a \cdot b$, to E, as shown in Table 5.3a. This table is not closed because the row $a/y \cdot b/y$ is not equivalent to any rows in S. Thus, the row $a/y \cdot b/y$ is moved to S, and the table is extended accordingly. The resulting table after filling in the missing elements by asking i/o queries is Table 5.3b.

Next, ${L_{NM}}^*$ checks whether Table 5.3b is closed. This table is closed, so ${L_{NM}}^*$ constructs a new conjecture that is isomorphic to M_0. Thus, the Teacher replies *yes* to this conjecture and ${L_{NM}}^*$ terminates with the correct conjecture as its output. The total number of i/o queries asked by the algorithm during this run is 300.

5.3 Experiments

We have performed a suite of experiments to demonstrate the applicability and scalability of our algorithm in practice. This suite is composed of (i) 9 samples of (partially and completely specified) NFSMs, either inspired by different papers (Hierons, 2004; Miao et al., 2010) or specifically designed, and (ii) random (partially specified) NFSMs with arbitrary sizes for the number of states. Furthermore, we have implemented our algorithm in Java, together with our proposed optimization.

We have also simulated the Teacher to answer the equivalence query by using the model checker *Labelled Transition System Analyser* (LTSA) (Magee and Kramer, 2006). To apply the LTSA, we first transform the NFSM to the corresponding *Labelled Transition System* (LTS). The transformation technique is straightforwardly modified from (Pacharoen et al., 2011). Then, the model checking tool checks the *trace equivalence relation* between two corresponding LTSs of the learned NFSM and the targeted NFSM.

As mentioned in (Berg et al., 2005b), the performance of the Teacher in answering an equivalence query depends on the method that is used to realize it. Thus, the time spent by the equivalence query is disregarded from the measurement. To evaluate the execution time of the algorithm, we measured the total execution time except for the time utilized for the equivalence queries.

The experiments were conducted using a Windows 7 system with an Intel Core i5, 2.67 GHz and 4 GB of memory, and LTSA version 3.0. In addition, the Learner and the Teacher were running on the same machine.

5.3.1 Sample Machines

The first set of experiments was conducted on 9 sample FSMs, 1 DFSM and 8 NFSMs, to evaluate our algorithm. All of the examples are different sizes in terms of the number of states. Moreover, we started with the time to query (k) to 1, and learned 10 times for each machine.

Table 5.4: Runs from 9 sample machines.

Machines	No. of States	k	I/O Queries	EQ	Avg. time (ms)
M1	3	1	14	1	7.2
M2	3	8	144	1	103.1
M3	4	7	140	1	65.9
M4	4	10	260	1	363
M5	4	7	154	1	95.8
M6	6	11	462	2	251.9
M7	6	6	336	3	176.1
M8	7	10	510	2	307.6
M9	8	10	570	2	329.4

Experiences

The learned machines are isomorphic to the original FSMs, as expected. With respect to the number of states of each sample machine (No. of States) and the time to query (k), which can guarantee the complete testing assumption, Table 5.4 shows the experimental results, including the number of used input/output queries (I/O Queries), the number of used equivalence queries (EQ), and the average execution time in milliseconds (Avg. time).

From the table, $L_{NM}{}^{*}$ can be applied with both DFSM (e.g., M1) and NFSM (e.g., M2–M9). In addition, $k = 1$ is obviously sufficient for learning any DFSM. Note that the efficiency of the algorithm not only depends on the number of states and the size of the input alphabet, but it also depends on the value of k. Let us consider machines M2 and M3, which have 3 and 4 states, respectively. Learning M3, which has more states, is expected to require more I/O queries; however, because the value of k for inferring M3 is less than that for inferring M2, learning M2 requires slightly more queries than M3.

5.3.2 Random Arbitrary Machines

Apart from the sample machines, we also performed a second set of experiments on random examples by varying the number of states. Specifically, we generated and learned NFSMs with sizes ranging between 10 and 100 states (in steps of 10), with an input size of 10 and an output size of 5. For each number of states n, we randomly generated 10 NFSMs, which have $n-1$ states plus one sink state, to observe the effectiveness of our optimization.

First, we fixed the time to query to 5. However, we found that we cannot guarantee the complete testing assumption with this value. To compare the scalability of ${L_{NM}}^*$, we varied the number of states of the target machines and fixed the time to query. Thus, in this experiment, we set the time to query to 20, and we leave the topic of how to define this value to be discussed in the next section.

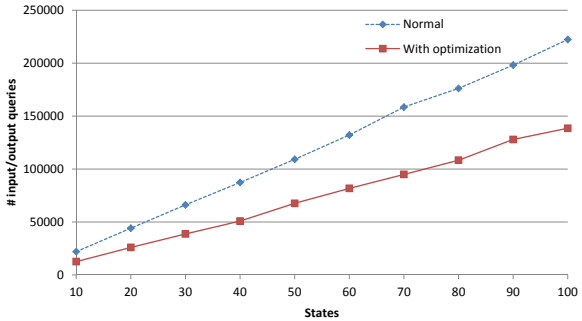

Figure 5.4: Random NFSM examples learned with normal ${L_{NM}}^*$ and with optimization, using $|I| = 10, |O| = 5$, and $k = 20$.

Experiences

We observed that the number of i/o queries relative to the number of states is linear and conforms to the part $kn|I|^2|O|$ in the complexity calculation (see Figure 5.4, in which we vary the number of states but fix the other factors such that $|I| = 10, |O| = 5$ and $k = 20$).

The number of i/o queries is reduced by an average of approximately 39% using the optimized version compared to the basic ${L_{NM}}^*$ algorithm. Moreover, the best reduction that we achieved in this setting was 43% in an NFSM with 10 states. With the specific example of a size of 100 states, the optimized Learner took approximately 10 minutes

with 138,546 i/o queries, a reduction of approximately 38%. The detailed results can be found in Table 5.5, in which it can be seen that the optimized Learner performs better in every case. This scenario might indicate that, for an NFSM with a certain structure, we can make the algorithm perform better through our optimization.

Note that, because the number of equivalence queries in the optimized version does not change from the number in the basic algorithm, we do not report the query in this experiment.

Table 5.5: Comparison of normal ${L_{NM}}^*$ with the optimized version using the random NFSM examples.

No. of States	I/O Queries	I/O Queries (with opt.)	Saved Queries (%)
10	22000	12620	43
20	44200	26020	41
30	66200	38760	41
40	87400	50900	42
50	109200	67640	38
60	132200	81760	38
70	158552	94950	40
80	176202	108366	38
90	198200	127960	35
100	222402	138546	38

Interestingly, when we plotted a graph to study the relationship between the actual number and the theoretical upper bound of the I/O queries, as shown in Figure 5.5, we observed that the part $kn|I|^2|O|$ of the calculated upper bound is far closer to the experimental results than the other part, i.e., $kmn^2|I||O|$. The reason for this similarity is that the Learner in our setting asks few equivalence queries in practice. Thus, a small number of columns will be added to the observation table, i.e., the maximum number of columns is $|I| + \varepsilon$, where ε is a small integer.

As a result, the revised calculation of the worst-case time complexity of the algorithm ${L_{NM}}^*$ in our setting is $O(kn|I|^2|O|)$ since the table has at most $n + n|I||O|$ rows and $|I| + \varepsilon$ columns.

5.4 Discussion and Summary

In this chapter we have presented a novel algorithm for NFSM inference, namely ${L_{NM}}^*$, which uses active-style learning similar to the original L^* algorithm. This algorithm can be applied to both deterministic and non-deterministic FSMs. However, to

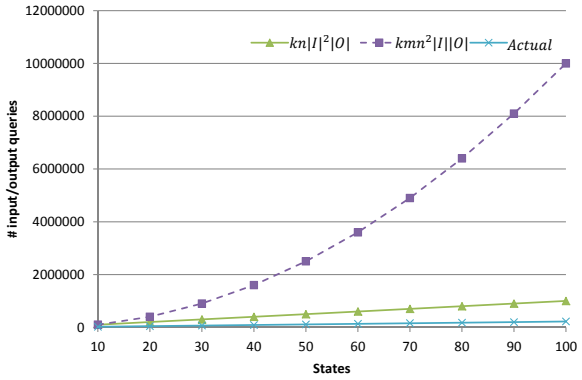

Figure 5.5: Comparison of the actual number of I/O queries of the normal $L_{NM}{}^*$ algorithm and the theoretical upper bound on the random NFSM examples.

infer the correct conjecture, the complete testing assumption must hold in the case of NFSMs. We have calculated the worst-case time complexity of $L_{NM}{}^*$, together with a refinement from the experimental results. Moreover, when $L_{NM}{}^*$ deals with NFSMs that have a particular structure, i.e., partially specified NFSMs, the algorithm offers a faster run by our proposed optimization. From the experimental results, the optimized Learner can reduce the number of i/o queries by approximately 39% on average.

In our setting, we used the LTSA model checker as the Teacher to answer equivalence queries. The results from testing the trace equivalent relation between the two LTSs that represent the learned NFSM and the targeted NFSM can be automatically provided in a short time, e.g., approximately 5 milliseconds for NFSMs with 90 states. Clearly, our method for answering equivalence queries does not work when the prior knowledge of the targeted NFSM is not given. However, it can easily be replaced by other methods, such as a conformance testing approach, with the additional cost of the test suite generation.

To answer the question of how to define the proper value of k to guarantee the complete testing assumption, we usually set the value of k to be a small integer, e.g., $k = |O|$. Because our algorithm is assured to terminate regardless of the complete testing assumption, we eventually obtain a conjecture NFSM. Using the equivalence query, the Learner can receive either the answer *yes* or a counterexample. We found that, if the provided counterexample has already been observed in the table but the recoded value is not the same. This situation means that we cannot explore every possible output of the machine with the current value of k. Therefore, we restart the algorithm with an increased value of k. On the other hand, if the provided counterexample has not been

observed in the table, the counterexample will be used to extend the table, as described in Section 5.2.3. Although this process can be run incrementally, performing incremental steps appears to be inefficient. Thus, it is a challenge to obtain a method for selecting the most appropriate value of k that may not necessarily be minimal but that is sufficient to ensure the complete testing assumption.

CHAPTER VI

CONCLUSION

6.1 Dissertation Summary

In this dissertation, we consider the conformance problem that whether an implementation conforms to the given formal specification. There are a number of attempts to solve this problem; however, the crucial limitation of the traditional techniques in verification process is that the internal structures of implementation have to be explicit. It does not need to be the case, since some implementations can be only observed their external behavior such as third party applications, or the applications implemented by the programming language like Java or .NET.

To solve such a problem, this dissertation proposes a novel method for conformance verification based on a technique so-called automata learning. Inspired by many works in the software verification literature, the automata learning technique is introduced to infer a behavioral model of the black box implementation in our approach. More specifically, we infer the Mealy machine from the implementation using the learning algorithm ${L_M}^*$. By transforming the obtained Mealy machine to the modeling formalism LTS, the model checker LTSA can be used for checking a trace equivalence relation which is the conformance criterion in this work. We also implemented a prototype of our framework based on the Web services composition, and the preliminary experimentation shows promising results.

Moreover, the assumption that the implementations have to be deterministic may be too restricted in some applications, such as a communication system or a component-based system. We present a novel algorithm for NFSM inference, namely ${L_{NM}}^*$, which uses an active-style learning similar to the original L^*. The proposed algorithm can be applied to both deterministic and non-deterministic FSMs. In addition, we have demonstrated the worst-case time complexity analysis and a proof of correctness of ${L_{NM}}^*$. However, to infer the correct conjecture in the case of NFSMs, the complete testing assumption must hold. Furthermore, when ${L_{NM}}^*$ deals with partially specified NFSMs, the algorithm offers a

faster run by our optimization.

6.2 Discussion on Limitations and Future Works

Despite of several benefits, there are some limitations that should be mentioned here. First, since the learning algorithm used in our approach is based on active learning paradigm, it could not be applied to the implementations from which we have only the given observations (e.g., log files).

Second, even though the proposed verification process can guarantee the conformance relation, it may be a time-consuming task which does not appropriate to some applications such as a simple program with a few states. If one may want to test such a system, the lower fault coverage process with lower cost such as some conformance testing methods seems more suitable.

Third, in case of non-deterministic implementation, the time complexity of the proposed learning algorithm ${L_{NM}}^*$ is depend on the factor k. Since the used value of k in this dissertation is derived from the verification process, we have to increase this value several times before we can find the suitable one. Thus it is a challenge to find the heuristic method for selecting a proper value of k that can ensure the complete testing assumption.

Last but not least, this dissertation mainly focusses on the approach for applying formal method to the practical applications. Because the proposed method for conformance verification might not fit with some implementations in other domains, we intend to continue our future research in this direction to obtain further improvements on our results.

6.3 Concluding Remark

Since our conformance verification method can be applied without knowledge of the internal structures of the implementation, we believe that the proposed approach could alleviate the tradition process in software verification. Moreover, the proposed approach and the proposed algorithm in this dissertation would be one prominent instance that promotes the use of formal methods to practical systems.

References

AboElFotoh, H., Abou-Rabia, O., and Ural, H. 1993. A test generation algorithm for systems modelled as non-deterministic FSMs. Software Engineering Journal 8.4 (July 1993): 184 –188.

Angluin, D. 1987. Learning regular sets from queries and counterexamples. Inf. Comput. 75.2 (November 1987): 87–106.

Barker, A., Walton, C. D., and Robertson, D. 2009. Choreographing web services. IEEE Trans. Serv. Comput. 2.2 (April 2009): 152–166.

Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., and Steffen, B. 2005a. On the correspondence between conformance testing and regular inference. In Cerioli, M. (ed.), Fundamental Approaches to Software Engineering, volume 3442 of Lecture Notes in Computer Science, pp. 175–189. : Springer Berlin Heidelberg.

Berg, T., Jonsson, B., Leucker, M., and Saksena, M. 2005b. Insights to Angluin's learning. Electr. Notes Theor. Comput. Sci. (2005): 3–18.

Berg, T., Jonsson, B., and Raffelt, H. 2006. Regular inference for state machines with parameters. In FASE'06, pp. 107–121. :

Bertolino, A. and Polini, A. 2005. The audition framework for testing web services interoperability. In Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO '05, pp. 134–142. Washington, DC, USA: IEEE Computer Society.

Bollig, B., Habermehl, P., Kern, C., and Leucker, M. 2009. Angluin-style learning of NFA. In Proceedings of the 21st international jont conference on Artifical intelligence, IJCAI'09, pp. 1004–1009. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., and Pretschner, A. 2005. Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science). Springer-Verlag New York, Inc., Secaucus, NJ, USA. ISBN 3540262784.

Bucchiarone, A., Melgratti, H., and Severoni, F. 2007. Testing service composition. In Proceedings of the 8th Argentine Symposium on Software Engineering, ASSE '07. :

Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., and Zavattaro, G. 2005. Choreography and orchestration: a synergic approach for system design. In Proceedings of the Third international conference on Service-Oriented Computing, ICSOC'05, pp. 228–240. Berlin, Heidelberg: Springer-Verlag.

Canfora, G. and Penta, M. 2009. Software engineering. chapter Service-Oriented Architectures Testing: A Survey, pp. 78–105. Berlin, Heidelberg: Springer-Verlag.

Chaki, S., Clarke, E., Sharygina, N., and Sinha, N. 2008. Verification of evolving software via component substitutability analysis. Form. Methods Syst. Des. 32.3 (June 2008): 235–266.

Cobleigh, J. M., Giannakopoulou, D., and Păsăreanu, C. S. 2003. Learning assumptions for compositional verification. In Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'03, pp. 331–346. Berlin, Heidelberg: Springer-Verlag.

Coste, F. and Fredouille, D. 2003. Unambiguous automata inference by means of state-merging methods. In Lavrac, N., Gamberger, D., Blockeel, H., and Todorovski, L. (ed.), Machine Learning: ECML 2003, volume 2837 of Lecture Notes in Computer Science, pp. 60–71. : Springer Berlin / Heidelberg.

de la Higuera, C. 2010. Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York, NY, USA. ISBN 0521763169, 9780521763165.

Denis, F., Lemay, A., and Terlutte, A. 2004. Learning regular languages using RFSAs. Theor. Comput. Sci. 313.2 (February 2004): 267–294.

Denis, F., Lemay, A., and Terlutte, A. 2000. Learning regular languages using non deterministic finite automata. In Oliveira, A. (ed.), Grammatical Inference: Algorithms and Applications, volume 1891 of Lecture Notes in Computer Science, pp. 213–214. : Springer Berlin / Heidelberg.

Ferrara, A. 2004. Web services: a process algebra approach. In Proceedings of the 2nd international conference on Service oriented computing, ICSOC '04, pp. 242–251. New York, NY, USA: ACM.

Foster, H., Uchitel, S., Magee, J., and Kramer, J. 2006. Model-based analysis of obligations in web service choreography. In Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services, AICT-ICIW '06, pp. 149–156. Washington, DC, USA: IEEE Computer Society.

Frantzen, L., Las Nieves Huerta, M., Kiss, Z. G., and Wallet, T. 2009. Web services and formal methods. chapter On-The-Fly Model-Based Testing of Web Services with Jambition, pp. 143–157. Berlin, Heidelberg: Springer-Verlag.

Fu, X., Bultan, T., and Su, J. 2005. Synchronizability of conversations among web services. IEEE Transactions on Software Engineering 31 (2005): 1042–1055.

Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. 1991. Test selection based on finite state models. IEEE Trans. Softw. Eng. 17.6 (June 1991): 591–603.

García, P., Parga, M. V., Álvarez, G. I., and Ruiz, J. 2008. Learning regular languages using nondeterministic finite automata. In Proceedings of the 13th international conference on Implementation and Applications of Automata, CIAA '08, pp. 92–101. Berlin, Heidelberg: Springer-Verlag.

GlassFish 2012. GlassFish application server [Online]. Available from: https://glassfish.dev.java.net/ [2012,March].

Gold, E. M. 1978. Complexity of automaton identification from given data. Information and Control 37.3 (1978): 302–320.

Groce, A., Peled, D., and Yannakakis, M. 2002. Adaptive model checking. In Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '02, pp. 357–370. London, UK, UK: Springer-Verlag.

Heckel, R. and Mariani, L. 2005. Automatic conformance testing of web services. In Proceedings of the 8th international conference, held as part of the joint European Conference on Theory and Practice of Software conference on Fundamental Approaches to Software Engineering, FASE'05, pp. 34–48. Berlin, Heidelberg: Springer-Verlag.

Hierons, R. M. 2004. Testing from a nondeterministic finite state machine using adaptive state counting. IEEE Trans. Comput. 53.10 (October 2004): 1330–1342.

Hinz, S., Schmidt, K., and Stahl, C. 2005. Transforming BPEL to Petri nets. In Proceedings of the 3rd international conference on Business Process Management, BPM'05, pp. 220–235. Berlin, Heidelberg: Springer-Verlag.

Hungar, H., Niese, O., and Steffen, B. 2003. Domain-specific optimization in automata learning. In Hunt, J., WarrenA.. and Somenzi, F. (ed.), Computer Aided Verification, volume 2725 of Lecture Notes in Computer Science, pp. 315–327. : Springer Berlin Heidelberg.

Ipate, F. 2006. Bounded sequence testing from non-deterministic finite state machines. In Proceedings of the 18th IFIP TC6/WG6.1 international conference on Testing of Communicating Systems, TestCom'06, pp. 55–70. Berlin, Heidelberg: Springer-Verlag.

JAX-WS 2012. Java API for XML web services (JAX-WS) [Online]. Available from: http://jax-ws.java.net/ [2012,March].

Kazhamiakin, R. and Pistore, M. 2006. Choreography conformance analysis: asynchronous communications and information alignment. In Proceedings of the Third international conference on Web Services and Formal Methods, WS-FM'06, pp. 227–241. Berlin, Heidelberg: Springer-Verlag.

Lee, D. and Yannakakis, M. 1996. Principles and methods of testing finite state machines - a survey. Proceedings of the IEEE 84.8 (1996): 1090–1123.

Li, J., Zhu, H., and Pu, G. 2007. Conformance validation between choreography and orchestration. In Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE '07, pp. 473–482. Washington, DC, USA: IEEE Computer Society.

Lohmann, N. 2008. A feature-complete Petri net semantics for WS-BPEL 2.0. In Proceedings of the 4th international conference on Web services and formal methods, WS-FM'07, pp. 77–91. Berlin, Heidelberg: Springer-Verlag.

Lucchi, R. and Mazzara, M. 2007. A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70.1 (2007): 96–118.

Luo, G., von Bochmann, G., and Petrenko, A. 1994. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. IEEE Trans. Softw. Eng. 20.2 (February 1994): 149–162.

Magee, J. and Kramer, J. 2006. Concurrency: state models & Java programs. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0-470-09355-2.

Mäkinen, E. and Systä, T. 2001. MAS an interactive synthesizer to support behavioral modelling in UML. In Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, pp. 15–24. Washington, DC, USA: IEEE Computer Society.

Miao, H., Liu, P., and Mei, J. 2010. An improved algorithm for building the characterizing set. In Proceedings of the 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, TASE '10, pp. 67–74. Washington, DC, USA: IEEE Computer Society.

Nakajima, S. 2005. Lightweight formal analysis of web service flows. Progress in Informatics 2 (2005): 57–76.

NetBeans 2012. Netbeans IDE homepage [Online]. Available from: http://netbeans.org/ [2012,March].

Niese, O. 2003. An integrated approach to testing complex systems. PhD thesis, University of Dortmund.

Oncina, J. and Garcia, P. 1992. Inferring regular languages in polynomial update time. Pattern Recognition and Image Analysis (1992): 49–61.

Ouyang, C., Verbeek, E., van der Aalst, W. M. P., Breutel, S., Dumas, M., and ter Hofstede, A. H. M. 2007. Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. 67.2-3 (July 2007): 162–198.

Pacharoen, W., Aoki, T., Bhattarakosol, P., and Surarerks, A. 2011. Verifying conformance between web service choreography and implementation using learning and model checking. In Proceedings of the 2011 5th International Conference on New Trends in Information Science and Service Science (NISS), volume 2, pp. 375–381. : IEEE Computer Society.

Peled, D., Vardi, M. Y., and Yannakakis, M. 1999. Black box checking. In Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), FORTE XII / PSTV XIX '99, pp. 225–240. Deventer, The Netherlands, The Netherlands: Kluwer, B.V.

Petrenko, A. and Yevtushenko, N. 2006. Conformance tests as checking experiments for partial nondeterministic FSM. In Grieskamp, W. and Weise, C. (ed.), Formal Approaches to Software Testing, volume 3997 of Lecture Notes in Computer Science, pp. 118–133. : Springer Berlin / Heidelberg.

Rivest, R. and Schapire, R. 1993. Inference of finite automata using homing sequences. In Hanson, S., Remmele, W., and Rivest, R. (ed.), Machine Learning: From Theory to Applications, volume 661 of Lecture Notes in Computer Science, pp. 51–73. : Springer Berlin / Heidelberg.

Salaün, G., Bordeaux, L., and Schaerf, M. 2004. Describing and reasoning on web services using process algebra. In Proceedings of the IEEE International Conference on Web Services, ICWS '04, pp. 43–51. Washington, DC, USA: IEEE Computer Society.

Shahbaz, M. and Groz, R. 2009. Inferring Mealy machines. In Proceedings of the 2nd World Congress on Formal Methods, FM '09, pp. 207–222. Berlin, Heidelberg: Springer-Verlag.

Shahbaz, M., Li, K., and Groz, R. 2007. Learning and integration of parameterized components through testing. In Petrenko, A., Veanes, M., Tretmans, J., and Grieskamp, W. (ed.), Testing of Software and Communicating Systems, volume 4581 of Lecture Notes in Computer Science, pp. 319–334. : Springer Berlin / Heidelberg.

Su, J., Bultan, T., Fu, X., and Zhao, X. 2008. Towards a theory of web service choreographies. In Proceedings of the 4th international conference on Web services and formal methods, WS-FM'07, pp. 1–16. Berlin, Heidelberg: Springer-Verlag.

Tasharofi, S. and Sirjani, M. 2009. Formal modeling and conformance validation for WS-

CDL using Reo and CASM. Electron. Notes Theor. Comput. Sci. 229.2 (July 2009): 155–174.

TcpMon 2012. A tool to monitor traffic on TCP connections (TCPMON) [Online]. Available from: http://java.net/projects/tcpmon/ [2012,March].

Tretmans, J. 2008. Formal methods and testing. chapter Model based testing with labelled transition systems, pp. 1–38. Berlin, Heidelberg: Springer-Verlag.

Yeung, W. L. 2006. Mapping WS-CDL and BPEL into CSP for behavioural specification and verification of web services. In Proceedings of the European Conference on Web Services, ECOWS '06, pp. 297–305. Washington, DC, USA: IEEE Computer Society.

Yokomori, T. 1995. Machine intelligence 13. chapter Learning non-deterministic finite automata from queries and counterexamples, pp. 169–189. New York, NY, USA: Oxford University Press, Inc.

Zaha, J. M., Barros, A., Dumas, M., and ter Hofstede, A. 2006. Let's Dance: a language for service behavior modeling. In Proceedings of the 2006 Confederated international conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I, ODBASE'06/OTM'06, pp. 145–162. Berlin, Heidelberg: Springer-Verlag.

Zhao, X., Yang, H., and Qiu, Z. 2006. Towards the formal model and verification of web service choreography description language. In Proceedings of the Third international conference on Web Services and Formal Methods, WS-FM'06, pp. 273–287. Berlin, Heidelberg: Springer-Verlag.

APPENDIX

APPENDIX

PUBLICATION

During my Ph.D. study, I have published several papers as follows.

International Conference Publications

1. W. Pacharoen, T. Aoki, P. Bhattarakosol, and A. Surarerks, “Verifying conformance between Web service choreography and implementation using learning and model checking.”, in Proceedings of the 5th International Conference on New Trends in Information Science and Service Science (NISS 2011), IEEE Computer Society, pp. 375–381, 2011.
2. W. Pacharoen, T. Aoki, A. Surarerks, and P. Bhattarakosol, “Conformance verification between Web service choreography and implementation using learning and model checking.”, in Proceedings of the of the 18th International Conference on Web Services (ICWS 2011), IEEE Computer Society, pp. 722–723, 2011.

Biography

Warawoot Pacharoen was born in Trat, Thailand, on December, 1980. Then, he moved to Rayong, Thailand, in 1991 and graduated from Rayongwittayakom school in 1999. He received B.Sc. and M.Sc., both in Computer Science, from Chulalongkorn University, Thailand, in 2003 and 2005, respectively. His bachelor degree has been supervised by Asst. Prof. Dr. Pattarasinee Bhattarakosol. His master and doctorate have been under the supervision of Asst. Prof. Dr. Athasit Surarerks and Asst. Prof. Dr. Pattarasinee Bhattarakosol. Since 2003, he has received a grant from the Commission on Higher Education of Thailand under the University Development Commission (UDC) Scholarship to study in Master and Doctoral degree. During Nov. 2009 - Nov. 2010, he visited the School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), Japan, for doing research under the supervision of Assoc. Prof. Dr. Toshiaki Aoki. The scholarship of this one year research was also supported by the UDC Scholarship. His field of interest includes various topics for applying formal method to Software Engineering with emphasis on software verification, software designing, and model checking. He is also educated in the field of Automata Learning.